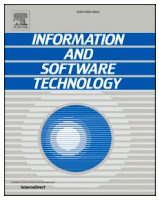
Information and Software Technology 177 (2025) 107564

Contents lists available at ScienceDirect

Information and Software Technology

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



Enhancing logic-based testing with EvoDomain: A search-based domain-oriented test suite generation approach

Akram Kalaee, Saeed Parsa *, Zahra Mansouri

School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

ARTICLE INFO

Keywords:
Software testing
Domain-oriented test suite generation
Logic-based testing
Mutation analysis
MC/DC

ABSTRACT

Context: Effective software testing requires test adequacy criteria. MC/DC, a widely used logic-based testing criterion, struggles to detect domain errors caused by incorrect arithmetic operations. Domain errors occur when test requirement boundaries shift or tilt, causing unpredictable behavior and system crashes.
Objective: To address the inadequacy of MC/DC in detecting domain errors, we present EvoDomain, a search-based testing technique.
Method: EvoDomain uses a memetic algorithm combining genetic and hill-climbing algorithms, along with the DBSCAN clustering algorithm to select diversified boundary test data. The memetic algorithm is designed to efficiently enhance the search process for covering boundary test data. We compared EvoDomain with two logic-based testing approaches, a domain-oriented test suite generation approach, and random testing.
Results: Evaluations on 30 case studies show EvoDomain increases fault detection by 74.44% over MC/DC and 65.06% over RoRG. Additionally, EvoDomain improves support for different fault types by up to 68.89% for MC/DC and 66.33% for RoRG. Compared to COSMOS, which uses static analysis, EvoDomain improves the convergence effectiveness of identifying feasible subdomains by 32%. It offers high accuracy (0.99-1) and F1-score (0.99-1). EvoDomain finds the subdomains in less than 1/3 the time of Random search.
Conclusion: EvoDomain effectively generates domain-oriented test suites, enhancing the accuracy and effectiveness of fault detection.

1. Introduction

A computer program can be seen as a function that maps the input space into a finite number of domains, assigning a separate path to each. Thus, the concepts of input domain and program path are essential for identifying and understanding errors. Program errors can be classified into two main groups: calculation and domain errors [1]. Calculation errors occur when the program executes the correct path, but the output is incorrect due to a computational mistake. Domain errors happen when the program executes the wrong path due to errors in conditional instructions or decision-making logic.

Amman and Offutt [2] propose a unique approach to effectively identify and rectify software program errors based on finding models to describe software and designing tests that cover them. They identify four key structures: graphs, logical expressions, input spaces, and syntax structures, each representing a category of coverage criteria. Multiple condition-decision coverage (MC/DC) is a logical-based testing criterion commonly used in safety-critical domains. It ensures that each condition in a decision independently affects the outcome of the decision. In a predicate that states $p = a \vee b \wedge c$, logic-based criteria test the individual clauses (a, b, c). However, logic-based criteria do not perform tests at the lower level of abstraction inside the clauses, such as $a = (m \leq n)$, $b = (d == e)$, and $c = (x > y)$. Integrating the power of the relational operator replacement (ROR), a mutation operator, into the logic-based criteria strengthens MC/DC to see whether the relations inside the clauses are formulated correctly [3]. This integration, called RoRG, is promising in finding more faults than MC/DC. However, it does not appropriately support domain errors, especially when there is an incorrect arithmetic operation. Consider the following faulty predicate: *if ((h + 100) > 0 & g)*, where the correct version is *if ((h - 100) > 0 & g)*. This error is not detectable until the value of *h* becomes less than or equal to -100. Such errors occur when the boundary of a test requirement, (e.g., a clause or predicate), is shifted or when the edge is tilted [1]. This can significantly impact the functionality of a software program, leading to unpredictable behavior, incorrect results, or even system crashes. Therefore, it's crucial to generate test data covering as many different scenarios as

* Corresponding author.
E-mail addresses: a_kalaee@comp.iust.ac.ir (A. Kalaee), prasa@iust.ac.ir (S. Parsa), zahra_mansouri@comp.iust.ac.ir (Z. Mansouri).

https://doi.org/10.1016/j.infsof.2024.107564
Received 31 October 2023; Received in revised form 21 August 2024; Accepted 22 August 2024
Available online 26 August 2024

possible, including edge cases and unexpected inputs, to ensure the robustness and reliability of a software system.

Various domain-oriented testing methods identify feasible subdomains—subsets of the input space where constraints hold—to uncover domain errors, each with unique advantages and disadvantages. Path-oriented Random Testing (PRT) [4] leverages constraint reasoning to generate random test data efficiently, significantly reducing rejected tests and enhancing CPU efficiency. Iterative Path-oriented Random Testing (IP-PRT) [5] refines this approach by using binary search to eliminate non-compliant subdomains, though it does not guarantee constraint solvability. Dynamic Domain Reduction (DDR) [6] dynamically adjusts input variable domains to minimize the search space, improving testing efficiency but facing challenges like looping and inefficient domain management. Rapid Dynamic Domain Reduction (RDDR) [7] enhances DDR by decomposing and refining subdomains, although it may create inconsistent subdomains and inadequate path coverage. COSMOS [8], a domain solver, surpasses conventional constraint solvers by identifying multiple inputs that fulfill constraints, effectively addressing DDR challenges and setting a new benchmark in constraint solving.

The current state of research in domain-oriented testing faces two limitations: (L1) it has not yet been applied to logic-based testing criteria, and (L2) it predominantly depends on static analysis. Static analysis examines the source code of a program without executing it. However, it struggles with intricate program structures and achieving full symbolic execution, which compromises its effectiveness [9]. To overcome these limitations, we present EvoDomain, a dynamic approach for generating domain-oriented test suites using a genetic algorithm. To enhance search space exploration and expedite the identification of boundary test data, we employ a memetic algorithm that combines the genetic algorithm with a local search technique, specifically a hill-climbing algorithm. After generating the test suite, we use DBSCAN, a robust and efficient clustering algorithm [10], to select a diverse set of boundary test data. The chosen test data is then used to augment an MC/DC-adequate test suite, enhancing the MC/DC test suite's ability to identify domain errors.

