The Journal of Systems & Software 168 (2020) 110635

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

FCCI: A fuzzy expert system for identifying coincidental correct test cases

Arash Sabbaghi [a], Mohammad Reza Keyvanpour [b,*], Saeed Parsa [c]

[a] Faculty of Computer and Information Technology Engineering, Qazvin Branch, Islamic Azad University, Qazvin, Iran
[b] Department of Computer Engineering, Alzahra University, Tehran, Iran
[c] Department of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

ARTICLE INFO

Article history:
Received 28 September 2019
Received in revised form 21 March 2020
Accepted 6 May 2020
Available online 12 May 2020

Keywords:
Software debugging
Spectrum-based fault localization
Coincidentally correct test cases
Fuzzy expert system

ABSTRACT

Spectrum-based fault localization (SBFL) is a promising approach to reduce the cost of program debugging and there has been a large body of research on introducing effective SBFL techniques. However, performance of these techniques can be adversely affected by the existence of coincidental correct (CC) test cases in the test suites. Such test cases execute the faulty statement but do not cause failures. Given that coincidental correctness is prevalent, it is necessary to precisely identify CC test cases and eliminate their effects from test suites. To do so, in this paper, we propose several important factors to identify CC test cases and model the CC identification process as a decision making system by constructing a fuzzy expert system and proposing a novel fuzzy CC identification method, namely FCCI. FCCI estimates the CC likelihood of passed test cases using the designed fuzzy rules, which effectively correlate the proposed CC identification factors. We evaluated FCCI by conducting extensive experiments on 17 popular and open source subject programs ranging from small- to large-scale containing both artificial and real faults. The experimental results indicate that FCCI successfully improves the accuracy of the CC identification as well as the accuracy of the representative SBFL techniques.

1. Introduction

When a test fails (i.e., running the program under test (PUT) with the corresponding test data leads to a failure), developers require to find the exact location of the corresponding fault in the source code to fix it. This process, which is referred to as fault localization, is a tedious, time-consuming and expensive task in the software debugging process (Vessey, 1985; Wong et al., 2016; de Souza et al., 2016) and given the ever-increasing complexity of the real-world applications, performing it manually is not only prohibitively inefficient but also error-prone (Wong et al., 2016; Goel, 1985; Xie et al., 2013). For this reason, in recent years, various techniques have been proposed to automate fault localization. Among them, spectrum-based fault localization (SBFL) techniques, which exploit the correlations between program failures and the coverage of program elements (e.g., statements or branches), have received much attention and shown promising results Gao et al. (2017), Abreu et al. (2009), Perez et al. (2014), Pearson et al. (2017).

SBFL techniques calculate the suspiciousness of program elements, which reflect their likelihood of being faulty, by leveraging dynamic information from test executions[1] (i.e., program spectrum) as well as the result (i.e., *passed* or *failed*) of each test de Souza et al. (2016). The key idea is that program elements are more suspicious if they correlate strongly with failed tests, and less suspicious if they correlate strongly with passed tests Artzi et al. (2010). After assigning suspiciousness scores to each program element, they are examined from the most suspicious to the least one by developers to find the faulty statements. To reduce the debugging costs, it is desirable to find the faulty statements by examining the least number of program elements.

One of the main factors that can adversely affect the performance of SBFL techniques is the presence of coincidental correct (CC) test cases in the test suite (Masri et al., 2009; Wang et al., 2009; Ball et al., 2003; Hierons, 2006). Such test cases execute faulty statements but do not cause failures. This occurs if after executing the faulty statement, the program has not transitioned into an infection state, or the infection has not propagated to the output (Masri and Assi, 2014; Voas, 1992; Masri and Podgurski, 2009). In many studies (e.g., Masri et al. (2009), Hierons (2006), Masri and Assi (2014), Masri and Podgurski (2009), Masri and Assi

* Corresponding author.
E-mail addresses: a.sabbaghi@qiau.ac.ir (A. Sabbaghi), keyvanpour@alzahra.ac.ir (M.R. Keyvanpour), parsa@iust.ac.ir (S. Parsa).

[1] In this paper, we use the terms "failed/passed execution", "failed/passed test case", and "failed/passed run" interchangeably. We also use "bugs" and "faults" interchangeably.

https://doi.org/10.1016/j.jss.2020.110635
0164-1212/

(2010), Miao et al. (2012), Xue et al. (2014)), it has been shown that coincidental correctness is prevalent and reduces the effectiveness of SBFL techniques by reducing the suspiciousness score of the faulty statements. Therefore, it is necessary to precisely identify CC test cases and eliminate their effects from test suites.

To identify CC test cases, various methods have been proposed in the literature, in which mostly employ the coverage information of the suspicious program elements executed by passed tests (e.g., Masri and Assi (2010), Feyzi and Parsa (2018c)), or utilize the similarity amongst structural execution profile of test cases in a variety of different ways (e.g., Masri and Assi (2014), Miao et al. (2012), Xue et al. (2014), Feyzi and Parsa (2018b), Liu et al. (2019)). Although these methods can help to improve the effectiveness of SBFL techniques to some extent, utilizing these features in isolation as well as neglecting the use of other important factors that can help to increase the chance of identifying CC test cases reduce their accuracy, thereby affecting the validation of SBFL techniques. In fact, a precise CC identification method requires to employ appropriate information and also requires to correlate them together properly.

In this paper, we propose several important CC identification factors that have a direct impact on increasing the likelihood of identifying coincidental correctness, and model the CC identification process as a decision making system. In this regard, we construct a fuzzy expert system and introduce a novel fuzzy CC identification approach, namely FCCI. FCCI first collects a set of information including suspiciousness (Abreu et al., 2009), fault-proneness, and fault-masking scores of program statements, in addition to the similarity of passed test cases to the failed ones to extract the value of the proposed CC identification factors. Then they are fed to the designed fuzzy expert system and FCCI estimates their likelihood of being coincidental correct using the designed fuzzy rules, which effectively correlate different CC identification factors. The estimated CC likelihoods are then used to identify CC test cases.

To our knowledge, this is the first attempt to design a fuzzy expert system for the CC identification problem and also the first one that takes into account some specific characteristics of test executions such as the fault-proneness and fault-masking of their covered statements. Utilizing fuzzy expert system in FCCI is so advantages, since identifying CC test cases encounter with different uncertainties and the fuzziness in the fuzzy expert systems could encapsulate them. The solution to those uncertainties can be obtained through a fuzzy representation of knowledge, uncertain reasoning, and combining various solutions for uncertainty. In fact, the expert rules can well explain the correlation between the relevant CC identification factors and help FCCI to more accurately identify CC test cases.

We evaluated FCCI by conducting extensive experiments using several popular open source subject programs ranging from small- to large-scale containing both artificial and real faults. These subjects are widely used in the literature to compare all sorts of fault localization related techniques. The experimental results show that FCCI outperforms state-of-the-art CC identification methods in terms of accurate identification of CC test cases and also improving the effectiveness of SBFL techniques.

The rest of the paper is organized as follows: Section 2 gives an overview of SBFL techniques and coincidental correctness, reviews the related works, and motivates FCCI. Section 3 describes the details of our proposed fuzzy CC identification approach. The experiment design and results analysis are shown in Section 4. Section 5 discusses threats to validity, and finally, Section 6 concludes the paper and outlines future directions.

$$\begin{array}{c} \\ t_1 \\ t_2 \\ t_3 \\ t_4 \\ \vdots \\ t_m \end{array} \begin{array}{c} \begin{array}{cccccc} s_1 & s_2 & s_3 & \cdots & s_n \end{array} \\ \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1n} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2n} \\ c_{31} & c_{32} & c_{33} & \cdots & c_{3n} \\ c_{41} & c_{42} & c_{43} & \cdots & c_{4n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & c_{m3} & \cdots & c_{mn} \end{bmatrix} \end{array} \qquad \begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ \vdots \\ r_6 \end{bmatrix}$$

(a) (b)

Fig. 1. Data collected for SBFL. (a) Coverage matrix, (b) result vector.

2. Background and related work

In this section, we first describe the background of SBFL techniques and coincidental correctness briefly and then review the existing methods which have been proposed to mitigate the negative impact of coincidental correctness on SBFL. Throughout this section, a motivating example will be analyzed to better show how SBFL techniques work and how coincidental correctness affect their accuracy. We will also utilize this example to motivate our approach and illustrate the limitations of the existent approaches.

2.1. Spectrum-based fault localization

Let us assume that we have a program under test *PUT* that consists of n statements. Also, consider that we have a test suite T that comprises m different test cases, where each test case t_i consists of input parameters I_i and the corresponding desired output O_i. In spectrum-based fault localization, firstly, the coverage information of executing all test cases on the *PUT* as well as their execution results, which can be *passed* or *failed* are collected. After executing t_i on *PUT*, the resulted output would be O'_i. For the purpose of this paper, a passed execution is one where the observed output of *PUT* matches the expected output (i.e., $O_i = O'_i$), and conversely, a failed execution is one where the observed output of *PUT* is different from that which is expected (i.e., $O_i \neq O'_i$). Therefore, the set of test cases in T is split into two disjoint categories: T_p for passed test cases and T_f for failed test cases. As depicted in Fig. 1, the collected data can be represented as a $m * n$ spectra matrix, the so-called coverage matrix (Wong et al., 2014), and a result vector. The coverage matrix and the result vector are binary such that each entry c_{ij} in the matrix is 1 if t_i covers s_j, and 0 if it does not; and each entry r_i in the result vector is 1 if t_i results in failure, and 0 otherwise. SBFL techniques then utilize the coverage matrix and the result vector to calculate the suspiciousness of program elements. The suspiciousness score assigned to each program element reflects its likelihood of being faulty.

There are different SBFL techniques in the literature, in which their main difference is how they contribute the coverage information of program elements obtained by passed and failed tests in the calculation of their suspiciousness scores. For instance, the suspiciousness score calculation formulas of four popular SFBL techniques, namely Tarantula (Jones et al., 2001), Ochiai (Abreu et al., 2006), DStar* (Wong et al., 2014), and Op^2 (Naish et al., 2011) are depicted in Table 1.

In these formulas, s is a statement, N_S and N_F indicate the total number of passed and failed tests respectively, $N_{CS}(s)$ is the number of passed tests that cover s, and $N_{CF}(s)$ means the number of failed tests that cover s. It should be noted that, in addition to the statement coverage, there are other elements such as statement frequency (Shu et al., 2016), call sequences (Dallmeier

Table 1
Four popular SFBL techniques.

Technique	Formula
Tarantula	$Suspiciousness_{Tarantula}(s) = \frac{N_{CF}(s)/N_F}{(N_{CF}(s)/N_F)+(N_{CS}(s)/N_S)}$
Ochiai	$Suspiciousness_{Ochiai}(s) = \frac{N_{CF}(s)}{\sqrt{(N_F \times N_{CF}(s))+(N_F \times N_{CS}(s))}}$
DStar*[a]	$Suspiciousness_{DStar}(s) = \frac{N_{CF}(s)^*}{(N_F - N_{CF}(s))+N_{CS}(s)}$
Op[a]	$Suspiciousness_{Op^2}(s) = N_{CF}(s) - \frac{N_{CS}(s)}{N_S+1}$

[a]We used $*=2$ and $*=3$, the most thoroughly-explored values for parameter $*$. Therefore, in the rest of this paper, we use DStar^2 and DStar^{3}to replace DStar*.

et al., 2005) and def-use pairs (Masri, 2010), which are employed to construct coverage information of test cases. In this paper, we consider statement coverage in the implementation of SBFL techniques.

After assigning suspiciousness of program elements, they are ranked in descending order according to their suspiciousness scores and are examined one-by-one by the developers starting from the top of the ranking until a faulty statement is identified. To reduce debugging costs, it is desirable to identify the faulty statements by examining the least number of program elements. The schematic of the spectrum-based fault localization is depicted in Fig. 2.

To better illustrate how SBFL techniques work, consider the *TestMe* procedure in Fig. 3. This program has four input variables (*flag*, *a*, *b*, *choice*). If the value of *flag* is negative, it calculates the absolute difference between *a* and *b*. Otherwise, the program performs some arithmetic operations according to the values of *a*, *b* and *choice*. We have seeded two different faults in the program in statements s_4 and s_6, which may generate an incorrect output. In the first faulty statement (i.e., s_4), the programmer mistakenly has written $a+b$ instead of $a-b$. In the second faulty statement (i.e., s_6), a/b is written instead of $a\%b$. As shown in Fig. 3, the test suite contains 11 test cases named t_1 to t_{11}, in which satisfies branch coverage testing criteria Zhu et al. (1997), Baluda et al. (2016). Among these test cases, t_1, t_7, t_8, and t_9 are failed ones and the residual are verified as passed test cases. The coverage information for each test case is also included, where the presence of "●" mark in the cells indicate the coverage of the corresponding statement.

Table 2 represents the suspiciousness score and rank of each program statement obtained using Ochiai, Op^2, DStar^2, and DStar^3 formulas for the code snippet shown in Fig. 3. It has been shown that they are among the best performing SBFL formulas in locating faults (Wong et al., 2016; Xie et al., 2013; Pearson et al., 2017; Wong et al., 2014; Tang et al., 2017; Le et al., 2013). As can be seen, the rank of faulty statement s_4, calculated by Ochiai, Op^2, DStar^2, and DStar^3 are 5, 2, 3 and 3, respectively. Also, the faulty statement s_6 is ranked 10 by all of these SBFL techniques.

2.2. Coincidental correctness

Although the performance of SBFL techniques is encouraging, in practice, their effectiveness may be adversely affected due to the coincidental correctness (Masri et al., 2009; Wang et al., 2009; Ball et al., 2003; Masri and Podgurski, 2009). Coincidental correct (CC) test cases execute faulty statements but do not cause failures and are labeled as *passed* consequently. According to the Propagation–Infection–Execution (PIE) model (Voas, 1992), for the occurrence of failure, three conditions must be satisfied: the faulty statement is executed, the program is transitioned into an infection state, and the infection is propagated to the output. Therefore, coincidental correctness can occur if after executing the faulty statement the program does not transition into an infectious state, or the program is changed into an infectious state, but the infection does not propagate to the output (Masri and Assi, 2014; Androutsopoulos et al., 2014). The former case results in weak CC test cases and the latter case, which is also referred to as Failed Error Propagation (FEP) (Androutsopoulos et al., 2014; Clark et al., 2019), results in strong CC test cases (Masri and Assi, 2014). Accordingly, in our illustrative example depicted in Fig. 3, the five passed test cases t_2 to t_6 are actually coincidentally correct. For example, although test case t_2 executes the faulty statement s_4, it does not produce an infectious state which results in the expected/correct output.

It has been shown that both weak and strong coincidental correctness are prevalent and are a safety reducing factor for SBFL (Masri et al., 2009; Hierons, 2006; Masri and Assi, 2014; Masri and Podgurski, 2009; Masri and Assi, 2010; Miao et al., 2012; Xue et al., 2014). In the following, we empirically show the negative impact of CC test cases on SBFL formulas using Table 3. This table lists the suspiciousness score and also the rank of all program statements of the *TestMe* procedure generated by the same SBFL formulas in two situations: with and without CC test cases. It should be noted that we removed the impact of CC test cases from the test suite by relabeling all CC test cases from *passed* to *failed*.

Table 3 clearly shows that how removing the effects of CC test cases from the test suite improves the performance of Ochiai, Op^2, DStar^2, and DStar^3 formulas. In this example, after relabeling CC test cases, all techniques assigned the rank of 1 to the faulty statement s_4.

So, according to the above-mentioned descriptions, to facilitate the debugging process, it is of great importance to deal with coincidental correctness carefully. In the next section, we provide a review of the related works and highlight gaps, strengths, and weaknesses.

2.3. Related works

There is a large body of work on reducing the vulnerability of SBFL techniques to coincidental correctness. These studies can be categorized into two major groups.

Studies in the first group propose different approaches to improve the effectiveness of SBFL techniques in the presence of coincidentally correct test cases in different ways such as adapting suspiciousness score calculation formulas (Bandyopadhyay, 2012; Bandyopadhyay and Ghosh, 2011; Zhou et al., 2015) or refining the execution profile of test cases (Wang et al., 2009). For example, Bandyopadhyay (2012), Bandyopadhyay and Ghosh (2011) replaced $N_{CS}(s)$ in the Ochiai formula depicted in Table 1 with the total weight of passed test cases that execute *s*. The weight of each passed test case is assigned based upon the proximity of its covered statements with that of the failed ones, such that the coincidentally correct test cases obtain low weights. To this end, they employed code coverage based proximity measure (CC-proximity) (Liu et al., 2008). Given the sets of statements S and S' covered respectively by test cases$t$ and t', the CC-proximity between t and t' is obtained by the expression $\frac{|S \cap S'|}{|S \cup S'|}$. Zhou et al. (2015) also defined a new suspiciousness metric by changing the calculation of the values of some variables in the Tarantula formula shown in Table 1 with coincidental correctness probability of passed runs. They estimate CCP for each passed run by employing dynamic control- and data-flow analysis. To calculate the CC probability of passed runs, a supervised machine learning algorithm, KNN, is employed by Li et al. (2016) and Liu et al. (2019). In this approach, the CC probability value of each passed test case p is calculated according to the distance of p with its top K nearest failed or proven CC test cases. In a single fault program, a proven CC test case is the one that executes all statements in the statement set covered by all failed test cases. Finally, the

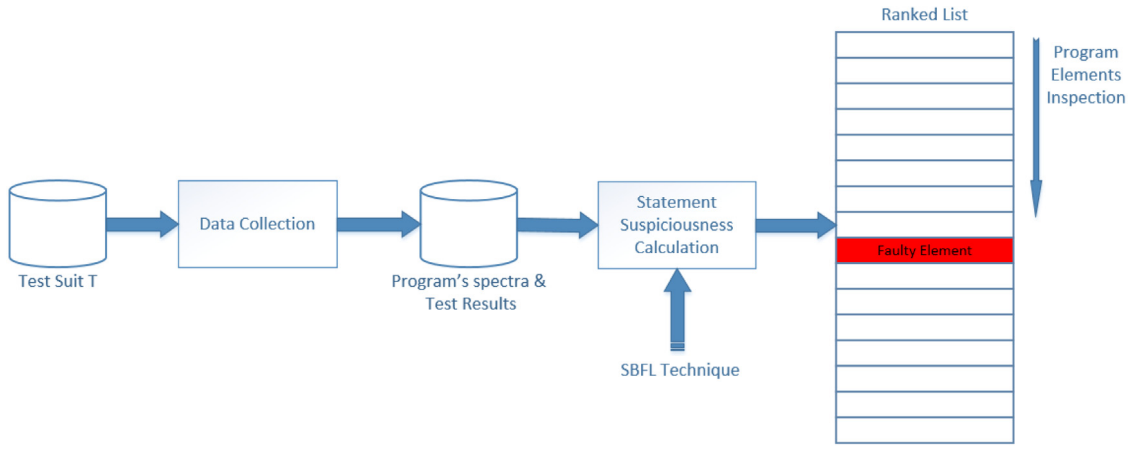

Fig. 2. The schematic of spectrum-based fault localization.