We evaluated our approach using 11 case studies from classic problems and 19 case studies from an industrial problem. The results are promising compared to the MC/DC and RoRG baselines. We achieved a 74.44% increase in fault detection rate compared to MC/DC and a 65.06% increase compared to RoRG across all studies. We also compared our approach to COSMOS and found a 32% improvement in convergence effectiveness. In the domain identification problem, our approach showed desirable accuracy (ranging from 0.99 to 1), F1-score (ranging from 0.99 to 1), precision (ranging from 0.34 to 0.97), and convergence (ranging from 0.03 to 0.66). EvoDomain consistently finds feasible subdomains in less than one-third of the time required by Random search. Moreover, our approach demonstrated significant practical benefits in improving the performance of spectrum-based fault localization techniques [11], which identify faulty statements by analyzing execution traces and computing the likelihood of faults based on passing and failing executions, compared to the baselines. Specifically, we observed up to a 53.16% improvement for Ochiai, 47.97% for Tarantula, and 51.68% for Jaccard. Additionally, EvoDomain enhances support for different fault types, with improvements of up to 68.89% for MC/DC and 66.33% for RoRG. In summary, the main contributions are as follows:

- Identifying feasible domains for specific constraints over a bounded input space, even when these domains are discontinuous, EvoDomain enhances domain error detection in software under test.
- By providing test data to cover individual clause domains within predicates, EvoDomain has expanded the MC/DC-adequate test suite to encompass various feasible domains, significantly enhancing its ability to detect domain-specific errors.
- Utilizing a memetic algorithm for the dynamic generation of domain-oriented test suites, EvoDomain has enhanced the efficiency and effectiveness of test suite generation, marking a significant advancement in the state-of-the-art of domain-oriented testing.
- Benchmarking domain-oriented test suite generation against established logic-based approaches, EvoDomain significantly enhances MC/DC criterion. It effectively identifies various fault types, addresses both computational and domain errors, and demonstrates practical benefits in fault localization.
- Introducing an openly available implementation of EvoDomain, accessible at https://github.com/IUST-EXPERT/EvoDomain, significantly facilitates easy adoption and encourages further research and development within the community, highlighting its innovative contribution to the field.

The remaining parts of this paper are as follows. Section 2 presents a motivating example illustrating the necessity of domain-based test suite generation. Section 3 offers an overview of relevant background information. In Section 4, we introduce our proposed approach in detail. Section 5 evaluates EvoDomain's effectiveness through some well-known case studies and benchmarks. Section 6 briefly reviews the related work. Finally, Section 7 concludes the paper and outlines future research.

2. Problem statement and motivation: an illustrative example

Example 1 highlights the limitation of MC/DC in detecting domain errors and underscores the need for a domain-oriented test suite to address this shortcoming.

Example 1: Fig. 1(a) illustrates a program under test that takes two variables and returns a boolean value. There is an error in the statement on Line 8. It should be $term2 = s * (a - t) * cos(x + 50)$ instead of $term2 = s * (a - t) * cos(x - 50)$. The input space is defined as $D = \{x \in (-13, 18), y \in (-9, 31)\}$. The test requirement is to achieve MC/DC coverage of the True branch for the predicate on Line 11, which consists of three clauses: C1: $(g <= 8)$, C2: $(x >= -9 \text{ and } x <= 14)$, and C3: $(y >= -7 \text{ and } y <= 17)$. In logic-based criteria, the internal structure of these clauses is ignored, treating the compound predicate as a simplified form (A, B, and C) without considering arithmetic operations. To achieve 100% MC/DC coverage, each condition must be varied independently while keeping the other conditions fixed. Fig. 1(b) presents four test data that cover all combinations. The symbol X indicates irrelevance. For instance, in test T_2, the status of $C3$ is irrelevant to the output since $C2$ is False. These tests cover all possible combinations of outcomes, considering the dependencies between conditions. However, this test suite cannot detect the fault in Line 8 which causes a domain error. Domain errors can be identified at boundaries or when a slight boundary change alters the behaviour, leading to a different program path. A domain-oriented test suite generation method can enhance the existing test suite by selecting at least one test data point on boundaries for each uncovered subdomain. For example, selecting $T_5 = (1.8, 1.93)$ and $T_6 = (-3.9, 16.71)$ for the running example, improves the chance of fault detection. To aid understanding, Fig. 1(c) shows the input space for the program under test. The regions within the blue rectangle indicate feasible subdomains for both the faulty and correct versions of the target predicate. Red regions denote the faulty version's feasible subdomains, while green areas represent the correct version. Cross points indicate the MC/DC test suite from Fig. 1(b), and dot points signify the added domain-oriented test data. The figure highlights three disconnected subdomains satisfying the predicate.

3. Background

This section offers a concise overview of the key concepts for comprehending our proposed approach.

```
  def Foo (x, y):
1. a = 1
2. b = 5.1 / (4 * pow(π, 2))
3. c = 5 / π
4. r = 6
5. s = 10
6. t = 1 / (8 * π)
7. term1 = pow((y - (b * pow(x, 2)) + c*x - r), 2)
8. term2 = s * (a - t) * cos(x-50) #Faulty statement
9. term3 = s
10. g = term1 + term2 + term3
11. if (g <= 8) and
        (x >= -9 and x <= 14) and
        (y >= -7 and y <= 17):
12.         return True   # Target branch point
13. else:
14.    return False
```

(a) The program under test

#	Conditions			Test data		Output
	C1	C2	C3	x	y	
T_1	F	X	X	5	12	F
T_2	T	F	X	15	10	F
T_3	T	T	F	-9	29	F
T_4	T	T	T	9	2	T

(b) A sample test suite achieving full MC/DC coverage

(c) The input space

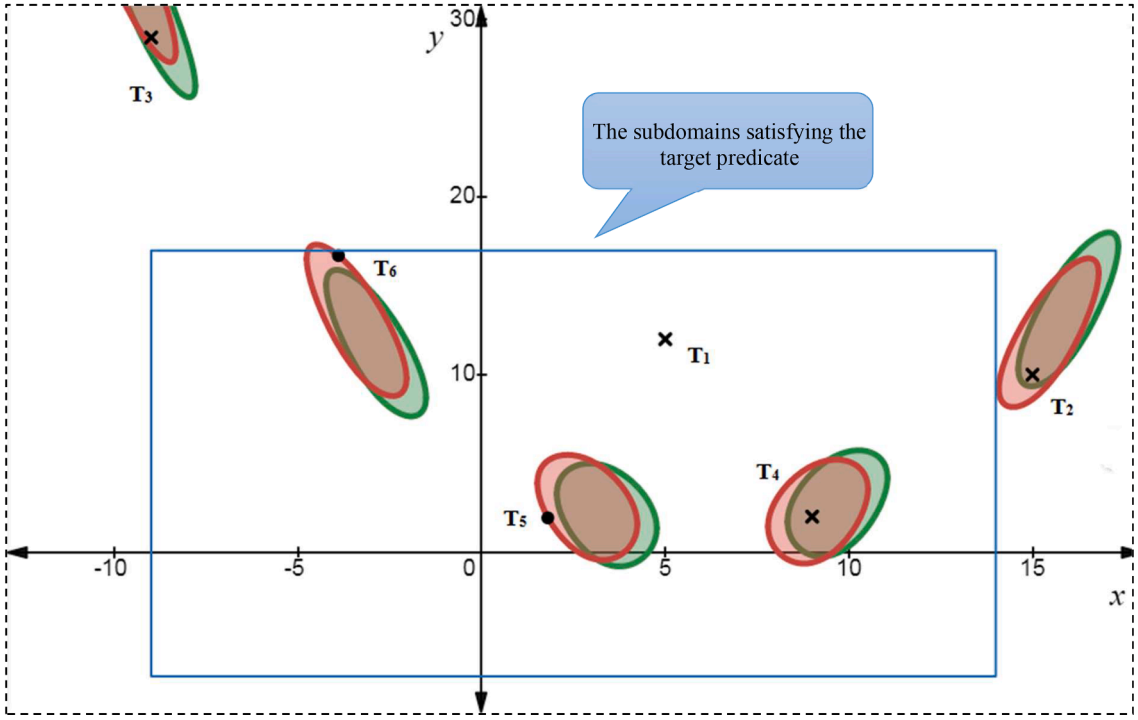


Fig. 1. The running example: MC/DC's limitation in domain error detection.