I_1	I_2	I_3	I_4	I_5	
a = 43 b = -22 flag = true choice = ""	a = 12 b = 0 flag = true choice = ""	a = 57 b = 1 flag = true choice = "Summation"	a = 51 b = 15 flag = true choice = "Minus"	a = 41 b = 9 flag = true choice = "Division"	
I_6	I_7	I_8	I_9	I_{10}	I_{11}
a = 31 b = 0 flag = true choice = "Division"	a = 25 b = 6 flag = true choice = "Multiplication"	a = 12 b = 36 flag = true choice = "Average"	a = 18 b = 17 flag = true choice = ""	a = 2 b = 38 flag = false choice = ""	a = 5 b = 3 flag = false choice = ""

S#		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
	`void TestMe(bool flag, int a, int b, string choice)`											
1	`if (flag) {`	●	●	●	●	●	●	●	●	●	●	●
2	`  if (a > 5)`	●	●	●	●	●	●	●	●	●		
3	`  a = a + 4;`	●		●	●	●	●	●	●	●		
4	`  if (a + b < 30) {  // Correct: a - b < 30`	●	●	●	●	●	●	●	●	●		
5	`  if (b != 0)`	●	●									
6	`  Console.Write(a / b);  // Correct: Console.Write(a % b);`	●										
7	`  else Console.Write("Invalid Input");}`		●									
8	`  else { if (choice == "Summation")`			●	●	●	●	●	●	●		
9	`  Console.Write(a + b);`			●								
10	`  else if (choice == "Minus")`				●	●	●	●	●	●		
11	`  Console.Write(a - b);`				●							
12	`  else if (choice == "division") {`					●	●	●	●	●		
13	`  if (b != 0)`					●	●					
14	`  Console.Write(a / b);`					●						
15	`  else Console.Write("Division by Zero");}`						●					
16	`  else if (choice == "Multiplication")`							●	●	●		
17	`  Console.Write(a * b);`							●				
18	`  else if (choice == "Average")`								●	●		
19	`  Console.Write((a + b) / 2);`								●			
20	`  else Console.Write("Wrong Choice!");}}`									●		
21	`else { if (a < b)`										●	●
22	`  Console.Write(b - a);`										●	
23	`  else Console.Write(a - b); }}`											●
Verified Test Results		F	P	P	P	P	P	F	F	F	P	P

Fig. 3. A sample program with two seeded faults and a set of eleven test cases.

calculated CC probability values are incorporated into the DStar[3] formula (Wong et al., 2014) to improve the effectiveness of SBFL. It should be noted that the effectiveness of this method is highly dependent on finding a sufficient number of proven CC test cases, and this may reduce its generalizability.

Wang et al. (2009) propose to refine the code coverage of test runs (i.e., structural execution profiles) in order to ease the coincidental correctness problem in fault localization techniques. Coverage refinement is performed using control- and data-flow patterns, which should be derived for common fault types by capturing the mechanism of how faults trigger failures. The conjecture is that the coverage refinement using these patterns can strengthen the correlation between program failures and the coverage of faulty program elements, which leads to an increase in the effectiveness of SBFL techniques. These patterns are specified using event expression (Bates, 1995) and then translated into extended finite state machines (EFSMs) (Sabbaghi and Keyvanpour, 2017) to perform pattern matching. Their empirical results

Table 2
The suspiciousness score and rank list of statements for the code snippet depicted in Fig. 3 obtained using Ochiai, Op^2, $DStar^2$, and $DStar^3$ formulas.

S#	$N_F(s)$	$N_S(s)$	$N_{CF}(s)$	$N_{CS}(s)$	Ochiai		Op^2		$DStar^2$		$DStar^3$	
					Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank(s)
1	4	7	4	7	0.60	7	3.12	4	2.28	6	9.14	5
2	4	7	4	5	0.66	5	3.37	2	3.20	3	12.8	3
3	4	7	4	4	0.70	2	3.5	1	4.00	2	16	2
4	4	7	4	5	0.66	5	3.37	2	3.20	3	12.8	3
5	4	7	1	1	0.35	14	0.87	14	0.25	14	0.25	14
6	4	7	1	0	0.50	10	1	10	0.33	10	0.33	10
7	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
8	4	7	3	4	0.56	9	2.5	8	1.80	9	5.4	8
9	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
10	4	7	3	3	0.61	8	2.62	7	2.25	7	6.75	7
11	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
12	4	7	3	2	0.67	4	2.75	6	3.00	5	9	6
13	4	7	0	2	0.00	15	-0.25	22	0.00	15	0	15
14	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
15	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
16	4	7	3	0	0.86	1	3	5	9.00	1	27	1
17	4	7	1	0	0.50	10	1	10	0.33	10	0.33	10
18	4	7	2	0	0.70	2	2	9	2.00	8	4	9
19	4	7	1	0	0.50	10	1	10	0.33	10	0.33	10
20	4	7	1	0	0.50	10	1	10	0.33	10	0.33	10
21	4	7	0	2	0.00	15	-0.25	22	0.00	15	0	15
22	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15
23	4	7	0	1	0.00	15	-0.12	15	0.00	15	0	15

Table 3
The suspiciousness score and rank list of statements for the code snippet depicted in Fig. 3 obtained using Ochiai, Op^2, $DStar^2$, and $DStar^3$ formulas in two situations: with and without CC test cases.

S#	Ochiai				Op^2				$DStar^2$				$DStar^3$			
	With CC		Without CC		With CC		Without CC		With CC		Without CC		With CC		Without CC	
	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank	Susp(s)	Rank
1	0.60	7	0.90	4	3.12	4	5	4	2.28	6	40.5	4	9.14	5	364.5	4
2	0.66	5	1	1	3.37	2	7	1	3.20	3	+∞	1	12.8	3	+∞	1
3	0.70	2	0.94	3	3.5	1	6	3	4.00	2	64	3	16	2	512	3
4	0.66	5	1	1	3.37	2	7	1	3.20	3	+∞	1	12.8	3	+∞	1
5	0.35	14	0.47	9	0.87	14	0	9	0.25	14	0.57	9	0.25	14	1.14	9
6	0.50	10	0.33	12	1	10	-1	12	0.33	10	0.12	12	0.33	10	0.12	12
7	0.00	15	0.33	12	-0.12	15	-1	12	0.00	15	0.12	12	0	15	0.12	12
8	0.56	9	0.88	5	2.5	8	5	4	1.80	9	24.5	5	5.4	8	171.5	5
9	0.00	15	0.33	12	-0.12	15	-1	12	0.00	15	0.12	10	0	15	0.12	12
10	0.61	8	0.81	6	2.62	7	4	6	2.25	7	12	6	6.75	7	72	6
11	0.00	15	0.33	12	-0.12	15	-1	12	0.00	15	0.12	12	0	15	0.12	12
12	0.67	4	0.74	7	2.75	6	3	7	3.00	5	6.25	7	9	6	31.25	7
13	0.00	15	0.47	9	-0.25	22	0	9	0.00	15	0.57	9	0	15	1.14	9
14	0.00	15	0.33	12	-0.12	15	-1	12	0.00	15	0.12	12	0	15	0.12	12
15	0.00	15	0.33	12	-0.12	15	-1	12	0.00	15	0.12	12	0	15	0.12	12
16	0.86	1	0.57	8	3	5	1	8	9.00	1	1.5	8	27	1	4.5	8
17	0.50	10	0.33	12	1	10	-1	12	0.33	10	0.12	12	0.33	10	0.12	12
18	0.70	2	0.47	9	2	9	0	9	2.00	8	0.57	9	4	9	1.14	9
19	0.50	10	0.33	10	1	10	-1	12	0.33	10	0.12	12	0.33	10	0.12	12
20	0.50	10	0.33	10	1	10	-1	12	0.33	10	0.12	12	0.33	10	0.12	12
21	0.00	15	0	15	-0.25	22	-4	23	0.00	15	0	20	0	15	0	21
22	0.00	15	0	15	-0.12	15	-3	21	0.00	15	0	20	0	15	0	22
23	0.00	15	0	15	-0.12	15	-3	22	0.00	15	0	20	0	15	0	23

demonstrate that this approach cannot improve the effectiveness of SFBL when coincidental correctness occurs in less than 20% of the total passed test runs. Also, the effectiveness and efficiency of this approach are strongly correlated to the maximal repetition count of Kleene closures.

Since the aforementioned techniques try to indirectly mitigate the negative impact of coincidental correctness on SBFL, their usage is limited to some specific cases or some specific SBFL techniques. It is worth noting that in Metallaxis-FL (Papadakis and Le Traon, 2015) Papadakis et al. utilize mutation analysis, which

is a fault-based technique, to assist the fault localization process. This so-called mutation-based fault localization (MBFL) approach introduces some defects named mutants for each statement s in the PUT based on simple syntactic rules, called mutation operators. Then each mutant e is assigned a suspiciousness score using the modified Ochiai formula Eq. (1), where $N_{KF}(e)$ represents the number of test cases that kill the mutant e and fail on the original program, and $N_{KS}(e)$ represents the number of test cases that kill the mutant e and pass on the original program. A test case kills a mutant if executing the test on the mutant results in a different test outcome than executing it on the original program.

$$Suspiciousness_{Ochiai}(e) = \frac{N_{KF}(e)}{\sqrt{(N_F \times N_{KF}(e)) + (N_F \times N_{KS}(e))}} \quad (1)$$

Finally, in this approach, the suspiciousness of a statement is equal to the maximum suspiciousness value of its respective mutants. In this way, instead of trying to associate the execution of program statements with failures, Metallaxis-FL tries to associate the killing of mutants with test failures or passes. The experimental results of this study show that Metallaxis-FL can effectively handle the coincidental correctness problem in fault localization. This is attributed to the infection propagation requirement, that is, mutants will be killed if the discrepancies introduced by them propagate to the output (Papadakis and Le Traon, 2015). However, mutation-based fault localization is computationally intensive, because a vast number of mutants have to be generated, and it needs to execute the entire test suite many times (once per mutant). To alleviate this problem, different mutant reduction techniques such as *selective mutation* (Papadakis and Le Traon, 2014), which perform reduction by selecting mutation operators, and *mutant sampling* (Papadakis and Le Traon, 2015), which sample mutants directly, have been suggested. But, the mutant reduction strategies may result in losing the precision of fault localization (Papadakis and Le Traon, 2014). Besides, even with these optimizations, MBFL techniques are still very time-consuming and costly, in which their run-time is several orders of magnitude larger than for SBFL techniques (Pearson et al., 2017). After all, although mutation-based fault localization is reported to outperform SBFL on artificial bugs (Papadakis and Le Traon, 2015; Moon et al., 2014), the experimental results of Pearson et al. (2017) show that MBFL performs poorly on real faults and is defeated by SBFL techniques. One of the reasons may be that some of the real faults involve non-mutable statements (e.g., break, continue, return).

Unlike the aforementioned techniques, studies in the second group utilize various types of information to directly identify a subset of passed test cases that are likely to be coincidentally correct and then try to remove their adverse effects on SBFL techniques either by cleansing or relabeling strategies. In the cleansing strategy, the identified CC test cases are removed from the test suite, while in the relabeling strategy, their labels are changed from *passed* to *failed*. Afterward, the suspiciousness score of statements are recalculated.

In this regard, Masri and Assi (2010) propose to first identify a set of program elements, called CCE, that are likely to be correlated with coincidentally correct tests. Each CCE's member (cc_e) is a program element that is executed by all failed runs and by a non-zero but not excessively large percentage of passed runs, which is specified using a threshold value θ. Then any passed test that executes one or more cc_e's are considered as a potential coincidentally correct test (cc_t). In order to reduce the high false positive rate of this approach, instead of choosing all cc_t's as coincidental correct, they propose to select a subset of them by estimating their likelihood of being coincidentally correct based on the following measure: ((average suspiciousness of the covered cc_e's) + (percent of cc_e's covered)). In this measure, the value of the first term varies between 0.5 and 1, while the second term ranges from 0 to 100. In this way, the coverage ratio of cc_e's is the main determinant of the Masri's measure. Feyzi and Parsa (2018c) have also utilized the average suspiciousness score and coverage ratio of high suspicious statements to identify CC test cases and give equal weight for each. They also merely consider statements included in the intersection of expanded backward dynamic slices of failed executions.

The similarity between execution profiles of passed and failed test cases is another factor that has been used in several studies (e.g., Masri and Assi (2014, 2010), Miao et al. (2012), Xue et al. (2014), Feyzi and Parsa (2018b), Miao et al. (2013), Li and Liu (2012, 2014), Yang et al. (2015)) to identify coincidentally correct test cases. According to the empirical observations, failed tests caused by the same fault tend to share some similarities and are more likely to have similar execution profiles (Dickinson et al., 2001; Podgurski et al., 1999; Ammann and Knight, 1988). Therefore, it is expected that passed test cases with higher similarity to the failed ones to be coincidental correct with a higher likelihood. To apply similarity metric to the CC identification problem, mostly, clustering techniques (e.g., Masri and Assi (2014, 2010), Miao et al. (2012, 2013), Li and Liu (2012, 2014), Yang et al. (2015)) and support vector machines (SVM) (e.g. Xue et al. (2014), Feyzi and Parsa (2018b)) are employed.

In clustering-based techniques, the structural execution profile of test cases is given as their features to the cluster analysis (Singh and Chauhan; Han et al., 2011) and it groups test cases into different clusters. Then according to a sampling strategy, CC test cases are selected. The key rationale in using cluster analysis is that most of test cases in the same cluster behave similarly (Miao et al., 2012; Dickinson et al., 2001; Yan et al., 2010) and this can be a clue to identify coincidentally correct test cases. These techniques mostly employ K-means (Hartigan and Wong, 1979; MacQueen, 1967) as the clustering algorithm, and differ in the way of defining execution profile of test cases, the number of clusters and also the sampling strategy.

Miao et al. (2012, 2013) consider statement coverage information of test cases as their execution profile and consider all passed test cases which are grouped into the same cluster with the failed ones as coincidentally correct. They also set the number of clusters according to the size of test suite T, that is $K = L \times |T|$, where $0 < L < 1$. Li and Liu (2012) consider the execution profile of test cases as the number of *true* and *false* evaluations of program predicates in test runs. Their sampling strategy is also based on the suspiciousness of statements within each cluster, which selects the most suspicious clusters as the final results. In this approach, the number of clusters is set according to the number of program functions with the rationale that similar test cases may exercise similar functions. They extend their work in Li and Liu (2014) and employ k-means++ (Arthur and Vassilvitskii, 2007) as the clustering algorithm. Besides, they propose an adaptive sampling strategy for selecting CC test cases from the resulted clusters that contain failed test cases. A clustering approach is also proposed by Yang et al. (2015), which identifies CC test cases regressively.

Masri and Assi (2010) propose another technique to identify coincidentally correct test cases from cc_t's by clustering them into two clusters. Between these two clusters, test cases in the cluster with a higher average Tarantula suspiciousness score of cc_e's are chosen as coincidentally correct. In case of a tie, the larger cluster is selected. A similar clustering-based technique is also proposed by Masri and Assi (2014). They discard program elements that are not cc_e's from the execution profiles, cluster the whole test suite into two clusters and pick a cluster that contains the majority of the failed test cases. All passed test cases within this cluster are marked as coincidentally correct. The limitations

of these techniques which depend on the identification of cc_e's are twofold. First, the user should specify a value for θ, which restricts their automation, and more importantly, there is no unique value for θ that works best for all benchmarks.

Masri et al. (2012) used multivariate visualization scatter plots (Leon et al., 2000) and in particular multidimensional scaling (Leon et al., 2000; Borg and Groenen, 2003; Leon et al., 2005) to produce a visual representation of test data. The representation is a scatter plot of points (i.e., test cases in the form of execution profile), such that points corresponding to similar test cases are placed close to each other and the points corresponding to dissimilar test cases are localized far apart. Finally, in this approach, the user has been given the opportunity to manually select the CC test cases based on this presentation.

Although clustering-based techniques are widely used to identify CC test cases, they are hampered by the challenging task of selecting the number of clusters. On one hand, if the number is set too small then dissimilar objects may be put together in a same cluster. On the other hand, if the number is set too large then similar objects may be put into different clusters (Shafeeq and Hareesha, 2012). According to the sampling strategy, this may increase the rate of false positives and/or false negatives. For example, Miao et al. (2012, 2013) empirically demonstrate that a larger number of clusters yields a higher rate of false negatives but a lower rate of false positives. In general, it can be concluded that there is no fixed number of clusters that perform well for all cases, and this may lead to the inaccuracy of these techniques. In contrast, supervised machine learning algorithms could avoid the impact of uncertain classification number (Kotsiantis et al., 2007) and treat CC test cases as mislabeled data whose test results should be flipped from *passed* into *failed*. In this regard, Xue et al. (2014) adapt ensemble-based support vector machines Kim et al. (2002) to identify CC tests. In this study, test cases for the training phase include all failed test cases along with a small subset of passed ones that are randomly selected from the test suite. Feyzi and Parsa (2018b) also utilized SVM with a customized kernel function. To build the kernel function, a sequence matching algorithm (Parsa and Naree, 2012), which measures the similarities between passed and failed executions is proposed. The main drawback of these SVM-based CC test case identification methods is that they randomly select passed test cases as labeled data without considering the true CC test cases, and this can reduce the accuracy of the classifier.

According to the above discussion, we can see most of the CC identification methods employ the coverage information of the suspicious program elements executed by passed tests, or utilize the similarity amongst execution profile of test cases. Using these features in isolation and also neglecting other important factors, lead to a decrease in the effectiveness of these methods.

2.4. Motivating study

To motivate our approach, we come back to the illustrative example depicted in Fig. 3. As mentioned earlier, there are five CC test cases (i.e., t_2 to t_6) available in the test suite which should be identified. In the following, we investigate the effectiveness of FCCI and also the state-of-the-art approaches in identifying CC test cases including Masri and Assi (2010) and Feyzi and Parsa (2018c) methods, as well as two clustering-based techniques proposed by Miao et al. (2012) and Yang et al. (2015) with different number of clusters. Table 4 reports the output of these techniques, in which the nine rows at the bottom give the execution result of each test case after identifying CC test cases using the corresponding CC identification method and performing the relabeling strategy. The green and red cells represent correct and incorrect predictions, respectively.

As can be seen, clustering techniques with different configurations suffer from a high ratio of false positives and/or false negatives. For example, three of them mistakenly determined t_{10} and t_{11} as CC test cases because of their structural similarity to t_1. Besides, they have several false negative predictions. This shows that merely utilizing similarity between executions is not so advantageous. Also, Masri and Feizi methods were unable to identify CC test case t_2, which has been identified with almost all clustering-based techniques. This indicates that using the coverage information of the suspicious program elements is not enough to precisely identify CC test cases. On the other side, by properly employing and correlating different types of information, FCCI was able to accurately identify CC test cases.

Next, we empirically show how incorrect prediction of CC test cases can harm the performance of SBFL techniques. Table 5 reports the statement suspiciousness scores and output ranking of Ochiai, Op^2, and $DStar^3$ before and after identifying CC test cases using the aforementioned CC identification methods. As can be seen, FCCI has led to further improvement in the performance of these methods. In fact, just FCCI has led to assigning the rank of 1 to the faulty statement s_4. This confirms the importance of accurately handling CC test cases.

3. The proposed approach

Given a *PUT* with a test suite T, which is comprised of two sets of passed tests T_p and failed tests T_f, where T_p might be composed of a subset of coincidentally correct test cases T_{CC}; the goal of FCCI is to identify T_{CC} from T_p. The set of identified CC test cases, our estimate of T_{CC}, is called T_{iCC}. To this end, we considered different characteristics of test executions by proposing several CC identification factors and designed a fuzzy expert system that estimates the CC likelihoods of passed test cases with respect to the defined fuzzy rules, which effectively correlate the proposed factors. Fig. 4 shows the framework of FCCI, which has three main components, namely initial data extraction, CC identification factors extraction, and the fuzzy expert system. In the first step, FCCI runs *PUT* with T and collects the structural execution profile of test cases along with their test results, and calculates the suspiciousness of program statements. In the next step, it uses the collected information to compute the value of the proposed CC identification factors for each passed test case p in T_p. Finally, the value of the CC identification factors is fed into the designed fuzzy expert system to calculate the CC likelihood of passed test cases. The estimated CC likelihoods are then used to identify CC test cases. In the following, the proposed CC identification factors and also the structure of the proposed fuzzy expert system are described in further detail.