3.1. Search-based testing

The search-based software testing (SBST) transforms the test data generation problem into an optimization problem, utilizing a fitness function to guide the search [12]. Genetic algorithms [13] are used frequently as a meta-heuristic search technique for test generation. Candidate solutions are randomly selected and improved through evolutionary operators like mutation and crossover, enhancing fitness values. In the case of unit test generation, the objective function can be the code coverage of the entire test suite.

SBST uses a fitness function to guide test data generation. The fitness of test data for covering specific test path branches is typically evaluated based on branch distance (BD) and approach level (AL) metrics [14]. The branch distance metric quantifies how close a predicate of the program is to being satisfied or violated using specific test data. *BD (Pr)* represents the BD of the predicate *Pr*. For example, consider the predicate $x > 10$ and x having the value 5. The branch distance to the true branch can be calculated as $10 - 5 + K = 5 + K$, where K is a constant value. Different predicates have different methods for calculating BD, as outlined in Table 1. Suppose that the traversed path of input x is $P(x)$, and the target path is P. The BD between $P(x)$ and P is the sum of the BD of those branch predicates in both paths. Thus, it can be defined as Eq. 1.

Table 1
The fitness function of different predicates designed by Tracey et al. [14].

Predicate	BD Computation
Boolean	if true, then 0 else K
a > b	if b – a < 0, then 0 else (b – a) + K
a ≥ b	if b – a ≤ 0, then 0 else (b – a) + K
a < b	if a – b < 0, then 0 lese (a – b) + K
a ≤ b	if a – b ≤ 0, then 0 else (a – b) + K
a = b	if |a – b| = 0, then 0 else abs (a – b) + K
a ∧ b	BD (a) + BD (b)
a ∨ b	Min [BD (A), BD (b)]

$$\mathrm{BD}(\mathrm{P_x},\ \mathrm{P}) = \sum_{i=1}^{N} \mathrm{BD}(\mathrm{Pr_i}) \Bigg) \tag{1}$$

The approach level metric assesses how close the corresponding execution path of a test data is to reaching the predicates of the test path. This essentially measures how many more branches are left to execute to complete the test path. The fitness function (FF) is then defined as Eq. 2. Normalization that maps the BD value into the interval [0, 1) is calculated according to Eq. 3. This is done to ensure that the effect of the approach level (the number of branches left to execute the test path) is greater than the branch distance in the fitness function.

$$\mathrm{FF}\ (\mathrm{P_x},\ \mathrm{P}) = \mathrm{AL}\ (\mathrm{P_x},\ \mathrm{P}) + \mathrm{Normalization}\ (\mathrm{BD}\ (\mathrm{P_x},\ \mathrm{P})) \tag{2}$$

$$\mathrm{Normalization}\ (\mathrm{x}) = \mathrm{x}\ /\ (\mathrm{x} + 1) \tag{3}$$

When the fitness function *FF (x)* of *x* equals zero, it indicates that the input *x* has traversed the target path. Furthermore, the smaller the value of *FF (x)*, the closer the input *x* is to covering the target path. Consequently, the problem of generating test cases for covering the path is transformed into a problem of minimizing the fitness function *FF (x)*.

3.2. Search optimization algorithms

Search optimization algorithms are pivotal in computational intelligence, offering three primary methodologies: global, local, and hybrid search. Each serves distinct purposes based on the problem's complexity and requirements.

Global Search Algorithms: Global search algorithms, such as genetic algorithms (GAs), aim to explore the entire search space to identify the optimal solution. They employ techniques like mutation and crossover to maintain diversity and prevent premature convergence on suboptimal solutions. This approach is particularly beneficial for problems with numerous local optima or when comprehensive exploration is necessary [11].

Local Search Algorithms: In contrast, local search algorithms, like hill climbing, focus on optimizing solutions within a limited neighborhood around the current solution. They excel at refining solutions quickly but risk getting trapped in local optima due to their restricted scope of exploration. Local search is especially useful for fine-tuning solutions or when computational efficiency is paramount [15].

Hybrid Algorithms: Hybrid algorithms combine the strengths of both global and local search methods to enhance optimization performance. An example of this approach is memetic algorithms (MAs). MAs start with a global search phase using an evolutionary algorithm to explore the solution space widely. This is followed by a series of local searches to refine promising solutions. This dual-phase strategy aims to balance exploration and exploitation, efficiently navigating toward the global optimum by avoiding premature convergence while still benefiting from the rapid convergence of local search methods [16].

3.3. Mutation testing

Mutation testing is a technique used in software testing to evaluate the effectiveness of test cases [17]. It involves introducing artificial faults, or mutants, into the code by creating modified program versions. These mutants are created by applying syntax-modifying mutation operators to the original code. Test cases are then executed against these mutants to determine if they can be detected as faulty. Mutation testing aims to assess the test suite's quality by measuring its ability to detect these artificial faults. It has effectively identified weaknesses in test suites and improved the overall quality of software testing. Several categories of mutation operators exist for arithmetic and boolean operators, variable replacements, access modifier changes, type cast operator insertions/deletions, and class-level mutation operators for complex objects.

3.4. MC/DC criterion

MC/DC is a widespread test criterion in logic-based testing, particularly in safety-critical domains [3]. It is mandated by the US federal aviation administration (FAA-DO178B) [18]. MC/DC ensures that every condition in a decision is tested with all possible outcomes independently. It requires that each condition alone affects the decision outcome and individually influences the outcome when combined with all possible combinations of other conditions. MC/DC generates test sets that can identify faults related to conditions and decisions in logical expressions.

3.5. ROR

ROR is a mutation operator that generates mutants by replacing relational operators in logical expressions [19]. Traditionally, this operator produces seven mutants for each relational operator. However, it has been suggested that only three of these mutants are necessary for effective testing. A test set that is ROR-adequate can identify faults introduced by the ROR mutation operator. By incorporating ROR adequacy into logic-based test criteria, such as MC/DC, the effectiveness of the test set can be enhanced. The ROR-adequate test set comprises tests identifying and eliminating the ROR mutant when a relational operator is substituted with either true or false.

3.6. Fault localization

During software testing, failures are detected, and debugging is used to fix errors and improve program quality. Debugging involves identifying, fixing, and verifying faults, but this can be time-consuming and costly. Recently, automatic fault localization techniques like spectrum-based fault localization (SBFL) [11] have been developed to help developers identify fault locations and reduce debugging effort. SBFL analyzes coverage information and correlates program entities with failures, producing a ranking list of suspicious program entities. Among the widely recognized methods are Tarantula [20], Ochiai [21], and Jaccard [22], each using different statistical approaches to evaluate the suspiciousness of statements, as shown in Table 2.