3.1. CC identification factors

Utilizing appropriate characteristics of test executions is the first step toward identifying CC test cases accurately. In fact, there are various factors that increase the likelihood of occurrence of coincidental correctness, and a CC identification method should consider them to give a proper CC likelihood estimation. In this section, we introduce these factors and motivate their usage for CC identification.

3.1.1. Suspiciousness score factor

The suspiciousness score of program statements executed by passed test cases is a primary factor to estimate their CC likelihoods. Statements with higher suspiciousness scores are more likely to be faulty and passed test cases that execute them are more likely to be coincidentally correct (Masri and Assi, 2010; Feyzi and Parsa, 2018c). Therefore, this factor should be involved

Table 4
The output of state-of-the-art CC identification methods for the code snippet depicted in Fig. 3.

Test Cases		t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}
Verified Test Results		F	CC	CC	CC	CC	CC	F	F	F	P	P
New Test Results Using	Masri Measure (Masri and Assi, 2010)	F	P	F	F	F	F	F	F	F	P	P
	k-means Clustering (k = 2) according to (Miao et al., 2012)	F	F	F	F	F	F	F	F	F	F	F
	k-means Clustering (k = 3) according to (Miao et al., 2012)	F	F	P	P	P	P	F	F	F	F	F
	k-means Clustering (k = 4) according to (Miao et al., 2012)	F	F	P	P	P	P	F	F	F	P	P
	k-means Clustering (k = 2) according to (Yang et al., 2015)	F	F	F	F	F	F	F	F	F	F	F
	k-means Clustering (k = 3) according to (Yang et al., 2015)	F	P	F	F	F	F	F	F	F	P	P
	k-means Clustering (k = 4) according to (Yang et al., 2015)	F	F	F	P	P	P	F	F	F	P	P
	Feizi Measure (Feyzi and Parsa, 2018)	F	P	F	F	F	F	F	F	F	P	P
	FCCI Measure	F	F	F	F	F	F	F	F	F	P	P

Table 5
Performance of Ochiai, Op^2, and DStar^3 SBFL techniques after identifying CC test cases using different CC identification methods.

S#	FCCI						CC_{Masri} / CC_{Feizi} / Clustering(k =3)[50]						Clustering(k=2)[19] / Clustering(k=2)[50]						Clustering(k=3)[19]					
	Ochiai		Op^2		DStar^3		Ochiai		Op^2		DStar^3		Ochiai		Op^2		DStar^3		Ochiai		Op^2		DStar^3	
	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R	Susp	R
1	0.90	4	5	4	40.50	4	0.85	6	2	6	21.33	5	1	1	11	1	+∞	1	0.79	1	-1	1	12.25	1
2	1.00	1	7	1	+∞	1	0.94	2	4	2	64.00	2	0.90	2	9	2	40.50	2	0.62	3	-3	6	4.16	2
3	0.94	3	6	3	64.00	3	1.00	1	5	1	+∞	1	0.85	4	8	4	21.33	4	0.53	5	-4	16	2.28	4
4	1.00	1	7	1	+∞	1	0.94	2	4	2	64.00	2	0.90	2	9	2	40.50	2	0.62	3	-3	6	4.16	2
5	0.47	9	0	9	0.57	9	0.25	19	-3	19	0.12	19	0.42	9	2	9	0.44	9	0.53	5	-2	3	0.80	9
6	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
7	0.33	12	-1	12	0.12	12	0.00	20	-4	20	0.00	20	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
8	0.88	5	5	4	24.50	5	0.93	4	4	2	49.00	4	0.79	5	7	5	12.25	5	0.42	11	-5	20	1.12	8
9	0.33	12	-1	12	0.12	10	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0.26	19	-4	16	0.14	19
10	0.81	6	4	6	12.00	6	0.86	5	3	5	18.00	6	0.73	6	6	6	7.20	6	0.46	10	-4	16	1.28	7
11	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0	20	-5	20	0.00	20
12	0.74	7	3	7	6.25	7	0.79	7	2	6	8.33	7	0.67	7	5	7	4.16	7	0.50	9	-3	6	1.50	6
13	0.47	9	0	9	0.57	9	0.50	9	-1	9	0.66	9	0.42	9	2	9	0.44	9	0	20	-6	23	0	20
14	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0	20	-5	20	0	20
15	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0	20	-5	16	0	20
16	0.57	8	1	8	1.50	8	0.61	8	0	8	1.80	8	0.52	8	3	8	1.12	8	0.65	2	-1	1	2.25	5
17	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
18	0.47	9	0	9	0.57	9	0.50	9	-1	9	0.66	9	0.42	9	2	9	0.44	9	0.53	5	-2	3	0.80	9
19	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
20	0.33	12	-1	12	0.12	12	0.35	11	-2	11	0.14	11	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
21	0.00	12	-4	23	0	20	0.00	20	-5	23	0	20	0.42	9	2	9	0.44	9	0.53	5	-2	3	0.80	9
22	0.00	12	-3	21	0	20	0.00	20	-4	20	0	20	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12
23	0.00	12	-3	22	0	20	0.00	20	-4	20	0	20	0.30	13	1	13	0.10	13	0.37	12	-3	6	0.16	12

in the identification process of CC test cases. In this regard, we suggest to calculate the suspiciousness score factor SS for each passed text case $p \in T_p$ according to Eq. (2).

$$SS(p) = \begin{cases} \frac{\sum_{s \in S_h} Suspiciousness_{Ochiai}(s)}{|S_h|} & if\ |S_h| > 0 \\ \frac{\sum_{s \in S_l} Suspiciousness_{Ochiai}(s)}{|S_l|} & if\ |S_h| = 0 \end{cases} \quad (2)$$

where S_l and S_h are the set of all program elements executed by p whose Ochiai suspiciousness scores are in the range of (0,0.5) and [0.5,1], respectively. In this way, more suspicious program statements play the main role in computing SS factor (the ones that their Ochiai suspiciousness score ranges from 0.5 to 1), and executing statements that are covered by none of the failed tests does not affect it (such statements are assigned suspiciousness score 0 by Ochiai). This facilitates the definitions of heuristic fuzzy rules related to the SS factor according to the following relationship between $SS(p)$ and CC likelihood of p: the higher the $SS(p)$, the higher the CC likelihood of p; A low value of $SS(p)$ also means that p is less likely to be coincidental correct.

We choose to use Ochiai in Eq. (2) since it is one of the best-studied SBFL techniques and has been reported to have promising results in locating faults (Wong et al., 2016; Abreu et al., 2009; Pearson et al., 2017; Le et al., 2013). Besides, the suspiciousness scores that it assigns to the program statements are bounded between 0 and 1 thereby there is no need for further normalization. We examined different ways of value handling and normalization for the SBFL techniques listed in Table 1 and obtained our best results with Ochiai.

3.1.2. Coverage ratio factor

The number of covered suspicious statements is another factor that should be considered to estimate the CC likelihoods. A passed test case that covers a large number of suspicious statements has a high chance to be coincidental correct because it is more

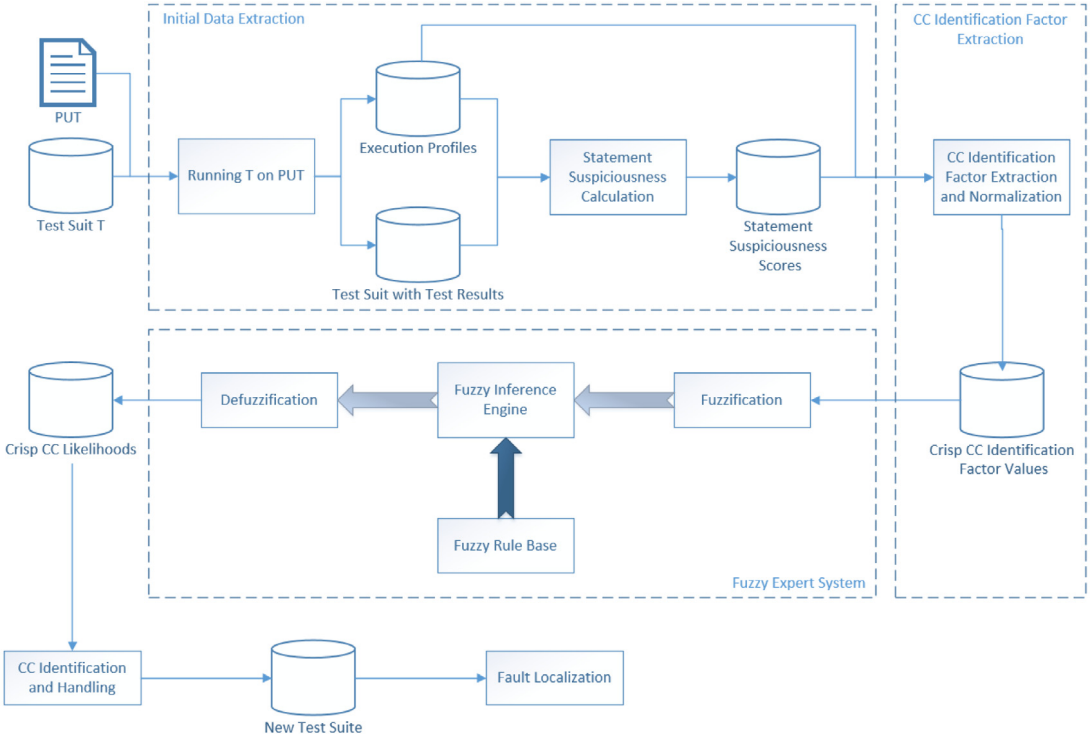

Fig. 4. The framework of FCCI.

likely to cover the faulty statement (Xue et al., 2014). For each passed text case $p \in T_p$, the coverage ratio factor CR is calculated according to Eq. (3).

$$CR(p) = \frac{|S_p|}{|S|} \tag{3}$$

where S_p is the set of all program statements executed by p and S is the set of all program statements. Both S and S_p include only those program statements whose Ochiai suspiciousness score ranges from 0.5 to 1. The higher the $CR(p)$, the higher the CC likelihood of p and vice versa. It should be noted that, without considering this factor, a CC test case that executes many suspicious statements but does not have a high value for the SS factor may be assigned a low CC likelihood.

3.1.3. Similarity factor

Some CC test cases may have a low value for the SS and/or CR factors relative to other ones, but have large similarity to a failed execution in terms of their execution profiles. Disregarding the similarity factor for such cases may lead to a low CC likelihood. For example, consider the CC test case t_2 in the motivating example depicted in Fig. 3, whose values of the SS and CR factors are 0.23 and 0.64, respectively. Since Masri and Assi (2010) and Feyzi and Parsa (2018c) methods do not consider the similarity of t_2 with the failed test cases, they assigned a low CC likelihood to t_2 and subsequently t_2 is not identified by these methods as a CC test case. On the other hand, clustering-based techniques with different configurations mostly identify t_2 as a CC test case because of its large similarity to the failed execution t_1. This shows that considering the similarity of passed executions to the failed ones can increase the chance of identifying such CC test cases. In this regard, we suggest calculating the similarity factor SF for each passed test case $p \in T_p$ using Eq. (4).

$$SF(p) = \frac{1}{\min_{1 \le i \le N_F} \left(D\left(e_p, e_{f_i}\right)\right)} \tag{4}$$

where N_F is the number of failed tests, e_p and e_{f_i} are respectively the execution profiles of$p$ and the ith failed test, and $D\left(e_p, e_{f_i}\right)$ is the Euclidean distance (Deza and Deza, 2009) between e_p and e_{f_i}. The higher the $SP(p)$, the higher the CC likelihood of p. Conversely, the lower the $SP(p)$, the lower the CC likelihood of p.

To obtain the similarity between executions we use a weighted execution profile in which the weight of each statement is its Ochiai suspiciousness score. More formally, given a test suite T that exercises statements $s_1, s_2, \ldots, s_n$, each test case t_i in T is represented by the following feature vector:

$$e_{t_i} = \langle c_i(s_1) * Suspiciousness_{Ochiai}(s_1), c_i(s_2) * Suspiciousness_{Ochiai}(s_2), \ldots, c_i(s_n) * Suspiciousness_{Ochiai}(s_n) \rangle \tag{5}$$

where c_i is a characteristic function defined as follows:

$$c_i(s) = \begin{cases} 1 \text{ if statement } s \text{ is exercised by } t_i, \\ 0 \text{ otherwise}. \end{cases} \tag{6}$$

Given two execution profiles $e_{t_i} =< e^1_{t_i}, e^2_{t_i}, \ldots, e^n_{t_i} >$ and $e_{t_j} =< e^1_{t_j}, e^2_{t_j}, \ldots, e^n_{t_j} >$, the Euclidean distance between e_{t_i} and e_{t_j} is calculated according to Eq. (7).

$$D\left(e_{t_i}, e_{t_j}\right) = \sqrt{\sum_{k=1}^{n} (e^k_{t_i} - e^k_{t_j})^2} \tag{7}$$

It is worth noting that we also examined other distance formulas such as Cosine (Singhal, 2001) and Correlation (Pearson, 1895) to calculate the distance between test cases, and observed no significant impact on the final results.

3.1.4. Fault-masking factor

As stated earlier, coincidental correctness occurs when faults fail to propagate, i.e., the faulty statement is executed, however, the same statement or the subsequent ones mask the fault and

Table 6
Three code snippets and their corresponding DRR values.

ID	Code construct	DRR
C_1	$y = x * 3;$	$\frac{5}{5} = 1$
C_2	$y = x\%3;$	$\frac{5}{3} = 1.67$
C_3	$if\ (x >= 3)\ \{y = 1;\}\ else\ \{y = 0;\}$	$\frac{5}{2} = 2.5$

prevent the occurrence of failure. In fact, operators in the computational and conditional expressions can have a direct impact on masking the faults and subsequently on the prevention of the occurrence of failures. As an example, in the procedure *TestMe* depicted in Fig. 3, the value of the variables a and b may be in an interval where the evaluation result of the conditional expression of the faulty statement s_4 does not lead to the execution of the wrong branch (as in the case of test cases t_2 to t_6). Also in case of the faulty statement s_6, regardless of the values of variable a, the output will be correct if the value of b is zero. As can be seen, there is a very higher chance for the fault to be masked in case of s_4 than that of s_6. In case of FEP, the program is transitioned into an infection state but the infection does not propagate to the output. This can be interpreted as information loss along the path starting at the infection location and ending at the output (Androutsopoulos et al., 2014; Voas and Miller, 1993; Clark and Hierons, 2012). In this regard, Voas and Miller (1993) devised the Domain-to-Range Ratio (DRR) metric to approximate the information loss between input and output. DRR is the ratio of the cardinality of the possible inputs to the cardinality of the possible output of a mathematical function or program statement(s). A larger DRR represents higher information loss and in our context covering code constructs with higher DRR increases the likelihood of coincidental correctness. The usage of DRR was represented by Masri and Assi (2014) using three code constructs namely C_1, C_2, C_3 depicted in Table 6, in which the last column represents their corresponding DRR values.

In these examples, let us assume that the variable x takes on the values (Vessey, 1985; Xie et al., 2013), of which the value 4 represents an infection. Considering C_1, there is a one-to-one mapping between x values and y values. That is, for each value of x a unique value for y is obtained, and this leads to the successful propagation of the infection. Put it differently, the infection $x = 4$ causes the infection $y = 12$. In contrast to C_1, C_2 and C_3 will cause information loss, but with varying degrees. In C_2, when x is infected (i.e., x is 4), y takes on the value 1. However, for $x = 1$ the value of y would also become 1. Therefore, y being 1 does not necessarily represent a propagated infection. In C_3, y becomes 1 if x is 3, 4, or 5. Thus, C_3 might cause coincidental correctness with a higher probability. In this way, code constructs similar to C_2 and especially C_3 might prevent the infection from propagation, and given that they are pervasive, it is necessary to consider them for the identification of CC test cases.

In this regard, we consider specifying an impact factor for the most-common operators based upon their impact on fault-masking and propose to compute the value of the fault-masking factor FM for each passed test case $p \in T_p$ according to Eq. (8).

$$FM\ (p) = \sum_{i=1}^{l} IF(S'_i) \tag{8}$$

Where S' is a sequence of program statements that are executed by p, affect the output, and located after the first suspicious statement with the suspiciousness score greater than 0.5, l is the length of the sequence, and the *IF* function returns the impact factor of the corresponding program statement in fault-masking. This factor can be used in accompany with SS and CR factors in the CC likelihood estimation of passed test cases (see Section 3.2.1).

Table 7
The specified impact factor of some common operators in masking faults.

Operators	Impact factor
$>$	0.79
$<$	0.79
$=$	0.79
$\geq$	0.79
$\leq$	0.79
$+$	0.08
$-$	0.08
$*$	0.08
$/$	0.08
mod	0.38
div	0.38

To quantify impact factors, we perform an empirical approach utilizing an improved Simulated Annealing (SA) algorithm, which has been recently proposed by Morales-Castañeda et al. (2019). SA (Kirkpatrick et al., 1983) is one of the most well-known metaheuristic methods which maintains remarkable characteristics such as robustness and efficiency. It also operates with low memory requirements and is easy to implement. These characteristics have motivated the use of SA in several engineering problems (Suman and Kumar, 2006; Amine, 2019). Morales-Castañeda et al. (2019) modified the original SA incorporating two new operators, namely folding and reheating, inspired by the ancient Japanese Swordsmithing technique to improve its search abilities. As stated earlier, we expect a very low impact factor for arithmetic operators such as $\times$ and /, medium impact factor for operators such as *mod*, and high impact factor for conditional operators. Therefore, in this procedure, we define the range of [0, 0.3), [0.3, 0.6) and [0.6, 1] for them, as the range of feature values, respectively. Here, the cost function is to maximize the average F-measure of the utilized benchmarks (see Section 4.2.2). Finally, we obtained the impact factors depicted in Table 7.

It should be noted that any computation that reduces the entropy of inputs will have the potential to lose error information, thereby leading to coincidental correctness. We leave examining the impact of other computations on identifying CC test cases for future work.

3.1.5. Static fault-proneness factor

Studies have shown that some parts of programs are more likely to contain faults (Fitzsimmons and Love, 1978; Menzies et al., 2006). This fact should be considered in the identification of CC test cases. In fact, execution of the high suspicious statements located in the fault-prone areas of the program by passed test cases should be reflected in their CC likelihood estimation. Zhang (2008) found that the distribution of software faults can be modeled by a Weibull probability distribution. Generally speaking, fault proneness of a part of a program is correlated with its complexity, which can be captured by different static code complexity metrics (Henry and Kafura, 1981; Pai and Dugan, 2007; Lewis and Henry, 1989). In this regard, several software fault prediction approaches have been proposed to predict fault-prone modules using software metrics (Catal, 2011; Dejaeger et al., 2012; Rathore and Kumar, 2019). However, there has been considerable debate about the extent to which software fault prediction models constructed from the static code features actually help software testing processes (Shepperd and Ince, 1994; Fenton and Pfleeger, 1997). More recently, the validity of such fault predictors has been empirically illustrated. For example, Menzies et al. (2006) argued that fault predictors based on static code features are useful, generalizable, easy to use, and widely-used. This has been

confirmed later in different studies such as Catal and Diri (2009), Menzies et al. (2010), Turhan et al. (2009), He et al. (2015). This motivates us to incorporate the fault-proneness analysis based upon static code metrics into the CC identification problem. This is done by constructing a fault prediction model, applying it to predict fault-prone areas of the subject programs, obtaining a static fault-proneness likelihood for program statements, and consequently calculating the static fault-proneness factor (SFP) for each passed execution.

The static code features that are used to construct our fault prediction model are depicted in Fig. 5, which includes Halstead (Halstead, 1977), McCabe (McCabe, 1976), and lines of code (locs) metrics. Several datasets such as NASA Metrics Data Program[2] and PROMISE Data Repository[3] include these metrics. To construct the fault prediction model we utilize logistic regression, which has been shown promising results (Hall et al., 2011; Bowes et al., 2018) and has the general form

$$P(Y) = \frac{1}{1 + e^{-(c+\beta_1 X_1+\beta_2 X_2+\cdots++\beta_n X_n)}} \tag{9}$$

where β_i are the logistic regression predicted constants and the X_i are the independent variables used for building the logistic regression model. In our case, these variables are principal components obtained after applying principal component analysis (PCA). After obtaining principal components, a ridge regression method is employed, which is capable of reducing multicollinearity in the construction of the prediction model and producing a model with high prediction accuracy (Parsa et al., 2008).

The static complexity code metrics are calculated by imagix4D,[4] and the fault prediction model is constructed using Weka machine learning and data mining tool.[5] Finally, the value of the static fault-proneness factor SFP for each passed test case p is calculated according to Eq. (10).

$$SFP(p) = \sum_{s \in S_p} \frac{SFPL(s)}{|S_p|} \tag{10}$$

where the $SFPL(s)$ function returns the static fault-proneness likelihood of statement s. In this way, $SFP(p)$ quantifies the static fault-proneness of high suspicious program statements of a passed execution p. The higher the $SFP(p)$, the higher the CC likelihood of p. However, it should be noted that the lower values of $SFP(p)$ do not necessarily imply that p is less likely to be coincidental correct. Because it is not necessary for all of the program faults to be located around the complex and fault-prone areas, and we should not let low static fault-proneness of p causes a reduction in its final CC likelihood estimation.

Finally, It is worth mentioning that the information gained by the static analysis of the PUT has been recently used to improve the performance of SBFL techniques (Neelofar et al., 2017; Feyzi and Parsa, 2018a). For example, Neelofar et al. (2017) use simple static analysis to divide statements into different categories including *assignment*, *conditional*, and *other*. For instance, they group if-then-else blocks, for loops, and while loops in the conditional category. Then each category is given a weight, which is added to the suspiciousness score calculated using SBFL techniques to compute final scores of the statements. Although these techniques try to assist fault localization, however, they are vulnerable to the coincidental correctness because they use suspiciousness scores calculated using pure SBFL techniques. Therefore, since FCCI accurately identify CC test cases, it is orthogonal to them and can increase their effectiveness by boosting the performance of SBFL formulas.

[2] http://www.menzies.us/data.html
[3] http://promise.site.uottawa.ca/SERepository/
[4] http://www.imagix.com/products/source-code-analysis.html
[5] http://www.cs.waikato.ac.nz/ml/weka/

3.2. Overview of the proposed fuzzy expert system

In the previous section, we proposed several CC identification factors, which have a direct impact on increasing the likelihood of identifying coincidental correctness. The next step is to find a way to properly correlate these factors to estimate the CC likelihood of passed test cases. To this end, it should be considered that vagueness or uncertainty exists when we infer the CC likelihoods from the characteristics of passed executions. For example, we know that a passed test with high similarity to a failed one tends to be associated with higher CC likelihood. However, it is difficult to provide a clear cut definition of what "high similarity" is. Moreover, other characteristics may confer low CC likelihood on a passed execution despite the high similarity to a failed test. Thus, it is difficult to deal with CC likelihood estimation using precise mathematical equations; instead, such vagueness and uncertainty can be addressed in fuzzy set theory (Zedeh, 1965).

In this regard, we model the CC test case identification as a decision making system and employ fuzzy expert system (specifically, a Mamdani-type fuzzy inference system) as the basis of FCCI. The schematic of the proposed fuzzy expert system is depicted in Fig. 6.

Fuzzy expert systems make use of fuzzy reasoning (Zadeh, 1983) which is based on the concepts of fuzzy sets theory and fuzzy rules (Pappis and Siettos, 2014). Fuzzy set theory provides a framework for handling uncertainties which was first initiated by Zedeh (1965). Unlike Boolean/Crisp sets in which an element is either a member of the set or not, in fuzzy sets, each element is given the degree of membership. Formally, if X is a collection of objects denoted generically by x, then a fuzzy set A in X is a set of ordered pairs:

$$A = \{(x, \mu_A(x) \,|x \in X)\} \tag{11}$$

where $\mu_A(x)$ is called membership/characteristic function which maps x to the interval [0,1]. 1 means full membership, 0 means no membership and any value in between, e.g., 0.5, is called graded membership. The nearer the value of $\mu_A(x)$ to unity, the higher the grade of membership of x in A (Zedeh, 1965). In fact, the fuzzy sets are characterized by membership functions (MFs).

A fuzzy expert system consists of fuzzification, knowledge base, inference, and defuzzification subsystems (Schneider et al., 1996; Wang, 1999; Ross, 2009). In the following, we will introduce them and explain our methodologies used in these parts.

3.2.1. Knowledge base

Knowledge base contains essential information about a problem domain which mostly represented and encoded as facts and rules (Abraham, 2005). In this study, we use a rule based knowledge base to represent the relations between the different values of the CC identification factors and CC likelihood of passed test cases. A rule base consists of a set of fuzzy IF-THEN rules and is the heart of the fuzzy system in the sense that all other components are used to implement these rules in a reasonable and efficient manner (Wang, 1999). In a Mamdani-type fuzzy system, premises and the consequences of the IF-THEN rules are linguistic variables associated with fuzzy concepts (Abraham, 2005). The canonical form of fuzzy IF-THEN rules in fuzzy rule base is as follows:

$$Ru^{(l)} : IF\ x_1\ is\ A_1^l\ and\ \ldots\ and\ x_n\ is\ A_n^l\ THEN\ y\ is\ B^l \tag{12}$$

where A_i^l and B^l are input and output linguistic terms represented by fuzzy sets in $U_i \subset R$ and $V \subset R$, respectively. U and V denote the universes of discourse in the input and output domains, R is real numbers, $x = (x_1, x_2, \ldots, x_n)^T \in U$ and $y \in V$ are the inputs and output variables of fuzzy system respectively. Also, $l = 1, 2, \ldots, M$ and M is the number of the rules in the fuzzy

m = Mccabe		$v(g)$	cyclomatic_complexity
		$iv(G)$	design_complexity
		$ev(G)$	essential_complexity
locs	loc		loc_total (one line = one count)
	loc(other)		loc_blank
			loc_code_and_comment
			loc_comments
			loc_executable
			number_of_lines
			(opening to closing brackets)
Halstead	h	N_1	num_operators
		N_2	num_operands
		μ_1	num_unique_operators
		μ_2	num_unique_operands
	H	N	length: $N = N_1 + N_2$
		V	volume: $V = N * \log_2 \mu$
		L	level: $L = V^*/V$ where
			$V^* = (2 + \mu_2^*) \log_2 (2 + \mu_2^*)$
		D	difficulty: $D = 1/L$
		I	content: $I = \hat{L} * V$ where
			$\hat{L} = \frac{2}{\mu_1} * \frac{\mu_2}{N_2}$
		E	effort: $E = V/\hat{L}$
		B	error_est
		T	prog_time: $T = E/18$ seconds

Fig. 5. Static code features used to construct the fault prediction model (Menzies et al., 2010).

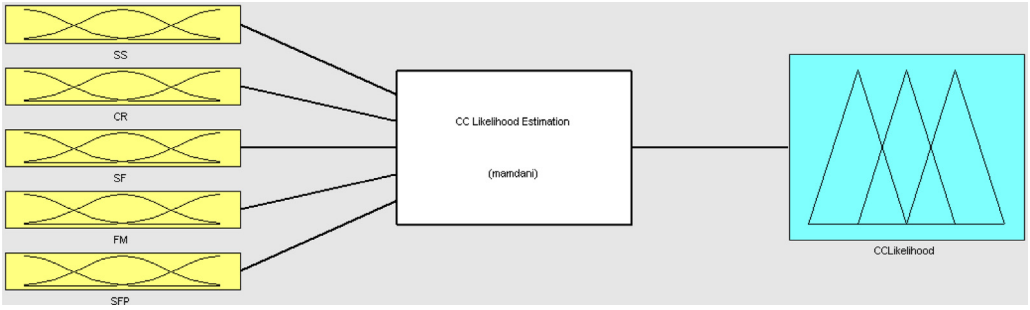

Fig. 6. The schematic of the proposed fuzzy expert system.

rule base. The antecedent describes to what degree the rule is applied, while the consequent assigns a membership function to the output variable (Abraham, 2005).

In general, the rules and membership functions could be obtained from the experts. In this study, the CC identification problem is formulated as a set of fuzzy rules (Table 8) according to the proposed CC identification factors and their relationships with the coincidental correctness likelihood of passed executions presented in Section 3.1.

In this regard, the value of the four inputs of the system including SS, CR, SF, and SFP, and also the output (i.e., the CC likelihood), are modeled using five linguistic variables (Very Low, Low, Medium, High and Very High) by Gaussian membership function $\mu_A(x) = e^{-\frac{1}{2}\left(\frac{X-C}{\sigma}\right)^2}$, where C and σ are the center and spread of the function, respectively. Since all inputs of the fuzzy expert system (i.e., CC identification factors) are normalized in the range of 0 to 1, we employ symmetrical shape and equal spread membership functions. Accordingly, we choose $\sigma = 0.1$ and $C = 0, 0.25, 0.50, 0.75, 1$ for VL (Very Low), L (Low), M (Medium), H (High), VH (Very High) membership functions, respectively. These membership functions are shown in Fig. 7 and can be mathematically represented as:

$$\begin{cases} \mu_{VH}(x) = e^{\frac{-(X-1)^2}{0.02}} \\ \mu_{H}(x) = e^{\frac{-(X-0.75)^2}{0.02}} \\ \mu_{M}(x) = e^{\frac{-(X-0.5)^2}{0.02}} \\ \mu_{L}(x) = e^{\frac{-(X-0.25)^2}{0.02}} \\ \mu_{VL}(x) = e^{\frac{-X^2}{0.02}} \end{cases} \quad (13)$$

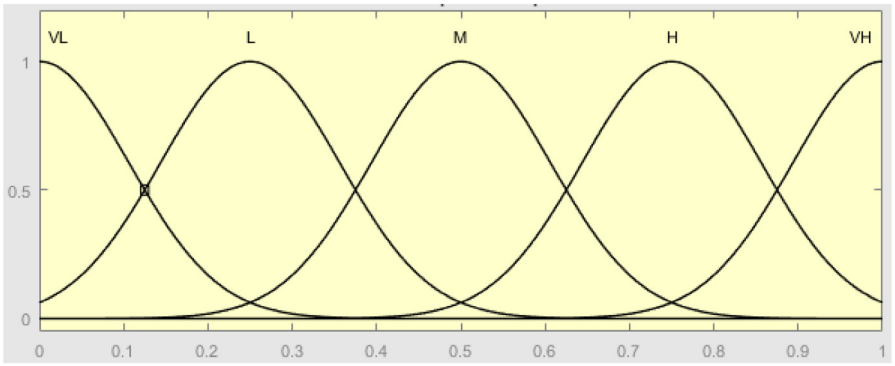

Fig. 7. Membership functions for the inputs SS, CR, SF, and SFP, and the output CC likelihood.

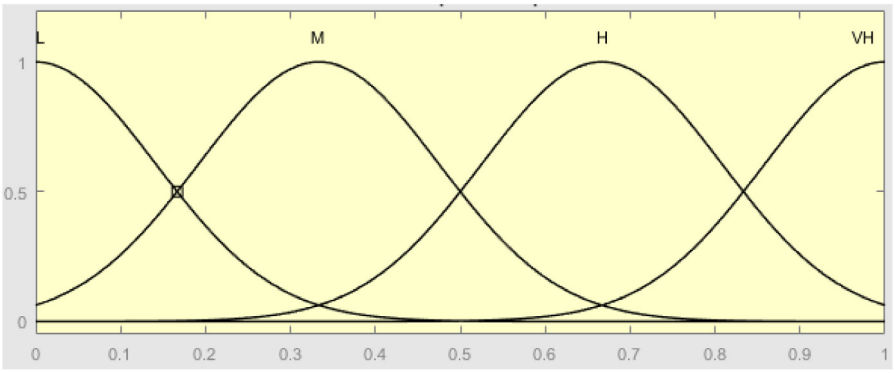

Fig. 8. Membership functions for the inputs FM.

Four membership functions were also developed for FM input (Fig. 8) representing Low (L), Medium (M), High (H), and Very High (VH) linguistic classes, which are defined with parameters $\sigma = 0.1$ and $C = 0, 0.3333, 0.6667, 1$ respectively. These membership functions can be mathematically represented as:

$$\begin{cases} \mu_{VH}(x) = e^{\frac{-(X-1)^2}{0.02}} \\ \mu_{H}(x) = e^{\frac{-(X-0.6667)^2}{0.02}} \\ \mu_{M}(x) = e^{\frac{-(X-0.3333)^2}{0.02}} \\ \mu_{L}(x) = e^{\frac{-(X)^2}{0.02}} \end{cases} \quad (14)$$

Based on the aforementioned relationships between the CC identification factors and coincidental correctness likelihood of passed test cases, 41 rules (Table 8) were developed to map the input membership functions to the output membership functions. For example, as stated earlier, the higher the *SP* (p), the higher the CC likelihood of p. Conversely, the lower the *SP* (p), the lower the CC likelihood of p. Therefore, we define 5 rules corresponding to the similarity factor to formulate this relationship (i.e., Rules 35 to 40). We follow the same idea for defining the respective rules for SS and CR factors, with the exception that a higher value of an FM factor in an antecedent would upgrade the influence of the respective SS and CR factors to a higher level on the final estimation of CC likelihood. For example, among two passed executions with medium SS factor, the one with high or very high FM factor are more likely to be coincidental correct. However, note that the lower values of the FM factor do not necessarily imply the lower CC likelihood. We formulate such a relationship through rules number 9 to 12. Also since the higher static fault-proneness of passed executions increases their likelihood of being coincidental correct and the reverse relationship does not hold, we define only two rules (i.e., rules number 40 and 41) corresponding to the static fault-proneness factor, which result in an increase in the CC likelihood of passed test cases with *high* or *very high* SFP factor. In this way, the low static fault-proneness of p does not lower its final CC likelihood. Notice that the weight of all fuzzy rules in the inference system is considered 1.

It should be noted that we have taken a fine-tuning step to design the proposed fuzzy expert system. Given the determinative role of membership functions in the performance of fuzzy systems, to choose the best-suited MFs for FCCI, we assessed four prevalent types of membership functions including triangular-shaped, trapezoidal-shaped, bell-shaped and Gaussian curves with different parameter values[6] similar to the procedure used in Sambariya and Prasad (2017), Zhao and Bose (2002), Kim et al. (2017). In this procedure, the performance of FCCI with different membership functions and with varying number of linguistic variables is compared on a subset of benchmarks (see Section 4.3) to get the best-suited MF. Since all inputs of the fuzzy expert system (i.e., CC identification factors) are normalized in the range of 0 to 1, we employ symmetrical shape and equal spread membership functions. In addition, the same type of MF is used for input and output variables. Furthermore, to calibrate our rule set with varying/increasing number of linguistic variables, we used the same rationale as that used to generate the set of rules depicted in Table 8. Finally, the best-performing FCCI is found using the above-mentioned settings. In our investigations, we observed that with increased linguistic variables as 6 or above, the improvement is very negligible. Moreover, modeling the FM factor using 4 linguistic variables reduced the rule set without negatively affecting the performance of FCCI.

[6] For example, we tried different widths for Gaussian membership functions or employ "narrow shoulder" and "wide shoulder" types for trapezoidal-shaped and bell-shaped membership functions.

Table 8
Heuristic fuzzy rules defined in the fuzzy system to assign CC likelihood of passed test cases.

Rule No	If (Antecedent)	Then (Consequent)	Weight
Rule 1	SS is *very low and* FM is *low*	CC likelihood is *very low*	1
Rule 2	SS is *very low and* FM is *medium*	CC likelihood is *low*	1
Rule 3	SS is *very low and* FM is *high*	CC likelihood is *low*	1
Rule 4	SS is *very low and* FM is *very high*	CC likelihood is *low*	1
Rule 5	SS is *low and* FM is *low*	CC likelihood is *low*	1
Rule 6	SS is *low and* FM is *medium*	CC likelihood is *low*	1
Rule 7	SS is *low and* FM is *high*	CC likelihood is *medium*	1
Rule 8	SS is *low and* FM is *very high*	CC likelihood is *medium*	1
Rule 9	SS is *medium and* FM is *low*	CC likelihood is *medium*	1
Rule 10	SS is *medium and* FM is *medium*	CC likelihood is *medium*	1
Rule 11	SS is *medium and* FM is *high*	CC likelihood is *high*	1
Rule 12	SS is *medium and* FM is *very high*	CC likelihood is *very high*	1
Rule 13	SS is *high and* FM is *low*	CC likelihood is *high*	1
Rule 14	SS is *high and* FM is *medium*	CC likelihood is *high*	1
Rule 15	SS is *high and* FM is *high*	CC likelihood is *very high*	1
Rule 16	SS is *high and* FM is *very high*	CC likelihood is *very high*	1
Rule 17	SS is very *high*	CC likelihood is very *high*	1
Rule 18	CR is *very low and* FM is *low*	CC likelihood is *very low*	1
Rule 19	CR is *very low and* FM is *medium*	CC likelihood is *low*	1
Rule 20	CR is *very low and* FM is *high*	CC likelihood is *low*	1
Rule 21	CR is *very low and* FM is *very high*	CC likelihood is *low*	1
Rule 22	CR is *low and* FM is *low*	CC likelihood is *low*	1
Rule 23	CR is *low and* FM is *medium*	CC likelihood is *low*	1
Rule 24	CR is *low and* FM is *high*	CC likelihood is *medium*	1
Rule 25	CR is *low and* FM is *very high*	CC likelihood is *medium*	1
Rule 26	CR is *medium and* FM is *low*	CC likelihood is *medium*	1
Rule 27	CR is *medium and* FM is *medium*	CC likelihood is *medium*	1
Rule 28	CR is *medium and* FM is *high*	CC likelihood is *high*	1
Rule 29	CR is *medium and* FM is *very high*	CC likelihood is very *high*	1
Rule 30	CR is *high and* FM is *low*	CC likelihood is *high*	1
Rule 31	CR is *high and* FM is *medium*	CC likelihood is *high*	1
Rule 32	CR is *high and* FM is *high*	CC likelihood is *very high*	1
Rule 33	CR is *high and* FM is *very high*	CC likelihood is *very high*	1
Rule 34	CR is *very high*	CC likelihood is *very high*	1
Rule 35	SF is *very low*	CC likelihood is *very low*	1
Rule 36	SF is *low*	CC likelihood is *low*	1
Rule 37	SF is *medium*	CC likelihood is *medium*	1
Rule 38	SF is *high*	CC likelihood is *high*	1
Rule 39	SF is *very high*	CC likelihood is *very high*	1
Rule 40	SFP is *high*	CC likelihood is *high*	1
Rule 41	SFP is *very high*	CC likelihood is *very high*	1

3.2.2. Fuzzification

Since the inputs of the fuzzy system are crisp real-valued numbers, we must construct interfaces between the fuzzy inference engine and the environment. Fuzzification is the process of converting input data into linguistic variables and the fuzzifier is defined as a mapping from a real-valued point $x^* \in U \subset R^n$ to a fuzzy set A' in U, where U denotes the universes of discourse in the input domain and R is real numbers. In the proposed system, the fuzzification of the inputs is performed according to the fuzzy membership functions shown in Figs. 7 and 8 using singleton fuzzifier.