4. Proposed approach

To generate a domain-oriented test suite, we start by identifying a feasible domain for the given test requirement, such as a predicate. The test data are then diversely selected from the boundary regions of this domain, where input values are most likely to affect the program's output or behaviour. Identifying these feasible domains can be challenging, especially for discontinuous domains due to the irregular and fragmented nature of these regions [23]. Therefore, we formulate this as a search-based testing problem. We introduce a new fitness function to guide the search in sampling boundary feasible points (Section 4.4.3). To further improve the search process, we integrate a global search (Section 4.4) with a local search (Section 4.5), specifically using the

Table 2
Some suspiciousness measures are used in SBFL.

Suspiciousness Measure	Algebraic Formula
Tarantula	$\frac{A_f^c}{A_f^c + A_f^n}$ / $\frac{A_f^c}{A_f^c + A_f^n} + \frac{A_p^c}{A_p^c + A_p^n}$
Ochiai	$\frac{A_f^c}{\sqrt{\left(A_f^c + A_f^n\right) * \left(A_f^c + A_p^c\right)}}$
Jaccard	$\frac{A_f^c}{A_f^c + A_f^n + A_p^c}$

Memetic Algorithm (Section 4.8). Once the feasible domain has been identified, it is used to guide the selection of a diverse test suite (Section 4.7). The Memetic Algorithm was chosen for its ability to balance extensive exploration and precise refinement, as detailed in Section 3. Empirical evidence supports its effectiveness across various domains, highlighting its flexibility and reliability [16,24]. While modern genetic algorithms like P3 [25] and NSGA-II [26] have their strengths, they do not suit our specific needs. P3 excels in certain optimizations but lacks the balance we require, and NSGA-II is better for multi-objective tasks [27], whereas our focus is on single-objective optimization. The hybrid approach of the Memetic Algorithm ensures comprehensive search coverage and fine-tuning of solutions.

4.1. EvoDomain overview

EvoDomain is a tool designed to assist test engineers in creating domain-oriented test suites that enhance structural coverage criteria, such as MC/DC, and provide insights into the decision-making process of the program under test (PUT). Fig. 2 illustrates an overview of EvoDomain. The process begins with the test engineer supplying a configuration file containing details about the PUT, test requirement, initial domain of variables, acceptable boundary thickness, the test suite size, parameters of the underlying algorithms, and the termination condition. For test requirements, the user can utilize EvoDomain's path extractor component to list the prime paths of the PUT and select one as the test requirement. Prime paths are those that are simple (do not repeat any nodes except possibly the first and last) and are not a subpath of any other simple path [2]. Prime path coverage is the practicable alternative to complete path coverage [2].

Before generating test data, the PUT is instrumented to track the path covered by test data which is necessary to calculate its fitness, detailed in Section 4.3. The test data generation module is configured with information such as the input variables types and initial input variables domains. The module employs a memetic algorithm, which combines global and local search techniques, to efficiently explore and exploit the search space. The memetic algorithm iteratively generates new test data based on previous results, aiming to identify feasible boundary test data. The EvoDomain framework manages the execution of these test data and the scoring of results. Once the termination criteria are met, the test data selection component clusters the feasible test dataset to identify subdomains. Test data is ranked for each subdomain based on its proximity to the boundaries. The final test suite consists of diverse boundary points from each specific subdomain, creating a domain-oriented test suite. This suite can be integrated with another test suite targeting structural test requirements to enhance behavioral coverage. It is also worth mentioning that EvoDomain supports whole program testing based on the prime path coverage concept as defined in traditional software testing [2]. To achieve this, it is only required to call EvoDomain for each specific prime path.

4.2. Running example

In the following sections, we will demonstrate how EvoDomain operates step by step using the program presented in Example 1 from Section 2. The program contains two prime paths: [1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 12] and [1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9 → 10 → 11 → 13 → 14], with each number representing a line in the program. The first path follows the True branch of the target predicate, while the second path follows the False branch. For this example, the first path is considered the test requirement.

4.3. Instrumentation

Instrumentation enhances a program under test by embedding additional code to monitor the behaviour and collect data during runtime. This approach is essential for assessing test data's fitness, particularly through observing function calls that track execution paths and predicate evaluations. To apply instrumentation, we adjust the predicates within the code to incorporate a call to a function that calculates the branch distance. An illustrative example involves the evaluate_branch_distance function, which accepts a predicate ID and a vector detailing the clause structure. For instance, for clause $a > b$, the vector might be *[1, 'Gt', a, b]*, where *1* represents the clause ID, *'Gt'* signifies the greater-than operation, and *a* and *b* are the operands. Listing 1 shows the instrumented version of the running example.

This example utilizes evaluate_branch_distance to calculate branch distances (True_Distance and False_Distance) based on the clause provided. By doing so, instrumentation facilitates the monitoring of condition evaluations without altering the program's logic. The function evaluate_branch_distance processes a predicate ID and a pre-order traversal, with each clause in the sequence represented by a vector containing the clause ID, operator, left operand, and right operand. The predicate's truth value depends on the traversal assessment. Additionally, branch distances quantify how near a predicate is to being true (True_Distance) or false (False_Distance), offering a numerical representation of proximity to these states, as detailed in Section 4.4.3.

4.4. Global search

This section describes an algorithm using a GA to find feasible sub-

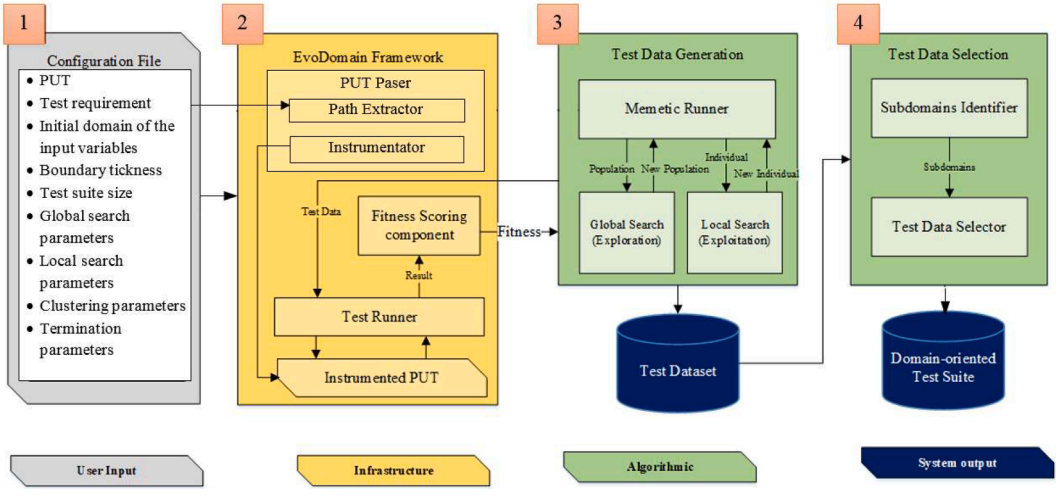


Fig. 2. The EvoDomain overview.

```
def Foo_instrumented (x, y):
# Lines 1 to 10 are the same as the original function
11. if evaluate_branch_distance (1, ['And',[1, 'Le', g, 8],
                                    'And',[2, 'Ge', x, -9],[3, 'Le', x, 14],
                                    'And',[4, 'Ge', y, -7],[5, 'Le', y, 17]]):
12.      return True # Target branch point
13. else:
14.      return False
```

Listing 1. Instrumented function for the example shown in Fig. 1.

domains within test requirements. It explains individual representation, population initialization, and a fitness function that guides the population towards boundary regions. Such regions are prone to faults [28-30]. The GA uses search operators to evolve the population.