3.2.3. Inference engine

Inference engine uses mechanisms to extract new knowledge based on the rules stored in knowledge base and the information provided by the user. A Mamdani style fuzzy inference engine evaluates the mentioned fuzzy rules in which implication and AND method for evaluation of rules are both set to algebraic product operator $(\mu_A(x).\mu_B(x))$. Also, aggregation process for combining fuzzy sets, which represents the output of each rule into a single fuzzy set, is accomplished by algebraic sum operator $(\mu_A(x) + \mu_B(x) - \mu_A(x).\mu_B(x))$.

3.2.4. Defuzzification

Since Mamdani systems yield fuzzy output and FCCI requires the crisp value for CC likelihood of passed tests, after performing the reasoning by inference engine, the aggregated output membership function $\mu_{B'}(x)$ should be transformed into a single value y^*, representing the result of inference which is actually the CC likelihood of the corresponding passed test case. Thus the defuzzifier is defined as a mapping from fuzzy set B' in $V \subset R$ to a crisp point $y^* \in V$, where V denotes the universes of discourse in the output domain. We used center of gravity defuzzifier which is the most widely adopted defuzzification strategy (Wang and

Algotithm 1: FCCI

Inputs: Program under test PUT, Test suite T
Output: The set of identified CC test cases T_{iCC}, *New test suite* T'

$T_{iCC} \leftarrow \{\}$
$T_p \leftarrow \{\}$
$T_f \leftarrow \{\}$
for each test case t **in** T
  $(O'_t, e_t) \leftarrow ConcreteRun(PUT, I_t)$
  $CovMat[t] \leftarrow e_t$
  if $O'_t = O_t$
    $T_p \leftarrow T_p \cup t$
    $ResVec[t] \leftarrow 0$
  else
    $T_f \leftarrow T_f \cup t$
    $ResVec[t] \leftarrow 1$
  end if
end for
for each statement s **in** PUT
  $Susps[s] \leftarrow CalculateSuspicoiusnessScores('Ochiai', CovMat, ResVec, s)$
  $SFPL[s] \leftarrow FaultPredictionModel(PUT, s)$
end for
for each p **in** T_p
  $FeatureVector_p \leftarrow \langle SS(p), CR(p), SF(p), FM(p), SFP(p)\rangle$ //*Utilizing Susps, SFPL, CovMat and ResVec*
  $CClikelihoods[p] \leftarrow FuzzyExpertSystem(FeatureVector_p)$
end for
$T_{iCC} \leftarrow$ CCIdentification($CClikelihoods$)
$T_p \leftarrow T_p \setminus T_{iCC}$
$T_f \leftarrow T_f \cup T_{iCC}$
$T' \leftarrow T_f \cup T_p$
return T_{iCC}, T'

Fig. 9. The proposed FCCI algorithm.

Kusiak, 2000) and also is the most prevalent and intuitively appealing among the defuzzification methods (Ross, 2009; Lee, 1990). This method calculates the y^* as the center of the area covered by the membership function B' and represented as:

$$y^* = \frac{\int_V y\mu_{B'}(y)dy}{\int_V \mu_{B'(y)dy}} \tag{15}$$

3.3. CC identification

After estimating CC likelihoods by the fuzzy expert system, we select passed test cases whose CC likelihoods ranges from 0.5 to 1 as coincidentally correct. We obtained the value of this threshold (i.e., δ) empirically by examining different ranges (see Section 4.4.4). Fig. 9 depicts the proposed FCCI algorithm.

4. Experimental evaluation

FCCI aims to provide a useful measure to identify coincidental correct test cases. Properly identifying such test cases and removing their effects from the test suite, improves the effectiveness of SBFL techniques and accordingly facilitates the debugging process. To evaluate the performance of FCCI, we designed a set of experiments addressing the following research questions:
RQ1. How well FCCI can improve the performance of SBFL techniques to locate faults?
RQ2. Does FCCI perform better in the identification of CC test cases compared to the state-of-the-art approaches?
RQ3. How does FCCI perform on multiple-fault programs? Since almost all real-world programs contain more than one fault, it is important to investigate the performance of CC identification methods on program versions containing multiple faults (Wong et al., 2016).
RQ4. What is the impact of different configurations on the performance of FCCI?

RQ1, RQ2, and RQ3 aim to provide a comprehensive comparison between FCCI and other popular methods from different perspectives, and RQ4 focuses on examining different values for the δ threshold and also impact factors obtained for computing fault-masking factor. RQ4 also investigates the impact of the number of failing test cases and also each CC identification factor on the performance of FCCI.

4.1. Subject programs

We have used 17 open source and popular programs including seven programs from *Siemens* suite (Hutchins et al., 1994), four programs from Unix utilities, and also six programs from *Defects4J* suite (Just et al., 2014). Table 9 provides an overview of these programs and their corresponding test suites. All of the *Siemens* suite and Unix utility programs are downloaded from Software Infrastructure Repository (SIR).[7] These programs have been widely used to evaluate fault localization techniques (e.g., Abreu et al. (2009), Wong et al. (2014), Naish et al. (2011), Jones and Harrold (2005), Feyzi and Parsa (2019), Parsa et al. (2014), Wong et al. (2012b)) as well as CC identification approaches (e.g., Masri and Assi (2014, 2010), Miao et al. (2012),

[7] https://sir.csc.ncsu.edu

Table 9
Subject programs used in the experiments.

Type	Program	Description	Faulty versions	LOC	#Tests	#Passed Tests		#CC Tests		Language	Fault type
						Max	Avg	Max	Avg		
Siemens suite	printtokens	Lexical analyzer	7	472	4056	3974	3938	1221	562	C	Seeded
	printtokens2	Lexical analyzer	10	399	4071	4007	3723	3527	1886	C	Seeded
	schedule	Priority scheduler	9	292	2650	2617	2481	1474	1327	C	Seeded
	schedule2	Priority scheduler	10	301	2680	2674	2662	2621	2157	C	Seeded
	tcas	An aircraft collision avoidance system	41	141	1578	1576	1498	1531	983	C	Seeded
	totinfo	Computes statistics given input data	23	440	1054	1051	989	1045	790	C	Seeded
	replace	Performs pattern matching and substitution	32	512	5542	5538	5447	4940	1411	C	Seeded
Unix utilities	gzip	File compression /decompression tool	23	6K	217	216	197	45	28	C	Seeded
	grep	Pattern matching engine based on regular expressions	17	12K	809	806	671	779	384	C	Seeded
	sed	Stream editor	17	12K	370	279	265	115	70	C	Real & Seeded
	space	Interpreter for an array definition language	38	9K	13585	13567	11059	12279	3486	C	Real
Defects4J	JFreeChart	A Java chart library	26	96K	2205	2186	1790	43	19	Java	Real
	Joda-Time	Provides a quality replacement for the Java date and time classes.	27	28K	4130	4084	3817	242	92	Java	Real
	Apache commons-lang	Provides a host of helper utilities for the java.lang API	65	22K	2245	2240	1745	14	9	Java	Real
	Apache commons-math	A library of lightweight, self-contained mathematics and statistics components for Java programming	106	85K	3602	3599	2476	57	15	Java	Real
	Mockito framework	A mocking framework for unit tests written in Java	38	23K	1457	1450	1104	218	90	Java	Real
	Closure compiler	A JavaScript checker and optimizer	133	90k	7927	7876	6602	4124	608	Java	Real

Feyzi and Parsa (2018c,b), Miao et al. (2013), Li and Liu (2012, 2014), Yang et al. (2015)). However, most of the faults in these subjects are hand-seeded or obtained from mutations. Just *space* and some versions of *sed* contain real faults. Since it is unclear whether artificial bugs capture true characteristics of real bugs in real programs (Pearson et al., 2017; Chekam et al., 2016), we employ *Defects4j* suite[8] to better evaluate FCCI on real faults. Defects4j is one of the largest available datasets of well-organized real-world java bugs (Just et al., 2014; Martinez et al., 2017), which has been widely used in prior fault localization work (Pearson et al., 2017; Feyzi and Parsa, 2018c,a; Zou et al., 2019). The dataset contains 6 software projects, namely *JFreeChart*,[9] *Apache commons-lang*,[10] *Apache commons-math*,[11] *Joda-Time*[12], *Mockito framework*[13], and *Google Closure complier*[14].

These programs vary dramatically in terms of size, functionality, number of faulty versions, types of faults and also number of test cases. This allows us to better generalize the finding and results of this paper. It should be noted that some faulty versions are eliminated due to the segmentation faults, compile errors, observing no failed test cases or locating the faults in the header files (e.g., versions 4 and 6 of *printtokens*, or version 9 of *schedule2*).

8 https://github.com/rjust/defects4j
9 http://jfree.org/jfreechart
10 http://commons.apache.org/proper/commons-lang
11 http://commons.apache.org/proper/commons-math
12 https://www.joda.org/joda-time/
13 https://site.mockito.org/
14 https://developers.google.com/closure/compiler

4.2. Evaluation metrics

We used a set of evaluation metrics in our experiments to compare the performance of FCCI on improving the effectiveness of SBFL techniques and on identifying CC test cases. In the following, we describe these metrics in further detail.

4.2.1. Evaluation metrics for fault localization

To evaluate the performance of CC identification methods in terms of improving the effectiveness of SBFL techniques, we use the following four major measurement metrics:

- *EXAM Score* (Pearson et al., 2017; Wong et al., 2014, 2010, 2012a; Renieres and Reiss, 2003): EXAM score is one of the most popular metrics to evaluate SBFL techniques (Wong et al., 2016; Pearson et al., 2017) and indicates the percentage of statements that have to be examined until the first faulty statement is reached. Formally, it is measured by the expression:

$$\text{EXAM score} = \frac{\#\ of\ statements\ examined}{\#\ of\ all\ program\ statements} \times 100\% \qquad (16)$$

- *Average Number of Statements Examined (ANSE)*: ANSE indicates the average number of statements that need to be examined with respect to all faulty versions (of a subject program) to find all faults. More formally, if a program

has z faulty versions and E and F are two different SBFL techniques, we can say that E is more effective than F if $\frac{\sum_{i=1}^{z} E(i)}{n} < \frac{\sum_{i=1}^{z} F(i)}{n}$, where $E(i)$ and $F(i)$ are the number of statements need to be examined to locate the fault in the ith faulty version by E and F respectively.

- *Safety Change* (Masri and Assi, 2010): Safety indicates the relative suspiciousness of the faulty code. If the suspiciousness score of the faulty statement computed using a SBFL technique E, has been increased after relabeling the identified CC test cases, we consider that the safety of E got better. Otherwise, the safety of E got worse.
- *Precision Change* (Masri and Assi, 2010): Precision indicates the reduction in the search space for the faulty statement. If the number of statements that has a larger suspiciousness score than the faulty statement, computed using a SBFL technique E, has been reduced after relabeling the identified CC test cases, we consider that the precision of E got better. Otherwise, the precision of E got worse.

4.2.2. Evaluation metrics for CC test cases identification

To evaluate the effectiveness of competing methods in terms of identifying CC test cases, we use the following three major measurement metrics:

- *Precision*: Precision indicates the ratio of the number of correctly classified CC test cases to the total number of test cases classified as coincidentally correct, which is formally defined using Eq. (17). A high value for precision ratio indicates that for most cases both true coincidentally correct and true non-coincidentally correct test cases are identified properly.

$$Precision = \frac{|T_{CC} \cap T_{iCC}|}{|T_{iCC}|} \tag{17}$$

- *Recall*: Recall indicates the ratio of the number of correctly classified CC test cases to the total number of actual CC test cases, which is formally defined using Eq. (18). A low value for recall ratio indicates that only a subset of all coincidentally correct test cases is classified correctly by the corresponding CC identification method.

$$Recall = \frac{|T_{CC} \cap T_{iCC}|}{|T_{CC}|} \tag{18}$$

- *F-measure*: Precision and recall have some shortcomings when using alone. For example, if a CC identification method A just estimates one passed test case p as coincidentally correct and p is actually coincidentally correct, the precision of A will be 1. However, if there are other CC test cases, the recall of A will be low. Therefore, F-measure is needed which combines precision and recall in a single measure according to Eq. (19) (Witten et al., 2016).

$$F - measure = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{19}$$

In our experiments, we also utilize the Friedman test (Friedman, 1937, 1940) followed by the corresponding post-hoc Nemenyi test (Nemenyi, 1963) to evaluate FCCI based on sound statistics. The Friedman test and the Nemenyi test are nonparametric[15] counterparts for analysis of variance (ANOVA) and Tukey test parametric methods, respectively. These tests are recommended when the comparison includes more than two methods over multiple benchmarks (Demšar, 2006; Garcia and Herrera, 2008), and are adopted in several software testing studies (e.g., Jiang et al. (2008), Troya et al. (2018)). The Friedman test tests whether there is a difference in the performance among the competing CC identification methods over benchmarks. Therefore, the null-hypothesis being tested is that all methods perform the same and the observed differences are merely random. The Friedman test first ranks the methods from best to worse for each benchmark separately and then computes the average rank for each method based on all datasets. If the averages are very different, the p-value will be small (Demšar, 2006; Garcia and Herrera, 2008). If we can conclude that there are some performance differences among CC identification methods (i.e., the null-hypothesis is rejected), we can proceed with the post-hoc test to find out which methods actually differ. The Nemenyi post-hoc test finds the concrete pairwise comparisons which produce differences. If the difference of average ranks between the two methods is larger than the value of the critical difference (CD) of Nemenyi test, it can be concluded that the performance difference between them is significant. Furthermore, to reveal the volume of differences, Cohen's d (Cohen, 1988) was used to evaluate the effect size, i.e., determine how much a technique improves with respect to other. To interpret the effect size, we use the interpretation given by Cohen: $d < 0.2$ means trivial; values over 0.2 indicates a small ($0.2 \leq d < 0.5$), medium ($0.5 \leq d < 0.8$), large ($0.5 \leq d < 0.8$) or very large difference ($d > 1.3$).

4.3. Methodology

Using the same set of benchmarks, we compared the effectiveness of FCCI with the four state-of-the-art approaches: clustering-based technique proposed by Miao et al. (2012) (Clustering for short), ensemble-SVM based method proposed by Xue et al. (2014) (SVM for short), Masri et al. method (Masri and Assi, 2010) (Masri for short), and the recent work proposed by Feyzi and Parsa (2018c) (Feyzi for short).

To answer RQ1, we have conducted a set of experiments to evaluate the performance of four state-of-the-art SBFL techniques, namely DStar[2]-DStar[3] (Wong et al., 2014), Op[2] (Naish et al., 2011), and Ochiai (Abreu et al., 2006) before and after identifying CC test cases by each competing CC identification method. We choose these techniques because of two main reasons. Firstly, these techniques and specially Ochiai, have been widely used to assess previous CC identification approaches (Masri and Assi, 2014; Xue et al., 2014; Feyzi and Parsa, 2018c,b). Secondly, it has been shown that they are among the best performing SBFL formulas in locating faults (Wong et al., 2016; Xie et al., 2013; Pearson et al., 2017; Wong et al., 2014; Tang et al., 2017; Le et al., 2013). For instance, Xie et al. (2013) theoretically analyzed 30 formulas, partitioned them into equivalent classes (either ER groups or individual formulas), and established the relative ordering of the accuracy values of most of them. They proved that formulas in ER1 group including Op[2] are maximal. Tang et al. (2017) expanded the formula graphs proposed by Xie et al. (2013) with additional nodes by computing the accuracy of other SBFL formulas including DStar[2] and DStar[3]. They show that DStar[2] and DStar[3] may also be maximal. That is, these formulas, together with ER1, perform better than other formulas, but they were incomparable among themselves because none is statistically ranked ahead of another. Wong et al. (2014) also demonstrate the superiority of DStar[*] over many SBFL techniques and show that DStar[3] outperforms DStar[2]. Moreover, there are other studies reporting the promising results of Ochiai and even its superiority over Op[2] in some situations (Pearson et al., 2017; Le et al., 2013).

It should be noted that the suspiciousness score that is assigned to each program statement by SBFL techniques may not be always unique. In such a situation, statements with the same

[15] These tests are considered nonparametric because they do not assume that the data originates from any particular distribution family.

suspiciousness score are tied for the same position in the ranking and this could adversely affect the accuracy of the ANSE and EXAM score metrics (Steimann et al., 2013; Ali et al., 2009). In this regard, assuming that the sorting function breaks ties arbitrary, we used the expected value for the position of such statements in the computation of these metrics (Pearson et al., 2017; Steimann et al., 2013; Ali et al., 2009). To answer RQ2, we compared the competing methods in terms of their effectiveness in identifying CC test cases by obtaining the actual CC test cases for each program version and using precision, recall, and F-measure metrics. RQ3 is also addressed by performing a set of experiments on the multiple-fault version of programs. Finally, we conducted a sensitivity analysis to investigate the impact of different configurations on the performance of FCCI to answer RQ4.

In this study, to fine-tune the parameters of the proposed fuzzy expert system, and also to determine the operators' impact factor for FM factor, we utilized a subset of benchmarks including all programs of the *Siemens* suite, two programs from Unix utilities, namely *gzip* and *space*, and also two programs from the *Defects4j* suite, namely *JFreeChart* and *Apache commons-math*. All experiments were run on a laptop equipped with an Intel Core i7 2.7 GHz processor with 8 GB of memory.

4.4. Experimental results and discussion

In this section we thoroughly describe the experiments and present the evaluation results.

4.4.1. Empirical results for fault localization

In the first type of experiments, to answer RQ1, we study the performance of FCCI on improving the effectiveness of SBFL techniques and start the comparison by the ANSE metric. To this end, the effectiveness of DStar^2, DStar^3, Op^2, and Ochiai are initially measured using the ANSE metric on all subject programs. Then using FCCI, clustering, SVM, Masri and Feyzi methods, CC test cases are identified and relabeled from *passed* to *failed*. Finally, we applied SBFL techniques on all subject programs again and computed the new ANSE values. Table 10 reports the obtained results. The results demonstrate that FCCI outperforms other CC identification methods in terms of improving the effectiveness of SBFL techniques on all subject programs from ANSE metric points of view. For example, using pure DStar^3 on *sed*, it is required to examine 156.3 statements on average to reach the first faulty statement. However, after performing CC identification and relabeling the identified CC test cases, this value is reduced to 111.4, 139.1, 123.6, 129.4 and 140.9 for FCCI, Masri, Feyzi, SVM and Clustering techniques, respectively.

Although the results are encouraging for FCCI, since the comparison is based on the average of all faulty versions for each subject program, we also used the Friedman test to present an evaluation from the statistical point of view. We perform this test for the experimental results of each SBFL technique separately. In this regard, the null hypothesis (H_0) states that there is not a statistically significant difference between the results obtained by different methods using a particular SBFL technique, while the alternative hypothesis (H_1) states that at least two methods have significantly different performance. The resulting p-values were $< 1^{-6}$ for the results of four SBFL techniques, leading us to reject H_0 for all of them. In order to compare CC identification methods to determine the specific techniques with statistically significant differences, we proceed with the Nemenyi post-hoc test. For $\alpha = 0.05$, the computed value for CD is 0.30. The pairwise comparisons revealed statistically significant differences between FCCI and other techniques using DStar^2, DStar^3, Op^2, and Ochiai SBFL techniques. For example, the performance difference between DStar^2-FCCI and DStar^2-Feizi is significant because the difference of their average ranks, which is 1.2, is greater than CD. We then conducted effect size analysis using Cohen's d and provided the results in Table 11. We can see that the improvements of FCCI over other techniques are not negligible.