4.4.1. Representation

In EvoDomain, each individual, representing a potential solution to the problem, is described as a string of numbers called genes. These genes are analogous to the input variables. Each set of input variables is considered an individual in the GA. The collection of these individuals forms a population. Each individual is a vector of input values, $V = < v_1, v_2, \ldots, v_n >$, to SUT. For our running example, the individual is a vector of size 2, as it receives x and y as inputs.

4.4.2. Initial population

In the first iteration, the search begins with a randomly generated population of individuals. This straightforward approach ensures diversity in the initial population, which is crucial for avoiding premature convergence to suboptimal solutions. As the algorithm progresses, new test data is generated, and the population is refined through genetic operations such as mutation and crossover (Section 4.4.4). For the running example, the input space (i.e., the initial domain) is defined as $D = \{x \in (-13, 18), y \in (-9, 31)\}$. Therefore, the individuals in the initial population are randomly sampled from this domain, ensuring a wide range of input values to explore different potential solutions effectively.

4.5. Fitness function

In Section 3.1, we highlighted a key issue with the conventional fitness function in SBST: when multiple test cases cover a test path equally well, they all receive a fitness score of zero. This makes it difficult to differentiate their effectiveness, especially in terms of their proximity to the target path's boundary within a control flow graph, revealing the limitations of this approach in domain-oriented test suite generation. To address this issue, we introduce a novel approach that enhances the evaluation of test cases by incorporating a signed version of the branch distance (SBD) and an extended version of the branch distance (EBD). These metrics serve as the foundation for calculating an extended fitness function (EFF), which aims to provide a more nuanced assessment of test case quality. By doing so, our proposed method not only identifies test cases that closely approximate the target path but also distinguishes between those that do so effectively and those that merely achieve a superficial coverage. This advancement significantly improves the ability to generate effective domain-oriented test suites by prioritizing test cases based on their actual contribution to uncovering potential defects along the target path. The EFF is calculated by adding the approach level, AL, and a normalization of the EBD. The normalized value is calculated according to a formula given in Eq.4.

$$\mathrm{EFF}\ (P_x, P) = \mathrm{AL}\ (P_x, P) + \mathrm{Normalization}\ (\mathrm{EBD}\ (P_x, P)) \tag{4}$$

AL measures how close a specific test data reaches a target in the control flow graph. On the other hand, the branch distance, BD, measures how close a specific test data is to making a different decision at a particular point in the control flow graph. The extension of BD to EBD and SBD allows for a more nuanced measure of the proximity of test data to path boundaries. The EBD is calculated based on the predicates and their desired branches. Here, different rules are applied for other predicates (like $a = b$, $a \neq b$, $a < b$, etc.), as mentioned in Table 3.

According to Eq. 5, test data to cover the domain boundaries of a test path, P, must satisfy two requirements:

1. Achieving zero value for AL (P_x, P): This requirement implies that the test data, x, should ideally align with the path P. The AL measures how well a given test data path, P_x, aligns with the target path, P. A zero value indicates a perfect alignment, meaning the test data follows the same path as the target path. This ensures that the test data covers the exact path we are interested in testing.
2. Having at least one *SBD (Pr_i)* value within the range [-δ, 0], called interior boundary: This requirement ensures that the test data covers the boundary of the path. The SBD measures how close the test data is to the boundary of the path. Eq. 6 calculates SBD, essentially a signed version of BD: if the desired branch of a predicate is covered, the SBD is negative; otherwise, it is positive. The SBD value within the range [-δ, 0] indicates that the test data is within the boundary's desired thickness (δ). This allows us to test the system's behaviour at boundary conditions, where errors are most likely to occur.

$$\mathrm{EBD}(P_x, P) = \begin{cases} 0 & \text{if } \max\limits_{1 \le i \le N} \mathrm{SBD}(Pr_i) \in [-\delta, 0] \\ \left| \max\limits_{1 \le i \le N} \mathrm{SBD}(Pr_i) \right| & \text{otherwise} \end{cases} \tag{5}$$

$$\mathrm{SBD}(Pr_i) = \mathrm{EBD}\ (Pr_i) \times \begin{cases} -1 & \text{If the desired branch of } Pr_i \text{ is covered} \\ 1 & \text{otherwise} \end{cases} \tag{6}$$

Table 4 illustrates the step-by-step calculation of the EFF for three test data points (T_7, T_8, and T_9) in an example where the threshold δ is set at 0.1. As detailed in Listing 1, the target path includes the true branches of predicates 1 through 5. To demonstrate the advantages of using EFF, we present two examples derived from Table 4.

Table 3
The branch distance for different predicates where K is a constant value.

Predicate	EBD computation	
	If the true branch is covered	If the false branch is covered
Boolean	K	K
a = b	K	|a – b|
a ≠ b	|a – b|	K
a < b	(b – a)	(a - b) + K
a ≤ b	(b - a) + K	(a - b)
a > b	(a - b)	(b - a) + K
a ≥ b	(a - b) + K	(b - a)
a ∨ b	Min (SBD (a), SBD (b))	
a ∧ b	Max (SBD (a), SBD (b))	
¬ a	Negation propagated over a	

Table 4
Extended Fitness Function Calculation for Three Test Data Points in the Running Example.

Steps	$T_7 = (3, 2.5)$	$T_8 = (1.7, 4.96)$	$T_9 = (1.6, 2.57)$
EBD (Pr_5)	(17 - 2.5) + 0.1 = 14.6000	(17 - 4.96) + 0.1 = 12.1400	(17 - 2.57) + 0.1 = 14.5300
SBD (Pr_5)	-14.6000	-12.1400	-14.5300
EBD (Pr_4)	(2.5 - (-7)) + 0.1 = 9.6000	(4.96 - (-7)) + 0.1 = 12.0600	(2.57 - (-7)) + 0.1 = 9.6700
SBD (Pr_4)	-9.6000	-12.0600	-9.6700
$\max_{i\in\{4,\ 5\}} SBD(Pr_i)$	Max(-9.6000,-14.6000) = -9.6000	Max(-12.0600,-12.1400) = -12.0600	Max(-9.6700,-14.5300) = -9.6700
EBD (Pr_3)	(14 - 3) + 0.1 = 11.1000	(14 - 1.7) + 0.1 = 12.4000	(14 - 1.6) + 0.1 = 12.5000
SBD (Pr_3)	-11.1000	-12.4000	-12.5000
EBD (Pr_2)	(3 - (-9)) + 0.1 = 12.1000	(1.7 - (-9)) + 0.1 = 10.8000	(1.6 - (-9)) + 0.1 = 10.7000
SBD (Pr_2)	-12.1000	-10.8000	-10.7000
$\max_{i\in\{2,\ 3\}} SBD(Pr_i)$	Max(-12.1000, -11.1000) = -11.1000	Max(-10.8000, -12.4000) = -10.8000	Max(-10.7000, -12.5000) = -10.7000
$\max_{i\in\{(2,\ 3),\ (4,\ 5)\}} SBD(Pr_i)$	Max(-11.1000,-9.6000) = -9.6000	Max(-10.8000,-12.0600) = -10.8000	Max(-10.7000,- 9.6700) = -9.6700
EBD (Pr_1)	(0.48 - 8) + 0.1 = 7.6200	(7.98 - 8) + 0.1 = 0.0800	8.69 - 8 = 0.6900
SBD (Pr_1)	-7.6200	-0.0800	0.6900
$\max_{i\in\{1,\ (2,\ 3,\ 4,\ 5)\}} SBD(Pr_i)$	Max(-7.6200,-9.6000)= -7.6200	Max(-0.0800,-10.8000)= -0.0800	Max(0.6900,-9.6700) = 0.6900
$EBD(P_{T_j}, P), j \in \{1, 2, 3\}$	|-7.6200| = 7.6200	0	|0.6900| = 0.6900
$AL(P_{T_j}, P), j \in \{1, 2, 3\}$	0	0	0
$EFF(P_{T_j}, P), j \in \{1, 2, 3\}$	0 + 7.6200 / (7.6200+1) = 0.8840	0 + 0/ (0+1) = 0	0 + 0.6900 / (0.6900+1) = 0.4083

Example 2: Both test data T_7 and T_8 traverse the target path, resulting in traditional fitness scores of zero. However, the EFF differentiates between these two test cases, as shown in Table 4. This highlights the ability of the extended branch distance and the EFF to enhance differentiation among test cases, particularly in terms of path boundary coverage.

Example 3: Test data T_9 does not cover the target path. According to the traditional fitness function, T_7 is superior to T_9. However, T_9 has a lower EFF than T_7, indicating it is closer to the boundary of the target path. This differentiation is not possible with the traditional fitness function. This example demonstrates how the extended branch distance and the EFF provide refined distinctions among test data, especially in complex scenarios with multiple decision points. Additionally, it reveals an interesting aspect: infeasible test data like T_9 might receive better (i.e., lower) fitness values than feasible test data such as T_7. This approach helps prioritize test data in search-based test data generation techniques, favoring infeasible test data near the boundary over distant feasible ones. Proximity to the boundary increases the likelihood of identifying potential coding issues. Thus, the extended branch distance and the EFF offer a sophisticated and effective means for assessing test data regarding path boundary coverage. The EFF prioritizes paths closer to the boundary, regardless of feasibility, rather than merely preferring feasible paths over infeasible ones.

4.5.1. Search operators

EvoDomain uses deterministic elitist selection, single-point crossover, and mutation, similar to traditional GAs. Elitist selection picks the best individuals as parents for the next generation, ensuring superior genes are carried forward and increasing the chances of finding the optimal solution [31]. This strategy enhances the performance of Roulette wheel selection methods by maintaining superior solutions, making GAs more robust and efficient for domain-oriented test suite generation. It balances exploiting the best current solutions with exploring new possibilities through crossover and mutation, increasing the likelihood of converging on an optimal solution.

Single-point crossover swaps segments of parent chromosomes at a random point to produce offspring, with the crossover rate determining the difference between parents and offspring. After crossover, mutation is applied with a specific probability to introduce randomness and prevent the algorithm from getting stuck in local optima.

4.6. Local search

The global search provides an overview of the search space and identifies promising regions. However, it may not precisely locate optimal solutions within these regions. The local search fine-tunes solutions within these promising regions to pinpoint optimal solutions. EvoDomain must balance the timing of switching from global to local search to avoid missing potential solutions or wasting time. The Hill Climbing algorithm [15], due to its simplicity and efficiency, is used for this balance. It starts with a candidate solution from the global search and iteratively samples neighboring candidates, evaluating them with a fitness function. If a better candidate is found, it becomes the new solution. This process continues until a satisfactory solution is found or the search budget is exhausted.

4.7. Termination condition

Our approach refines results until satisfactory precision, monitored by changes in the F1 score over time and a predefined iteration limit. We evaluate the F1 score, a key metric for classification performance, every N iteration using a Random Forest classifier [32], known for its accuracy and ability to handle imbalanced datasets, on the collected test data. Cross-validation with k-fold (10 splits) ensures reproducibility [33]. If the mean F1 score shows slight improvement or the iteration limit is reached, the algorithm stops. Future studies may explore oversampling and Synthetic Minority Over-Sampling techniques.

4.8. Test data selection

To ensure diverse test data selection, it's crucial to identify discontinuous subdomains within the feasible data, as continuous regions exhibit the same behavior [2]. Using the DBSCAN algorithm [10], which clusters densely packed data points without needing a pre-specified number of clusters, helps identify distinct behavioral boundaries. This is particularly effective for detecting errors in safety-critical systems. Each cluster represents a separate behavioral boundary, and sampling from each ensures comprehensive domain coverage. Combining this with traditional coverage criteria like MC/DC achieves thorough test coverage, leveraging both domain-oriented and structural test data generation strengths.

```
Input:  input_variables_initial_domain, population_size, convergence_rate,
        number_of_generations, local_search_budget, mutation_rate, crossover_rate,
        stagnation_limit, generation_step, test_suite_size
Output: test_suite

Begin
    // Step 1: Initialization
1.  generation ← 0
2.  F1_score_per_generation ← ∅
3.  data_set ← ∅
4.  population ← create_initial_population(input_variables_initial_domain, population_size)
5.  population ← evaluate_population(population) // According to Eq. 4
6.  data_set ← data_set ∪ labeling(population)
7.  While (generation < number_of_generations)
        // Step 2: Exploration
8.      parents ← select_parents(population)
9.      new_population ← ∅
10.     For each adjacent pair of elements in parents
11.         offspring ← generate_offspring(parentes, crossover_rate, mutation_rate)
12.         new_population ← new_population ∪ offspring
13.     End For
        // Step 3: Exploitation
14.     If (generation % local_search_rate == 0)
15.         sort_population_by_fitness_descending(population)
16.         selected_individuals ← select_top_individuals(population, local_search_budget)
17.         For each individual in selected_individuals
18.             refined_individual ← perform_local_search(individual)
19.             new_population ← new_population ∪ {refined_individual}
20.         End For
        // Step 4: Fitness Evaluation
21.     population ← new_population
22.     population ← evaluate_population(population) // According to Eq. 4
23.     generation ← generation + 1
        // Step 5: Update the dataset
24.     data_set ← data_set ∪ labeling(population)
        // Step 6: Check termination conditions
25.     If (generation % generation_step == 0) and (has_zero_fitness(population))
26.         F1_score ← calculate_F1_score(population)
27.         F1_score_per_generation.append(F1_score)
28.     If length(F1_score_per_generation) ≥ stagnation_limit and
29.        variation_in_F1_scores(F1_score_per_generation) < convergence_rate
30.          break
31.     End If
32. End While
    // Step 7: Clustering
33. subdomains ← apply_DBSCAN_clustering(data_set)
    // Step 8: Test data selection
34. test_suite ← select_test_data(subdomains, test_suite_size)
35. return test_suite
End
```

Listing 2. The proposed algorithm for EvoDomain.