In summary, from Tables 10 and 11, we can draw the conclusion that FCCI has the best performance to improve the effectiveness of SBFL techniques followed by the Feizi method. To complete our comparisons, we have also investigated the percentage of faults that have been localized by examination of the top-5, top-10 and top-200 statements of the output ranking of different DStar^3-based methods. The results presented in Table 12 show the superiority of FCCI.

Next, we discuss the effectiveness of the competing CC identification methods on improving the precision and safety of the Dstar^3 SBFL technique. Figs. 10 and 11 illustrate the percentage of program versions which have a precision or safety improvement after applying CC identification methods, respectively. As can be seen, FCCI outperforms other methods on all subject programs. To take an example, applying FCCI, Masri, Feizi, SVM, and clustering methods result in precision improvement of the Dstar^3 on about 94.1%, 83.8%, 90%, 82.7%, and 85.6% of *grep*'s versions, respectively. This shows that FCCI performs better than other competing CC identification methods on increasing the suspiciousness scores of the faulty statements and also on elevating their rankings.

In summary, after applying FCCI, the precision and safety of the Dstar^3 have been improved for about 88% and 77% of all program versions with an average rate of 9.94% and 6.9%, respectively. For program versions with decreased precision and safety, the decrements are relatively small. More specifically, 12% of all program versions have decreased precision with an average rate of 0.87%, and 23% of all program versions have decreased safety with an average rate of 0.54%, which is relatively small.

We also used the Friedman test with Nemenyi post-hoc test, accompanied by effect size analysis to verify that FCCI leads to more improvement in precision and safety than other methods when using the DStar^3 SFBL technique. The resulting p-values of the Friedman test were $< 1^{-7}$ for the results of safety and precision change. The Nemenyi post-hoc test also reveals the significant difference between FCCI and other CC identification methods. Finally, we find that the effect sizes for the safety and precision change comparisons between DStar^3-FCCI and DStar^3-Feyzi (i.e., DStar^3-FCCI > DStar^3-Feyzi) are 1.069 and 1.177 respectively, which indicate large differences. Besides, the effect sizes for other comparisons are all above 1.3, which indicates very large differences.

Eventually, we investigate the effectiveness of FCCI and other methods with respect to the EXAM score. Fig. 12 presents EXAM score-based comparison between the competing methods on all subject programs using DStar^3 after addressing coincidental correctness. The *x*-axis represents the percentage of statements examined while the *y*-axis represents the percentage of faults are located by the examination of an amount of code less than or equal to the corresponding value of the *x*-axis. The results indicate the superiority of our proposed fuzzy CC identification approach over the other methods. For example, based on Fig. 12(a), we find that on the *Siemens* suite, by examining less than 10% of the code, DStar^3-FCCI can locate 86.5% of faults. In contrast, by examining less than 10% of code, DStar^3-Feizi (the second best method) can only locate 78.1% of the faults. The percentages for DStar^3-Masri, DStar^3-SVM, and DStar^3-Clustering are 63.0%, 61.1%, and 55.2%, respectively.

In summary, these experiments answer RQ1 as follows: FCCI successfully improves the performance of SBFL techniques and outperforms state-of-the-art CC identification methods in terms of ANSE, EXAM score, precision change and also safety change metrics.

Table 10
Average number of statements examined with respect to all faulty versions.

Technique	Siemens	gzip	grep	sed	space	JFreeChart	Joda-Time	commons-lang	commons-math	Mockito	Closure compiler
DStar^2	38.6	141.7	255.2	161.8	109.8	5625.2	274.5	1642.8	3751.6	764.6	3923.1
DStar^2-FCCI	19.9	112.6	211.8	116.7	86.5	5540.0	219.8	1565.4	3682.7	719.1	3839.1
DStar^2-Masri	31.1	130.6	238.5	144.2	94.7	5602.5	243.2	1625.4	3732.2	752.9	3882.4
DStar^2-Feyzi	24.3	119.2	224.4	127.3	88.4	5576.7	224.8	1596.2	3710.1	729.6	3864.1
DStar^2-SVM	28.4	126.3	232.3	135.6	91.6	5594.5	236.6	1611.2	3721.4	741.3	3870.4
DStar^2-Clustering	30.1	132.7	236.8	142.4	96.2	5609.1	249.3	1631.6	3735.2	752.3	3905.4
DStar^3	35.5	135.9	251.6	156.3	106.3	5610.1	271.7	1635.1	3742.6	758.4	3907.6
DStar^3-FCCI	16.9	107.1	207.8	111.4	81.4	5532.6	215.6	1559.6	3669.5	711.5	3837.4
DStar^3-Masri	27.2	127.7	233.9	139.1	91.8	5589.2	238.4	1610.1	3730.8	744.5	3879.8
DStar^3-Feyzi	21.2	114.1	220.7	123.6	84.4	5569.4	217.6	1590.9	3695.6	721.9	3852.2
DStar^3-SVM	24.5	119.4	227.4	129.4	87.6	5580.7	228.3	1594.7	3711.4	719.3	3865.1
DStar^3-Clustering	26.7	127.2	230.1	140.9	94.4	5596.8	240.1	1624.3	3719.3	748.6	3900.6
Op^2	42.6	159.1	279.4	182.8	125.8	5691.5	331.9	1688.8	3911.1	832.9	4124.3
Op^2-FCCI	22.1	122.4	236.1	133.1	97.0	5601.7	279.4	1621.7	3847.1	794.3	4020.9
Op^2-Masri	34.7	148.9	260.2	166.2	116.5	5676.2	306.8	1664.1	3889.7	820.4	4099.1
Op^2-Feyzi	27.8	130.8	243.1	141.9	103.4	5654.4	288.2	1642.2	3862.2	812.8	4063.3
Op^2-SVM	31.5	140.7	254.3	156.4	110.2	5669.7	295.6	1651.7	3875.9	816.6	4081.5
Op^2-Clustering	35.2	150.7	263.5	168.9	117.1	5680.8	303.4	1669.6	3886.2	827.7	4108.5
Ochiai	47.6	170.5	292.7	188.1	142.5	5659.6	303.4	1661.7	3829.5	801.7	4063.6
Ochiai-FCCI	25.3	128.1	248.8	135.8	105.5	5575.1	254.4	1592.1	3747.8	765.3	3945.2
Ochiai-Masri	38.8	155.4	279.2	170.5	136.7	5635.1	280.5	1650.1	3799.6	792.2	4023.4
Ochiai-Feyzi	30.4	135.2	261.1	149.2	121.2	5610.5	261.1	1627.2	3770.5	784.1	3986.5
Ochiai-SVM	35.1	144.7	271.4	160.2	128.6	5628.3	273.7	1638.7	3785.2	780.4	3997.9
Ochiai-Clustering	37.7	158.6	283.5	166.6	133.5	5640.3	284.2	1644.6	3791.4	796.3	4036.5

Table 11
Results of effect size analysis for ASNE metric.

Comparisons Winner > Loser	$\bar{d}$	Comparisons Winner > Loser	$\bar{d}$
DStar^2-FCCI >DStar^2	0.42	Op^2-FCCI > Op^2	0.46
DStar^2-FCCI > DStar^2-Masri	0.36	Op^2-FCCI > Op^2-Masri	0.39
DStar^2-FCCI > DStar^2-Feyzi	0.22	Op^2-FCCI > Op^2-Feyzi	0.21
DStar^2-FCCI > DStar^2-SVM	0.26	Op^2-FCCI > Op^2-SVM	0.24
DStar^2-FCCI > DStar^2-Clustering	0.31	Op^2-FCCI > Op^2-Clustering	0.35
DStar^3-FCCI > DStar^3	0.43	Ochiai-FCCI > Ochiai	0.40
DStar^3-FCCI > DStar^3-Masri	0.36	Ochiai-FCCI> Ochiai-Masri	0.34
DStar^3-FCCI > DStar^2-Feyzi	0.23	Ochiai-FCCI> Ochiai-Feyzi	0.24
DStar^3-FCCI > DStar^3-SVM	0.28	Ochiai-FCCI> Ochiai-SVM	0.29
DStar^3-FCCI > DStar^3-Clustering	0.34	Ochiai-FCCI> Ochiai-Clustering	0.30

Table 12
Percentage of failures whose faulty statements appear within the top-5, top-10 and top-200 of the DStar^3-based methods' output ranking.

Debugging Scenario									
Technique	Best case Scenario			Worst case scenario			Average case scenario		
	top-5	top-10	top-200	top-5	top-10	top-200	top-5	top-10	top-200
DStar^3	33%	40%	85%	19%	26%	59%	21%	28%	70%
DStar^3-FCCI	47%	52%	96%	27%	38%	71%	33%	41%	83%
DStar^3-Masri	34%	43%	87%	18%	25%	63%	21%	30%	72%
DStar^3-Feyzi	39%	47%	93%	24%	29%	66%	29%	36%	79%
DStar^3-SVM	35%	45%	89%	20%	28%	63%	26%	33%	77%
DStar^3-Clustering	33%	42%	87%	19%	25%	61%	23%	30%	74%

4.4.2. Empirical results for CC identification

To answer RQ2, in this section we study the effectiveness of FCCI and other competing methods in terms of identifying CC test cases using precision, recall, and F-measure metrics. Table 13 reports the average results for all subject programs. As can be seen, the average precision, recall and F-measure of FCCI is 92.45%, 93.00%, and 92.71% respectively, which indicates that FCCI is more accurate in the identification of CC test cases and effectively identifies most of the true coincidentally correct test cases. Table 14 presents the detailed experimental results for each subject program. Overall, FCCI outperforms all competing methods in terms of all evaluation metrics on all subject programs. This reveals that the presented fuzzy expert system with regard to the proposed CC identification factors and also the designed fuzzy rules perform well to estimate CC likelihoods and this helps FCCI to more accurately identify CC test cases.

Table 13
The average performances of FCCI and other competing methods on all subject programs.

Metric	FCCI	Masri	Feyzi	SVM	Clustering
Precision	92.54%	78.09%	86.54%	80.18%	76.09%
Recall	92.90%	72.90%	87.81%	78.45%	68.81%
F-measure	92.71%	75.37%	87.14%	79.26%	72.23%

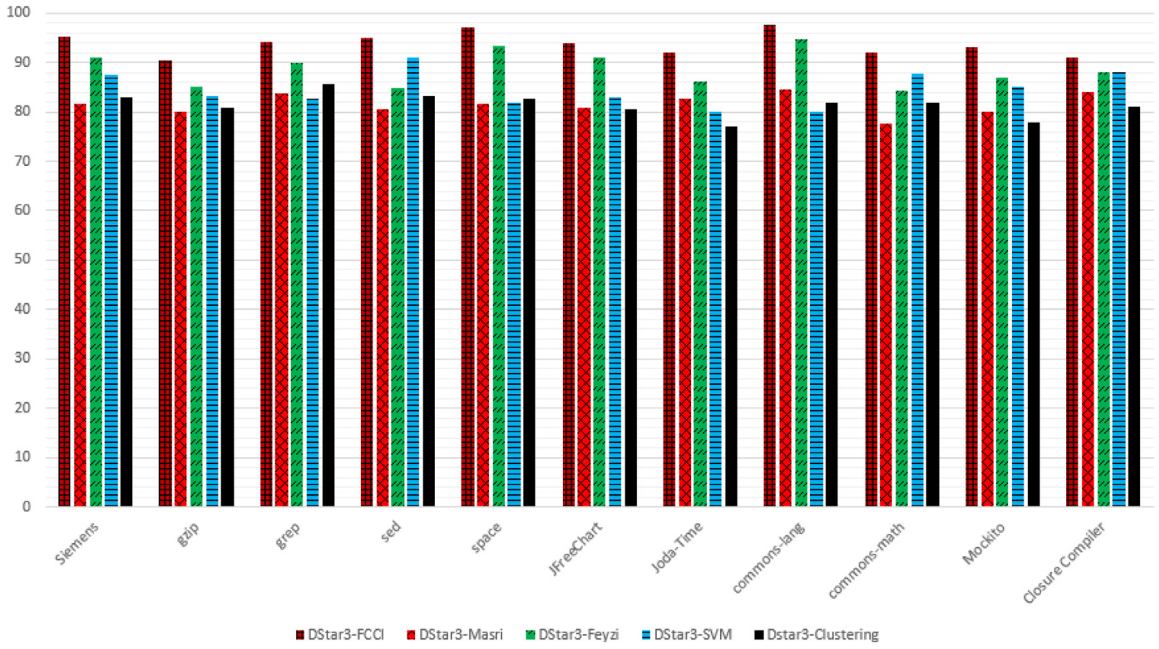

Fig. 10. Improvement of the DStar[3]'s precision on all subject programs after performing different CC identification methods.

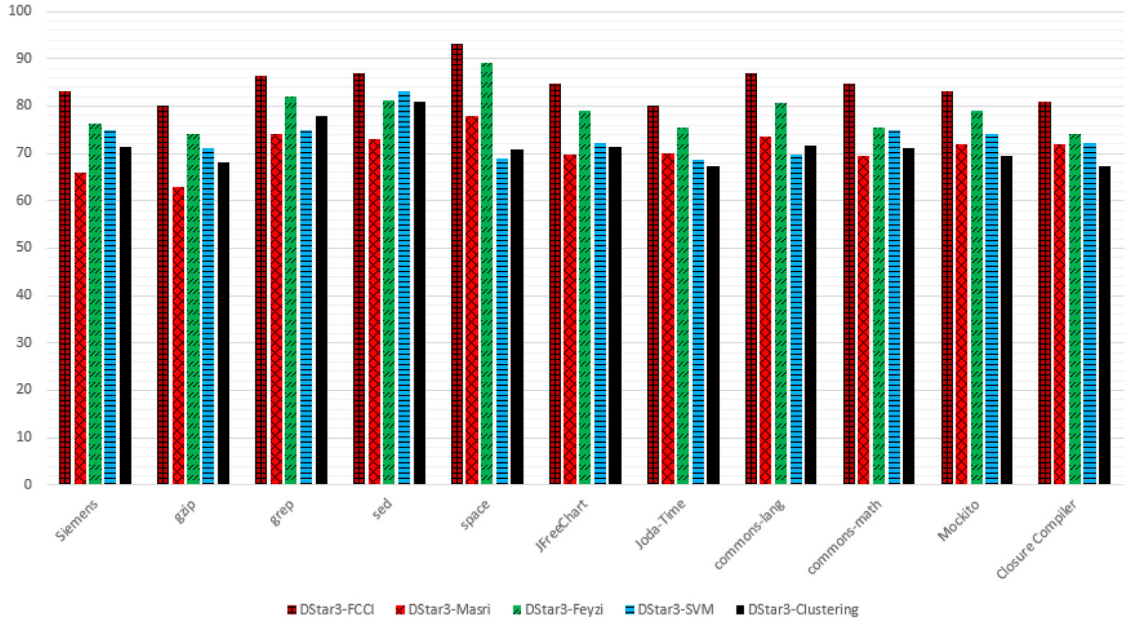

Fig. 11. Improvement of the DStar[3]'s safety on all subject programs after performing different CC identification methods.

4.4.3. Empirical results on multiple-fault programs

To answer RQ3, we evaluate the effectiveness of FCCI in the case of multi-bug programs. To this end, we combined different versions of programs and selected 30 different faulty versions with 3, 4 and 5 bugs for *gzip*, *grep*, *sed*, *space*, *JFreeChart*, *Joda-Time*, *commons-lang*, and *commons-math*, *Mockito*, and *Closure Compiler*. Then, we applied FCCI, Masri, Feyzi, SVM, and Clustering methods to these multi-faults versions using the one-fault-at-a-time approach. In this approach, one fault is identified and fixed, and then the fault localization process is repeated by re-running test cases to detect subsequent failures, locating the next fault and fixing it. The average number of statements required to be examined to locate the first bug using the DStar[3] SBFL technique before and after addressing coincidental correctness by the competing methods are reported in Table 15. To take an example, the average number of statements examined by DStar[3], DStar[3]-FCCI, DStar[3]-Masri, DStar[3]-Feyzi, DStar[3]-SVM, and DStar[3]-Clustering to locate the first fault in the 3-fault versions of *space* is 180.23, 155.11, 176.24, 162.31, 207.01, and 192.11, respectively.

From Table 15, we can see that on all the 3- and 4-bug, and also several 5-bug subject programs, FCCI outperforms other CC identification methods. We also utilized statistical tests to statistically compare the performance of CC identification methods on multiple-fault programs and validate the superiority of FCCI. In the Friedman test, the resulting p-values were $< 1^{-3}$ for the results of DStar[3]. Since the null hypothesis was rejected by the Friedman test, we proceed with the Nemenyi post-hoc test to examine whether FCCI is statistically significantly better than other CC identification methods on multiple-fault programs. This test showed statistically significant differences between FCCI

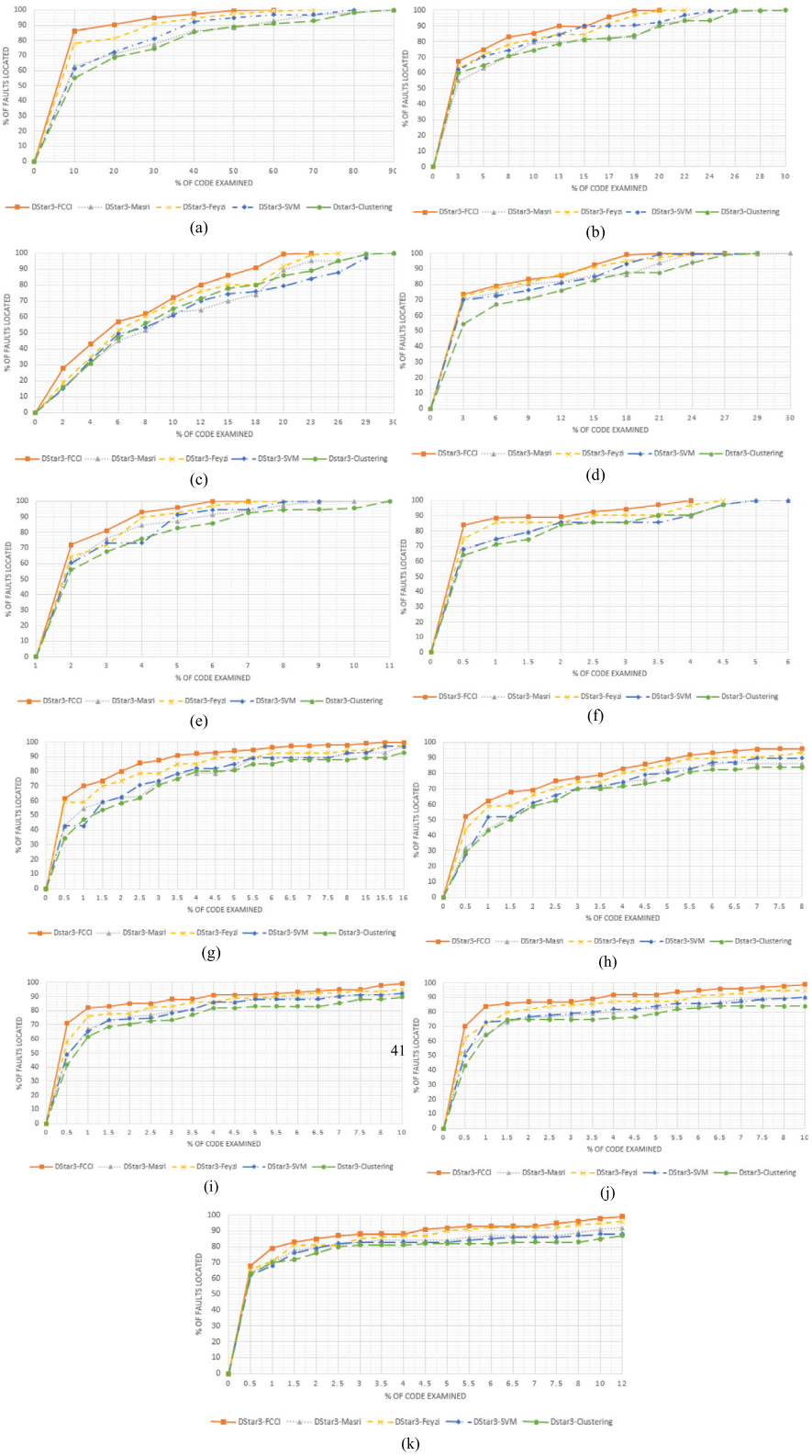

Fig. 12. EXAM score-based comparison on all subject programs. (a) *Siemens* suite, (b) *gzip*, (c) *grep*, (d) *sed*, (e) *space*, (f) *Joda-Time*, (g) *JFreeChart*, (h) *commons-lang*, (i) *commons-math*, (j) *Mockito framework*, (k) *Closure Compiler*.

and other techniques since the differences between the average rank of FCCI and the other methods are larger than the critical difference value of the Nemenyi test. To reveal the volume of differences, the effect size analysis is also conducted using Cohen's d. Table 16 depicts the effect size statistics. We can see that the improvements of FCCI over other techniques on multiple-fault programs are not negligible.