4.9. Algorithm

Listing 2 presents EvoDomain, a memetic algorithm combining genetic and hill-climbing methods. It balances exploration and exploitation, refines solutions, and maintains diversity. Periodic updates and clustering ensure a comprehensive test suite, while termination conditions ensure efficient convergence. Here are the detailed algorithm steps:

1. **Initialization:** The algorithm starts by initializing key variables: generation is set to 0, F1_score_per_generation is an empty list to track performance, and data_set is an empty set to store labeled data (lines 1-3). The initial population is created based on the input variables' domain and the specified population size (line 4). This population is then evaluated for fitness (line 5). Labeled data is added to the dataset (line 6). If the fitness value is less than or equal to zero, the label is 1 (feasible test data); otherwise, it is 0. The labeled test data gradually form a dataset.
2. **Exploration:** In the exploration phase, the algorithm selects parents from the current population to generate new offspring through crossover and mutation (lines 8-13), as detailed in Section 4.4.4. This phase introduces diversity into the new population, preventing premature convergence on suboptimal solutions.
3. **Exploitation:** Every few generations, determined by local_search_rate, the algorithm focuses on exploitation. It sorts the population by fitness and selects the top individuals for local search

(lines 15 and 16). This local search refines these individuals and adds them to the new population (lines 17-20). This step ensures continuous improvement of the best solutions.
4. **Fitness Evaluation:** The new population replaces the previous one and is evaluated for fitness (lines 21 and 22).
5. **Update the Dataset:** After each generation, the dataset is updated with the newly labeled population (line 24). This continuous update ensures that the dataset grows and evolves alongside the population, capturing a wide range of scenarios.
6. **Check Termination Conditions:** The algorithm periodically assesses termination criteria, specifically checking the F1 score every few generations to identify if any individual has zero fitness (lines 25-27). If the F1 scores exhibit minimal change across several generations, suggesting stagnation, the algorithm halts (lines 28-31). This mechanism prevents endless execution and guarantees convergence towards a solution. Utilizing the F1 score is advantageous due to its balance between precision and recall, essential in class-imbalanced scenarios. Precision gauges the correctness of positive predictions, whereas recall evaluates the effectiveness of detecting all positives. The F1 score, calculated as the harmonic mean of precision and recall, ensures a holistic assessment of the algorithm's performance, emphasizing accurate and consistent identification of viable test data.
7. **Clustering:** Once the evolutionary process is complete, the dataset undergoes clustering using the DBSCAN algorithm (line 33). This step identifies distinct subdomains within the dataset, which is crucial for understanding the different areas covered by the test data.
8. **Test Data Selection:** Finally, the algorithm selects the test suite from these clustered subdomains, focusing on boundary test data (line 34). This ensures that the test suite is comprehensive, covering a wide range of scenarios, including edge cases that are critical for robust testing.

5. Evaluation

This section provides an overview of our experiments, including the research questions, selected baselines, subject predicates, conducted experiments, and obtained results. The main goal was to evaluate EvoDomain's capabilities and limitations.

5.1. Research questions

This section aims to evaluate the efficacy of EvoDomain in comparison to other logic-based testing tools, particularly in identifying faults. The evaluation will address the following research questions:

RQ1 (Fault Detection): Can EvoDomain enhance the effectiveness of logic-based testing in detecting faults? Traditional methods like MC/DC and RoRG, despite their widespread use, fall short in detecting domain errors. To address this, we employ mutation testing to explore how EvoDomain can enhance the fault detection capabilities of these logic-based approaches. We use MutMut [34], a mutation testing tool for Python, which creates mutants using six fault categories:

1. relational operator replacement (ROR): Replaces a relational operator (e.g., <, >, ==, !=, <=, >=) with another, potentially triggering computation errors by altering the program's logic.
2. conditional statement modification (CSM): Modifies the condition in a conditional statement (e.g., if, while, for), changing the program's flow and potentially causing computation errors.
3. return value modification (RVM): Alters the return value of a function or method, which can lead to computation errors by changing the function's output or approach.
4. arithmetic operator replacement (AOR): Replaces an arithmetic operator (e.g., +, -, *, /) with another, potentially causing computation errors by altering calculations.
5. logical operator replacement (LOR): Replaces a logical operator (e.g., &&, ||, !) with another, which can trigger computation errors by changing the program's logic.
6. statement deletion (SD): Deletes a statement in the program, potentially causing computation errors by removing part of the program's logic.

- **RQ2 (Fault Types): How effective is EvoDomain in identifying various fault types?** This study aims to determine the most effective method for detecting faults across different subjects and types, including ROR, CSM, RVM, AOR, LOR, and SD.
- **RQ3 (Practical Advantage): What practical advantages does EvoDomain offer?** This question explores how EvoDomain's domain-oriented test suite impacts SBFL compared to baseline methods. Many SBFL approaches do not differentiate individual test case contributions, potentially limiting effectiveness. This evaluation will reveal how EvoDomain's enhanced test data generation improves fault localization over other logic-based testing tools.
- **RQ4 (Performance): How well does EvoDomain perform in detecting subdomains and boundary regions?** To rigorously assess EvoDomain's capability in detecting subdomains and boundary regions, we use performance metrics such as F1 score, accuracy, precision, and convergence rate as well as run time.
- **RQ5 (Comparison with COSMOS): How does EvoDomain compare with COSMOS?** This question examines the accuracy of domain detection by comparing EvoDomain with COSMOS, the leading domain-oriented test suite generation approach.
- **RQ6 (Search Effectiveness): Is feasible domain identification a non-trivial problem?** We compare our memetic algorithm with Random Search in terms of precision, convergence, and other metrics to demonstrate that sub-domain identification is a non-trivial task.

5.2. Subject predicates

Table 5 evaluates the EvoDomain approach using various predicates across classic and industrial problems [35,36]. Each predicate is characterized by metrics such as a unique identifier, the number of literals (variables), their value ranges, clause complexity (CC) [37], and the number of disconnected feasible sub-domains identified through a random search with a 7-hour budget. Analyzing these predicates helps us understand the EvoDomain method's scalability and performance. Higher CC and fewer feasible domains indicate more complex problems. The inclusion of traffic collision avoidance system (TCAS) predicates demonstrates the approach's practical applicability. The diversity in literal domains and numbers of literals highlights the method's versatility and robustness.