Table 14
CC test case identification performances of FCCI and other competing methods for each subject program.

Program	Metric	FCCI	Masri	Feyzi	SVM	Clustering
Siemens	Precision	94%	83%	87%	80%	77%
	Recall	94%	76%	91%	82%	72%
	F-measure	94.00%	79.34%	88.95%	80.98%	74.41%
gzip	Precision	96%	80%	93%	82%	78%
	Recall	95%	72%	90%	79%	68%
	F-measure	95.49%	75.78%	91.47%	80.47%	72.65%
grep	Precision	94%	82%	88%	79%	79%
	Recall	94%	69%	89%	76%	65%
	F-measure	94.00%	74.94%	88.49%	77.47%	71.31%
sed	Precision	92%	81%	86%	86%	82%
	Recall	96%	78%	92%	83%	74%
	F-measure	93.95%	79.47%	88.89%	84.47%	77.79%
space	Precision	92%	82%	85%	81%	76%
	Recall	95%	76%	89%	77%	70%
	F-measure	93.47%	78.88%	86.95%	78.94%	72.87%
JFreeChart	Precision	91%	75%	81%	78%	74%
	Recall	89%	71%	82%	72%	66%
	F-measure	89.98%	72.94%	81.49%	74.88%	69.77%
Joda-Time	Precision	95%	79%	90%	83%	80%
	Recall	92%	72%	85%	74%	69%
	F-measure	93.45%	75.33	87.42%	78.24%	74.09%
commons-lang	Precision	92%	72%	85%	75%	74%
	Recall	93%	70%	88%	80%	71%
	F-measure	92.49%	70.98%	86.47%	77.41%	72.46%
commons-math	Precision	90%	75%	83%	78%	73%
	Recall	91%	72%	84%	77%	64%
	F-measure	90.49%	73.46%	83.49%	77.49	68.20%
Mockito	Precision	90%	74%	86%	79%	71%
	Recall	92%	71%	89%	80%	67%
	F-measure	90.98%	72.46%	87.47%	79.49%	68.94%
Closure compiler	Precision	92%	76%	88%	81%	73%
	Recall	91%	75%	87%	83%	71%
	F-measure	91.49%	75.49%	87.49%	81.98%	71.98

Table 15
Average number of statements examined to locate the first bug in multiple-fault programs using DStar^3 SBFL technique.

Program	# of faults	DStar^3	DStar^3-FCCI	DStar^3-Masri	DStar^3-Feyzi	DStar^3-SVM	DStar^3-Clustering
gzip	3-fault	129.24	105.53	123.14	114.01	126.33	135.84
	4-fault	116.92	90.27	120.31	103.79	136.84	136.12
	5-fault	147.01	143.13	152.99	147.18	159.22	146.39
grep	3-fault	152.74	122.73	143.23	132.02	147.84	146.23
	4-fault	236.92	197.18	215.36	205.42	245.15	247.84
	5-fault	290.22	287.32	297.61	295.96	324.34	309.06
sed	3-fault	88.41	79.87	107.35	88.61	106.43	95.43
	4-fault	116.29	113.14	128.87	120.89	133.17	131.83
	5-fault	168.14	179.49	187.43	185.78	206.39	209.65
space	3-fault	180.23	155.11	176.24	162.31	207.01	192.11
	4-fault	162.12	145.19	162.76	150.27	180.49	158.98
	5-fault	228.93	231.33	252.68	235.98	267.39	254.25
JFreeChart	3-fault	4705.2	4650.4	4697.1	4677.7	4717.4	4730.7
	4-fault	3801.4	3789.6	3813.9	3801.7	3804.5	3800.1
	5-fault	5460.1	5487.9	5489.4	5483.1	5489.2	5504.5
Joda-Time	3-fault	297.9	282.1	293.2	288.1	302.4	307.4
	4-fault	320.1	302.2	316.7	309.3	322.1	332.7
	5-fault	293.1	299.8	326.4	303.9	314.3	315.8
commons-lang	3-fault	1657.4	1617.3	1648.1	1630.4	1641.3	1679.4
	4-fault	2242.7	2196.2	2240.7	2209.1	2244.9	2229.6
	5-fault	2370.6	2380.3	2398.4	2389.3	2408.1	2401.7
commons-math	3-fault	3652.7	3640.7	3660.9	3648.5	3656.3	3664.2
	4-fault	3947.5	3928.4	3951.3	3935.5	3954.3	3960.4
	5-fault	3281.6	3288.3	3294.6	3291.6	3298.9	3268.7
Mockito	3-fault	783.4	757.7	781.2	768.4	789.1	792.1
	4-fault	825.3	798.5	831.2	810.2	833.6	831.7
	5-fault	876.9	877.8	893.1	881.4	895.3	902.5
Closure compiler	3-fault	2427.5	2395.1	2439.3	2412.7	2421.4	2451.3
	4-fault	2953.4	2934.7	2955.8	2947.2	2951.5	2957.2
	5-fault	3256.7	3258.4	3281.5	3267.1	3279.4	3280.3

Table 16
Results of effect size analysis for ASNE metric on multiple fault programs.

ComparisonsWinner > Loser	$\overline{d}$
DStar^3-FCCI > DStar^3	0.31
DStar^3-FCCI > DStar^3-Masri	0.24
DStar^3-FCCI > DStar^3-Feyzi	0.22
DStar^3-FCCI > DStar^3-SVM	0.26
DStar^3-FCCI > DStar^3-Clustering	0.27

4.4.4. Sensitivity analysis

Finally, in this section, we investigate the impact of different configurations on the effectiveness of FCCI to answer RQ4. To this end, firstly, we examine the impact of the δ threshold on the performance of FCCI. In this regard, we conducted a set of experiments with three threshold values (i.e., $\delta = [0.4, 1], \delta = [0.5, 1], \delta = [0.6, 1]$) on all benchmarks using Dstar^3 and achieved our best performance with $\delta = [0.5, 1]$. Table 17 reports the results. In fact, using other threshold values increases the rates of false positives and/or false negatives and this adversely affects the performance of FCCI.

To evaluate the impact of individual CC identification factors on the performance of FCCI, we repeated the experiments on all benchmarks using Dstar^3 while excluding one factor each time. To exclude a factor $f \in \{SS, CR, FM, SF, SPF\}$, the corresponding rules are removed from the rule base. The results are depicted in Table 18. We use $FCCI_F$ to denote FCCI with full set of factors, and $FCCI_{F-f}$ to denote FCCI while attribute f being removed. Also, the second column represents the number of rules remained in the rule base after excluding the corresponding factor.

In summary, form Table 18, we can draw the conclusion that each of the proposed factors has a direct impact on the effectiveness of FCCI, and this shows the necessity of using them.

Also, to show how different values of the impact factors, which have been used in the calculation of fault-masking factor, affect the performance of FCCI, we conducted a set of experiments using two sets of values: for the first set, all the impact factors are subtracted by 0.05 (i.e., $FCCI_{FM-0.05}$), and for the second one, they are added by 0.05 (i.e., $FCCI_{FM+0.05}$). Table 19 reports the results, where $FCCI_{FM}$ denotes FCCI with the impact factors depicted in Table 7. A conclusion could be made from Table 19 is that the impact factors are tuned well using the adopted Simulated Annealing algorithm.

To investigate the influence of the number of failing tests on the performance of the FCCI and other competing methods, we randomly select a variable number of failed test cases and evaluate the methods using the newly generated test suites. We repeated this experiment six times with 80, 70, 60, 50, 40, 30% of failed test cases, and for space reasons report the average results (see Table 20). For each of these experiments, we involve program versions according to the required number of failed test cases. More specifically, since the number of failed executions is comparatively small as compared to all available executions, some selections of failed executions are not possible for some of the faulty versions.

As expected, the results reveal that utilizing a fewer number of failed executions adversely affects the performance of pure Dstar^3, FCCI, and also other competing methods. However, as can be seen, despite the degradation of the FCCI's performance, it still outperforms other competing methods when utilizing a smaller number of failed test cases.

5. Threats to validity

The main threats to the external validity of our approach, which addresses the generalization ability of our experimental results and findings, are related to the choice of subject programs and also the fault types. As explained in Section 4.1, we chose a set of 17 open source and popular subject programs ranging from small- to large-scale that are widely used in the literature to compare all sorts of fault localization related techniques. These programs vary dramatically in terms of size, functionality, number of faulty versions, types of faults and also number of test cases. All of the faults in the *Siemens* suite's programs and most of the faults in the selected programs from Unix utilities are hand-seeded or obtained from mutations. Just *space* and some versions of *sed* contain real faults. Since it is unclear whether artificial bugs capture true characteristics of real bugs in real programs, we employ 6 programs from *Defects4j* suite to better evaluate FCCI on real faults. *Defects4j* is one of the largest available datasets of well-organized real-world java bugs which facilitates using real multiple-fault programs in the experiments. All of these, allow us to better generalize the findings and results of this paper. However, performing the experiments on more programs can better assess the effectiveness of the FCCI.

The threat to construct validity is related to the results measurements. To minimize this threat, we employed widely used measurements in fault localization and CC test case identification. In terms of fault localization accuracy, we used ANSE, EXAM score, safety change, and precision change to compare the competing CC identification methods in improving the effectiveness of Ochiai, Op^2, Dstar^2, and Dstar^3 SBFL techniques, which are the most effective SBFL techniques in case of both artificial and real faults (Pearson et al., 2017). In terms of CC identification accuracy, we used precision, recall, and F-measure. Besides, in the experiments, we also utilized the Friedman test followed by the corresponding post-hoc Nemenyi test to evaluate FCCI based on sound statistics. However, these metrics may not be the best ones, and in practice, there may be other metrics demonstrating the effectiveness of CC identification and fault localization methods. We plan to use more metrics to measure the experimental results as future work.

The threats to the internal validity of our approach are related to our assumptions. We assumed that the tests' oracle is available (i.e., the execution results of test cases can be decided as *passed* or *failed*) and the faults must be deterministic (i.e., the execution results of test cases are not affected by the run-time environment). These assumptions are widely adopted by previous studies and we believe the threats are limited.

6. Conclusion and future works

Spectrum-based fault localization is one of the most studied and evaluated fault localization approaches, which leverages the coverage information of test executions as well as their results to assign a suspiciousness score to each program element that reflects their likelihood of being faulty. Although SBFL techniques have shown promising results, their performance can be adversely affected due to the presence of coincidental correct test cases in the test suite. Such test cases execute faulty statements but do not cause failures. Given that coincidental correctness is prevalent and reduces the effectiveness of SBFL techniques by reducing the suspiciousness score of faulty statements, it is necessary to identify CC test cases precisely and eliminate their effects from test suites.

In this paper, we propose several important CC identification factors that have a direct impact on increasing the likelihood of identifying coincidental correctness, and model the CC identification process as a decision making system. In this regard, we construct a fuzzy expert system and propose a novel fuzzy CC identification method, namely FCCI. FCCI at first extracts the values of the proposed CC identification factors for each passed

Table 17
Average number of statements examined with respect to all faulty versions with three values for δ threshold.

Threshold value	Siemens	gzip	grep	sed	space	JFreeChart	Joda-Time	commons-lang	commons-math	Mockito	Closure compiler
[0.4, 1]	22.1	115.9	216.1	111.2	82.1	5571.1	219.6	1571.1	3691.2	726.8	3896.5
[0.5, 1]	16.9	107.1	207.8	111.4	81.4	5532.6	215.6	1559.6	3669.5	711.5	3837.4
[0.6, 1]	28.4	122.1	228.8	132.3	88.1	5565.8	214.1	1582.1	3781.5	744.5	4057.2

Table 18
Average number of statements examined with respect to all faulty versions with different CC identification factors.

Technique	#Rules	Siemens	gzip	grep	sed	space	JFreeChart	Joda-Time	commons-lang	commons-math	Mockito	Closure compiler
$FCCI_F$	41	16.1	107.1	207.8	111.4	81.4	5532.6	215.6	1559.6	3669.5	711.5	3837.4
$FCCI_{T-SS}$	24	19.8	112.6	218.7	120.8	83.7	5560.2	217.3	1587.9	3687.4	718.4	3845.5
$FCCI_{T-CR}$	24	19.1	111.4	215.4	120.5	83.1	5555.4	217.1	1579.4	3691.3	717.6	3843.6
$FCCI_{T-FM}$	17	17.9	109.6	211.7	118.7	82.4	5549.4	216.2	1570.3	3681.2	715.1	3841.4
$FCCI_{T-SF}$	36	18.5	110.1	213.4	118.9	82.9	5557.6	217.3	1583.3	3684.3	716.6	3841.7
$FCCI_{T-SPF}$	39	17.6	108.9	211.8	115.8	82.1	5543.9	216.7	1572.4	3680.8	714.3	3842.1

Table 19
Average number of statements examined with respect to all faulty versions with different impact factors.

Technique	Siemens	gzip	grep	sed	space	JFreeChart	Joda-Time	commons-lang	commons-math	Mockito	Closure compiler
$FCCI_{FM-0.05}$	17.1	108.0	210.2	114.3	81.8	5545.3	216.3	1564.2	3675.7	712.9	3839.8
$FCCI_{FM}$	16.1	107.1	207.8	111.4	81.4	5532.6	215.6	1559.6	3669.5	711.5	3837.4
$FCCI_{FM+0.05}$	17.6	108.3	209.4	112.9	81.9	5541.7	216.8	1565.1	3673.8	713.8	3840.4

Table 20
Average number of statements examined with respect to all faulty versions with limited number of failed test cases.

Technique	Siemens	gzip	grep	sed	space	JFreeChart	Joda-Time	commons-lang	commons-math	Mockito	Closure compiler
DStar^3	68.7	286.4	412.8	293.7	204.6	7118.1	385.1	2536.6	5134.9	1329.2	6233.7
DStar^3-FCCI	55.4	257.3	380.4	275.5	191.9	6875.3	350.4	2490.3	5011.8	1297.4	6181.8
DStar^3-Masri	64.9	279.4	408.3	287.3	201.8	7084.1	380.4	2524.2	5107.5	1321.5	6228.5
DStar^3-Feyzi	61.3	263.1	399.5	280.5	199.2	7003.4	369.7	2518.7	5087.4	1317.1	6214.5
DStar^3-SVM	60.8	265.4	405.7	284.9	198.7	7036.5	376.8	2507.5	5101.9	1309.5	6212.7
DStar^3-Clustering	63.7	282.5	408.9	290.4	203.56	7052.7	382.1	2530.3	5120.8	1320.1	6224.1

test case and then feeds them to the designed fuzzy expert system to estimate their likelihood of being coincidental correct using the designed fuzzy rules, which effectively correlate different CC identification factors. The estimated CC likelihoods are then used to identify CC test cases. We extensively evaluated FCCI on 17 popular and open source subject programs ranging from small- to large-scale containing both artificial and real faults. Our experimental results indicate that FCCI is highly effective in addressing the coincidental correctness and reducing the developers' efforts to locate faults. It outperforms state-of-the-art CC identification methods in terms of accurate identification of CC test cases and also improving the effectiveness of SBFL techniques. This shows that the presented fuzzy expert system with regard to the proposed CC identification factors and also the designed fuzzy rules perform well to estimate CC likelihoods and this helps FCCI to more accurately identify CC test cases.

In the future, we plan to examine the impact of coverage refinement techniques on the effectiveness of FCCI. In addition, we plan to utilize other code complexity metrics especially the object-oriented metrics for Java programs to calculate the fault-proneness of program statements. We are also working to improve the performance of FCCI from different aspects. One of these improvements can be obtained by automatic analysis and tuning the proposed fuzzy expert system using proper optimization techniques. Besides, we will examine additional input attributes and how different ways of value handling and normalization may generate different results.

CRediT authorship contribution statement

Arash Sabbaghi: Software, Validation, Formal analysis, Investigation, Resources, Data curation, Writing - original draft, Visualization. **Mohammad Reza Keyvanpour:** Supervision, Validation, Data curation, Writing - review & editing, Project administration. **Saeed Parsa:** Supervision, Formal analysis, Resources, Writing - review & editing.

Declaration of competing interest

No author associated with this paper has disclosed any potential or pertinent conflicts which may be perceived to have impending conflict with this work. For full disclosure statements refer to https://doi.org/10.1016/j.jss.2020.110635.

References

Abraham, A., 2005. Rule-based expert systems. In: Handbook of Measuring System Design.

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

Abreu, R., Zoeteweij, P., Van Gemund, A.J., 2006. An evaluation of similarity coefficients for software fault localization. In: Dependable Computing, 2006. PRDC'06. 12th Pacific Rim International Symposium on. IEEE.

Ali, S., Andrews, J.H., Dhandapani, T., Wang, W., 2009. Evaluating the accuracy of fault localization techniques. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society.

Amine, K., 2019. Multiobjective simulated annealing: Principles and algorithm variants. Adv. Oper. Res. 2019.
Ammann, P.E., Knight, J.C., 1988. Data diversity: An approach to software fault tolerance. IEEE Trans. Comput. (4), 418–425.
Androutsopoulos, K., Clark, D., Dan, H., Hierons, R.M., Harman, M., 2014. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In: Proceedings of the 36th International Conference on Software Engineering.
Arthur, D., Vassilvitskii, S., 2007. K-means++: The advantages of careful seeding. In: Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied Mathematics.
Artzi, S., Dolby, J., Tip, F., Pistoia, M., 2010. Directed test generation for effective fault localization. In: Proceedings of the 19th International Symposium on Software Testing and Analysis. ACM.
Ball, T., Naik, M., Rajamani, S.K., 2003. From symptom to cause: localizing errors in counterexample traces. In: ACM SIGPLAN Notices. ACM.
Baluda, M., Denaro, G., Pezze, M., 2016. Bidirectional symbolic analysis for effective branch testing. IEEE Trans. Softw. Eng. 42 (5), 403–426.
Bandyopadhyay, A., 2012. Mitigating the effect of coincidental correctness in spectrum based fault localization. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE.
Bandyopadhyay, A., Ghosh, S., 2011. Proximity based weighting of test cases to improve spectrum based fault localization. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. IEEE Computer Society.
Bates, P.C., 1995. Debugging heterogeneous distributed systems using event-based models of behavior. ACM Trans. Comput. Syst. (TOCS) 13 (1), 1–31.
Borg, I., Groenen, P., 2003. Modern multidimensional scaling: theory and applications. J. Educ. Meas. 40 (3), 277–280.
Bowes, D., Hall, T., Petrić, J., 2018. Software defect prediction: do different classifiers find the same defects?. Softw. Qual. J. 26 (2), 525–552.
Catal, C., 2011. Software fault prediction: A literature review and current trends. Expert Syst. Appl. 38 (4), 4626–4636.
Catal, C., Diri, B., 2009. A systematic review of software fault prediction studies. Expert Syst. Appl. 36 (4), 7346–7354.
Chekam, T.T., Papadakis, M., Traon, Y.L., 2016. Assessing and comparing mutation-based fault localization techniques. arXiv preprint arXiv:1607.05512.
Clark, D., Hierons, R.M., 2012. Squeeziness: An information theoretic measure for avoiding fault masking. Inform. Process. Lett. 112 (8–9), 335–340.
Clark, D., Hierons, R.M., Patel, K., 2019. Normalised squeeziness and failed error propagation. Inform. Process. Lett. 149, 6–9.
Cohen, J., 1988. Statistical Power Analysis for the Behavioral Sciences. Lawrence Earlbam Associates, Hillsdale, NJ.
Dallmeier, V., Lindig, C., Zeller, A., 2005. Lightweight bug localization with AMPLE. In: Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging. ACM.
de Souza, H.A., Chaim, M.L., Kon, F., 2016. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. arXiv preprint arXiv:1607.04347.
Dejaeger, K., Verbraken, T., Baesens, B., 2012. Toward comprehensible software fault prediction models using bayesian network classifiers. IEEE Trans. Softw. Eng. 39 (2), 237–257.
Demšar, J., 2006. Statistical comparisons of classifiers over multiple data sets. J. Mach. Learn. Res. 7 (Jan), 1–30.
Deza, M.M., Deza, E., 2009. Encyclopedia of distances. In: Encyclopedia of Distances. Springer, pp. 1–583.
Dickinson, W., Leon, D., Podgurski, A., 2001. Finding failures by cluster analysis of execution profiles. In: Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society.
Fenton, N.E., Pfleeger, S.L., 1997. Software Metrics: A Rigorous and Practical Approach, second ed. Thomson Computer Press.
Feyzi, F., Parsa, S., 2018a. FPA-FL: Incorporating static fault-proneness analysis into statistical fault localization. J. Syst. Softw. 136, 39–58.
Feyzi, F., Parsa, S., 2018b. Kernel-based detection of coincidentally correct test cases to improve fault localization effectiveness. arXiv preprint arXiv:1803.09226.
Feyzi, F., Parsa, S., 2018c. A program slicing-based method for effective detection of coincidentally correct test cases. Computing 1–43.
Feyzi, F., Parsa, S., 2019. Inference: effective fault localization based on information-theoretic analysis and statistical causal inference. Front. Comput. Sci. 13 (4), 735–759.
Fitzsimmons, A., Love, T., 1978. A review and evaluation of software science. ACM Comput. Surv. 10 (1), 3–18.
Friedman, M., 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. J. Am. Stat. Assoc. 32 (200), 675–701.
Friedman, M., 1940. A comparison of alternative tests of significance for the problem of m rankings. Ann. Math. Stat. 11 (1), 86–92.
Gao, R., Wong, W.E., Chen, Z., Wang, Y., 2017. Effective software fault localization using predicted execution results. Softw. Qual. J. 25 (1), 131–169.
Garcia, S., Herrera, F., 2008. An extension onstatistical comparisons of classifiers over multiple data setsfor all pairwise comparisons. J. Mach. Learn. Res. 9 (Dec), 2677–2694.
Goel, A.L., 1985. Software reliability models: Assumptions, limitations, and applicability. IEEE Trans. Softw. Eng. (12), 1411–1423.
Hall, T., Beecham, S., Bowes, D., Gray, D., Counsell, S., 2011. A systematic literature review on fault prediction performance in software engineering. IEEE Trans. Softw. Eng. 38 (6), 1276–1304.
Halstead, M.H., 1977. Elements of Software Science, vol. 7. Elsevier, New York.
Han, J., Pei, J., Kamber, M., 2011. Data Mining: Concepts and Techniques. Elsevier.
Hartigan, J.A., Wong, M.A., 1979. Algorithm AS 136: A k-means clustering algorithm. J. R. Stat. Soc. Ser. C. Appl. Stat. 28 (1), 100–108.
He, P., Li, B., Liu, X., Chen, J., Ma, Y., 2015. An empirical study on software defect prediction with a simplified metric set. Inf. Softw. Technol. 59, 170–190.
Henry, S., Kafura, D., 1981. Software structure metrics based on information flow. IEEE Trans. Softw. Eng. (5), 510–518.
Hierons, R.M., 2006. Avoiding coincidental correctness in boundary value analysis. ACM Trans. Softw. Eng. Methodol. 15 (3), 227–241.
Hutchins, M., Foster, H., Goradia, T., Ostrand, T., 1994. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In: Software Engineering, 1994. Proceedings. ICSE-16. 16th International Conference on. IEEE.
Jiang, Y., Cukic, B., Ma, Y., 2008. Techniques for evaluating fault prediction models. Empir. Softw. Eng. 13 (5), 561–595.
Jones, J.A., Harrold, M.J., 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering. ACM.
Jones, J.A., Harrold, M.J., Stasko, J.T., 2001. Visualization for fault localization. In: Proceedings of ICSE 2001 Workshop on Software Visualization. Citeseer.
Just, R., Jalali, D., Ernst, M.D., 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ACM.
Kim, S., Lee, M., Lee, J., 2017. A study of fuzzy membership functions for dependence decision-making in security robot system. Neural Comput. Appl. 28 (1), 155–164.
Kim, H.-C., Pang, S., Je, H.-M., Kim, D., Bang, S.-Y., 2002. Support vector machine ensemble with bagging. In: Pattern Recognition with Support Vector Machines. Springer, pp. 397–408.
Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P., 1983. Optimization by simulated annealing. Science 220 (4598), 671–680.
Kotsiantis, S.B., Zaharakis, I., Pintelas, P., 2007. Supervised machine learning: A review of classification techniques. Emerg. Artif. Intell. Appl. Comput. Eng. 160, 3–24.
Le, T.-D.B., Thung, F., Lo, D., 2013. Theory and practice, do they match? a case with spectrum-based fault localization. In: Software Maintenance (ICSM), 2013 29th IEEE International Conference on. IEEE.
Lee, C.-C., 1990. Fuzzy logic in control systems: fuzzy logic controller. I. IEEE Trans. Syst. Man Cybern. 20 (2), 404–418.
Leon, D., Podgurski, A., Dickinson, W., 2005. Visualizing similarity between program executions. In: Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on. IEEE.
Leon, D., Podgurski, A., White, L.J., 2000. Multivariate visualization in observation-based testing. In: Proceedings of the 22nd International Conference on Software Engineering. ACM.
Lewis, J., Henry, S., 1989. A methodology for integrating maintainability using software metrics. In: Proceedings. Conference on Software Maintenance-1989. IEEE.
Li, Z., Li, M., Liu, Y., Geng, J., 2016. Identify coincidental correct test cases based on fuzzy classification. In: Software Analysis, Testing and Evolution (SATE), International Conference on. IEEE.
Li, Y., Liu, C., 2012. Using cluster analysis to identify coincidental correctness in fault localization. In: 2012 Fourth International Conference on Computational and Information Sciences. IEEE.
Li, Y., Liu, C., 2014. Identifying coincidental correctness in fault localization via cluster analysis. J. Softw. Eng. 8 (4), 328–344.
Liu, Y., Li, M., Wu, Y., Li, Z., 2019. A weighted fuzzy classification approach to identify and manipulate coincidental correct test cases for fault localization. J. Syst. Softw.

Liu, C., Zhang, X., Han, J., 2008. A systematic study of failure proximity. IEEE Trans. Softw. Eng. 34 (6), 826–843.
MacQueen, J., 1967. Some methods for classification and analysis of multivariate observations. In: Proceedings of the fifth Berkeley Symposium on Mathematical Statistics and Probability. Oakland, CA, USA.
Martinez, M., Durieux, T., Sommerard, R., Xuan, J., Monperrus, M., 2017. Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset. Empir. Softw. Eng. 22 (4), 1936–1964.
Masri, W., 2010. Fault localization based on information flow coverage. Softw. Test. Verif. Reliab. 20 (2), 121–147.
Masri, W., Abou-Assi, R., El-Ghali, M., Al-Fatairi, N., 2009. An empirical study of the factors that reduce the effectiveness of coverage-based fault localization. In: Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). ACM.
Masri, W., Assi, R.A., 2010. Cleansing test suites from coincidental correctness to enhance fault-localization. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE.
Masri, W., Assi, R.A., 2014. Prevalence of coincidental correctness and mitigation of its impact on fault localization. ACM Trans. Softw. Eng. Methodol. 23 (1), 8.
Masri, W., Assi, R.A., Zaraket, F., Fatairi, N., 2012. Enhancing fault localization via multivariate visualization. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. IEEE.
Masri, W., Podgurski, A., 2009. Measuring the strength of information flows in programs. ACM Trans. Softw. Eng. Methodol. 19 (2), 5.
McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.
Menzies, T., Greenwald, J., Frank, A., 2006. Data mining static code attributes to learn defect predictors. IEEE Trans. Softw. Eng. 33 (1), 2–13.
Menzies, T., Milton, Z., Turhan, B., Cukic, B., Jiang, Y., Bener, A., 2010. Defect prediction from static code features: current results, limitations, new approaches. Autom. Softw. Eng. 17 (4), 375–407.
Miao, Y., Chen, Z., Li, S., Zhao, Z., Zhou, Y., 2012. Identifying coincidental correctness for fault localization by clustering test cases. In: SEKE.
Miao, Y., Chen, Z., Li, S., Zhao, Z., Zhou, Y., 2013. A clustering-based strategy to identify coincidental correctness in fault localization. In. J. Softw. Eng. Knowl. Eng. 23 (05), 721–741.
Moon, S., Kim, Y., Kim, M., Yoo, S., 2014. Ask the mutants: Mutating faulty programs for fault localization. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation. IEEE.
Morales-Castañeda, B., Zaldívar, D., Cuevas, E., Maciel-Castillo, O., Aranguren, I., Fausto, F., 2019. An improved simulated annealing algorithm based on ancient metallurgy techniques. Appl. Soft Comput. 84, 105761.
Naish, L., Lee, H.J., Ramamohanarao, K., 2011. A model for spectra-based software diagnosis. ACM Trans. Softw. Eng. Methodol. 20 (3), 11.
Neelofar, N., Naish, L., Lee, J., Ramamohanarao, K., 2017. Improving spectral-based fault localization using static analysis. Softw. - Pract. Exp. 47 (11), 1633–1655.
Nemenyi, P., 1963. Distribution-Free Multiple Comparisons (Ph.D. thesis). Princeton University.
Pai, G.J., Dugan, J.B., 2007. Empirical analysis of software fault content and fault proneness using Bayesian methods. IEEE Trans. Softw. Eng. 33 (10), 675–686.
Papadakis, M., Le Traon, Y., 2014. Effective fault localization via mutation analysis: a selective mutation approach. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing.
Papadakis, M., Le Traon, Y., 2015. Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. 25 (5–7), 605–628.
Pappis, C.P., Siettos, C.I., 2014. Fuzzy reasoning. In: Search Methodologies. Springer, pp. 519–556.
Parsa, S., Naree, S.A., 2012. Software online bug detection: applying a new kernel method. IET Softw. 6 (1), 61–73.
Parsa, S., Vahidi-Asl, M., Asadi-Aghbolaghi, M., 2014. Hierarchy-debug: a scalable statistical technique for fault localization. Softw. Qual. J. 22 (3), 427–466.
Parsa, S., Vahidi-Asl, M., Naree, S.A., 2008. Finding causes of software failure using ridge regression and association rule generation methods. In: 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. IEEE.
Pearson, VII, K., 1895. Note on regression and inheritance in the case of two parents. Proc. R. Soc. London 58 (347–352), 240–242.
Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M.D., Pang, D., Keller, B., 2017. Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press.
Perez, A., Abreu, R., Riboira, A., 2014. A dynamic code coverage approach to maximize fault localization efficiency. J. Syst. Softw. 90, 18–28.
Podgurski, A., Masri, W., McCleese, Y., Wolff, F.G., Yang, C., 1999. Estimation of software reliability by stratified sampling. ACM Trans. Softw. Eng. Methodol. 8 (3), 263–283.
Rathore, S.S., Kumar, S., 2019. A study on software fault prediction techniques. Artif. Intell. Rev. 51 (2), 255–327.
Renieres, M., Reiss, S.P., 2003. Fault localization with nearest neighbor queries. In: Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. IEEE.
Ross, T.J., 2009. Fuzzy Logic with Engineering Applications. John Wiley & Sons.
Sabbaghi, A., Keyvanpour, M.R., 2017. State-based models in model-based testing: A systematic review. In: Knowledge-Based Engineering and Innovation (KBEI), 2017 IEEE 4th International Conference on. IEEE.
Sambariya, D.K., Prasad, R., 2017. Selection of membership functions based on fuzzy rules to design an efficient power system stabilizer. Int. J. Fuzzy Syst. 19 (3), 813–828.
Schneider, M., Langholz, G., Kandel, A., Chew, G., 1996. Fuzzy Expert System Tools. Wiley.
Shafeeq, A., Hareesha, K., 2012. Dynamic clustering of data with modified k-means algorithm. In: Proceedings of the 2012 Conference on Information and Computer Networks.
Shepperd, M., Ince, D.C., 1994. A critique of three metrics. J. Syst. Softw. 26 (3), 197–210.
Shu, T., Ye, T., Ding, Z., Xia, J., 2016. Fault localization based on statement frequency. Inform. Sci. 360, 43–56.
Singh, S.S., Chauhan, N., 2011. K-means v/s K-medoids: A comparative study. In: National Conference on Recent Trends in Engineering & Technology.
Singhal, A., 2001. Modern information retrieval: A brief overview. IEEE Data Eng. Bull. 24 (4), 35–43.
Steimann, F., Frenkel, M., Abreu, R., 2013. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM.
Suman, B., Kumar, P., 2006. A survey of simulated annealing as a tool for single and multiobjective optimization. J. Oper. Res. Soc. 57 (10), 1143–1160.
Tang, C.M., Chan, W., Yu, Y.T., Zhang, Z., 2017. Accuracy graphs of spectrum-based fault localization formulas. IEEE Trans. Reliab. 66 (2), 403–424.
Troya, J., Segura, S., Parejo, J.A., Ruiz-Cortés, A., 2018. Spectrum-based fault localization in model transformations. ACM Trans. Softw. Eng. Methodol. 27 (3), 1–50.
Turhan, B., Menzies, T., Bener, A.B., Di Stefano, J., 2009. On the relative value of cross-company and within-company data for defect prediction. Empir. Softw. Eng. 14 (5), 540–578.
Vessey, I., 1985. Expertise in debugging computer programs: A process analysis. Int. J. Man-Mach. Stud. 23 (5), 459–494.
Voas, J.M., 1992. PIE: A dynamic failure-based technique. IEEE Trans. Softw. Eng. 18 (8), 717–727.
Voas, J.M., Miller, K.W., 1993. Semantic metrics for software testability. J. Syst. Softw. 20 (3), 207–216.
Wang, L.-X., 1999. A Course in Fuzzy Systems. Prentice-Hall press, USA.
Wang, X., Cheung, S.-C., Chan, W.K., Zhang, Z., 2009. Taming coincidental correctness: Coverage refinement with context patterns to improve fault localization. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on. IEEE.
Wang, J., Kusiak, A., 2000. Computational Intelligence in Manufacturing Handbook. CRC Press.
Witten, I.H., Frank, E., Hall, M.A., Pal, C.J., 2016. Data Mining: Practical Machine Learning Tools and Techniques. Morgan Kaufmann.
Wong, W.E., Debroy, V., Choi, B., 2010. A family of code coverage-based heuristics for effective fault localization. J. Syst. Softw. 83 (2), 188–208.
Wong, W.E., Debroy, V., Gao, R., Li, Y., 2014. The DStar method for effective software fault localization. IEEE Trans. Reliab. 63 (1), 290–308.
Wong, W.E., Debroy, V., Golden, R., Xu, X., Thuraisingham, B., 2012a. Effective software fault localization using an RBF neural network. IEEE Trans. Reliab. 61 (1), 149–169.
Wong, W.E., Debroy, V., Xu, D., 2012b. Towards better fault localization: A crosstab-based statistical approach. IEEE Trans. Syst. Man Cybern. C 42 (3), 378–396.
Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F., 2016. A survey on software fault localization. IEEE Trans. Softw. Eng. 42 (8), 707–740.
Xie, X., Chen, T.Y., Kuo, F.-C., Xu, B., 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Trans. Softw. Eng. Methodol. 22 (4), 31.

Xue, X., Pang, Y., Namin, A.S., 2014. Trimming test suites with coincidentally correct test cases for enhancing fault localizations. In: Computer Software and Applications Conference (COMPSAC), 2014 IEEE 38th Annual. IEEE.

Yan, S., Chen, Z., Zhao, Z., Zhang, C., Zhou, Y., 2010. A dynamic test cluster sampling strategy by leveraging execution spectra information. In: Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE.

Yang, X., Liu, M., Cao, M., Zhao, L., Wang, L., 2015. Regression identification of coincidental correctness via weighted clustering. In: 2015 IEEE 39th Annual Computer Software and Applications Conference. IEEE.

Zadeh, L.A., 1983. The role of fuzzy logic in the management of uncertainty in expert systems. Fuzzy Sets and Systems 11 (1), 199–227.

Zedeh, L.A., 1965. Fuzzy sets. Inf. Control 8 (3), 338–353.

Zhang, H., 2008. On the distribution of software faults. IEEE Trans. Softw. Eng. 34 (2), 301–302.

Zhao, J., Bose, B.K., 2002. Evaluation of membership functions for fuzzy logic controlled induction motor drive. In: IEEE 2002 28th Annual Conference of the Industrial Electronics Society. IECON 02, IEEE.

Zhou, X., Wang, H., Zhao, J., 2015. A fault-localization approach based on the coincidental correctness probability. In: 2015 IEEE International Conference on Software Quality, Reliability and Security. IEEE.

Zhu, H., Hall, P.A., May, J.H., 1997. Software unit test coverage and adequacy. ACM Comput. Surv. 29 (4), 366–427.

Zou, D., Liang, J., Xiong, Y., Ernst, M.D., Zhang, L., 2019. An empirical study of fault localization families and their combinations. IEEE Trans. Softw. Eng..

Arash Sabbaghi received his BSc and MSc in software engineering from Islamic Azad University, Qazvin Branch. Currently, He is pursuing PhD in Software engineering at Islamic Azad University, Qazvin Branch, Iran. His research interests include software engineering, software testing, and software debugging.

Mohammadreza Keyvanpour is an Associate Professor at Alzahra University, Tehran, Iran. He received his BSc in software engineering from Iran University of Science & Technology, Tehran, Iran. He received his MSc and PhD in software engineering from Tarbiat Modares University, Tehran, Iran. His research interests include software engineering and data mining.

Saeed Parsa received his BSc in mathematics and computer science from Sharif University of Technology, Iran, and received his MSc and PhD in Computer Science from the University of Salford, England. He is an Associate Professor and head of software group in Iran University of Science and Technology. His research interests include software engineering, software testing, and software debugging.