6. Baseline approaches

This section introduces the baseline approaches for domain-oriented and logic-based test suite generation. To evaluate the practical effectiveness of our approach, it also includes baselines from traditional SBFL methods. Additionally, it outlines the configuration setup of EvoDomain used in the experiments.

Domain-oriented Test Suite Generation. We use COSMOS, a highly effective approach from existing literature, as our baseline. COSMOS employs a partition method to assess partition coherence with the underlying constraint. For our analysis, we set the number of partitions, K, to 3, while keeping other parameters at their default settings.

Logic-Based Test Suite Generation. We compare EvoDomain to two well-known logic-based test suite generation approaches: MC/DC [38] and its improved version, RoRG [3].

Spectrum-Based Fault Localization. We evaluate the effectiveness of our approach using three well-known SBFL techniques— Tarantula

Table 5
The subject predicates. CC: clause complexity.

Predicate NO.	#Literals	Literal Initial Domain	CC	#Feasible Sub-Domains	Problem
1	2	(0,50), (0,30)	3.599	4	Classic Optimization
2	2	(-2,2), (-3,3)	2.1	2	
3	2	(-1,5), (-1,1)	1.2	3	
4	2	(-5,5), (-5,5)	1.7999	3	
5	2	(-10,10), (-10,10)	3	5	
6	2	(-5,5), (-5,5)	4.8	8	
7	2	(-8,8), (-3,7)	3.6	1	
8	2	(-10,10), (-10,10)	2.4	2	
9	2	(-5,5), (-5,5)	4.2	1	
10	3	(0,20), (0,40), (0, 50)	1.2	3	
11	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	Traffic Collision Avoidance System
12	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	6	
13	2	(0,2000), (0,2000)	1.8	2	
14	2	(0,2000), (0,2000)	0.6	3	
15	2	(0,2000), (0,2000)	0.6	4	
16	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	
17	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	
18	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	6	
19	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	
20	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	
21	4	(0,2000), (0,2000), (0,2000), (0,2000)	3	5	
22	3	(0,2000), (0,2000), (0,2000)	3	5	
23	3	(0,2000), (0,2000), (0,2000)	3	5	
24	3	(0,2000), (0,2000), (0,2000)	3	5	
25	3	(0,2000), (0,2000), (0,2000)	3	5	
26	3	(0,2000), (0,2000), (0,2000)	3	6	
27	3	(0,2000), (0,2000), (0,2000)	3	6	
28	3	(0,2000), (0,2000), (0,2000)	3	5	
29	3	(0,2000), (0,2000), (0,2000)	3	5	
30	2	(0,0.1), (0.02, 0.16)	8.4	1	

[20], Ochiai [21], and Jaccard [22]—to determine the suspiciousness score of each statement.

Setup. Based on established guidelines [13,31, 39-41], we determined the following parameters for our genetic algorithm: a crossover rate of 0.9 to ensure sufficient exploration of the search space, supported by studies like those by Katoch et al. [31]; a mutation rate calculated by dividing 1 by the number of variables encoded as a chromosome, which maintains diversity and avoids premature convergence [39]; and a population size of 100, balancing computational efficiency and solution quality [40]. The hill climbing algorithm's local search rate is set at 5, with a local search budget of 20, enhancing the genetic algorithm's performance [41]. We use the DBSCAN clustering algorithm with basic configurations to determine subdomains, balancing accuracy and computational cost [42]. The random forest algorithm is configured with a max_depth parameter set at 0.1 of the total generated test data and 10 trees, yielding robust models with good generalization capabilities [43]. Other parameters, set up empirically, include a convergence rate of 0.01, 1000 generations, a stagnation limit of 3, a generation step of 100, the number of boundary test data set to the number of detected subdomains for each subject, and input variables' initial domain set according to the initial domains listed in Table 5.

6.1. RQ1: Fault detection

Mutation testing assesses the efficacy of a test data generation approach, with higher mutation scores indicating better fault detection. Using MutMut tools, we compared EvoDomain's fault-finding capabilities with MC/DC and RoRG. Table 6 shows that EvoDomain significantly improves upon MC/DC (36.22% to 79.44%) and outperforms RoRG (1.43% to 65.06%). EvoDomain surpasses MC/DC in all cases and RoRG in 25 out of 30 cases, with minimal deficiencies in the worst cases (-4.29% to -8.57%). A test suite with full MC/DC coverage does not guarantee success. Domain-oriented test suites enhance reliability by covering a wide range of behaviors. RoRG performs better than MC/DC by considering relational operators but fails to cover all feasible subdomains due to neglecting other operators, resulting in lower mutation scores than EvoDomain.

We used the Wilcoxon rank-sum test [44] and Vargha and Delaney's $\widehat{A}_{12}$ statistic [45] to compare EvoDomain's fault detection power with the baselines. These non-parametric tests were chosen to avoid assumptions about normality and equal variances. The Wilcoxon test identifies significant differences, while $\widehat{A}_{12}$ measures effect size. Table 6 shows that EvoDomain had a statistically significant difference (p-value < 0.05) compared to MC/DC for all subjects. Compared to RoRG, EvoDomain performed significantly better for 19 out of 30 subjects, with no significant difference for the remaining 11 subjects.

The predicates being tested have multiple feasible subdomains (1 to

Table 6
Fault detection of the selected approaches by predicates. The average results are reported. EvoDomain outperforms MC/DC in all cases and RoRG in most cases.

Predicate	Total Faults	Faults Found (%)			Improvement (%)		EvoDomain vs. MC/DC		EvoDomain vs. RoRG	
		MC/DC	RoRG	EvoDomain	MC/DC	RoRG	$\hat{A}_{12}$	P-value	$\hat{A}_{12}$	P-value
1	19	16.84	23.16	**74.21**	57.37	51.05	**1**	**< 0.001**	**1**	**< 0.001**
2	24	42.08	42.08	**90.84**	48.75	48.75	**1**	**< 0.001**	**1**	**< 0.001**
3	35	39.14	39.14	**80.57**	41.43	41.43	**1**	**< 0.001**	**1**	**< 0.001**
4	15	26.67	50.67	**77.34**	50.67	26.67	**1**	**< 0.001**	**1**	**< 0.001**
5	22	43.75	43.75	**85.72**	41.97	41.97	**1**	**< 0.001**	**1**	**< 0.001**
6	33	27.27	39.09	**89.10**	61.83	50.01	**1**	**< 0.001**	**1**	**< 0.001**
7	31	48.13	48.13	**87.74**	39.62	39.62	**1**	**< 0.001**	**1**	**< 0.001**
8	18	44.44	44.44	**80.66**	36.22	36.22	**1**	**< 0.001**	**1**	**< 0.001**
9	30	29.44	33.33	**88.00**	58.56	54.67	**1**	**< 0.001**	**1**	**< 0.001**
10	22	22.73	46.36	**83.63**	60.90	37.27	**1**	**< 0.001**	**1**	**< 0.001**
11	7	42.86	**100.00**	91.43	48.57	-8.57	**1**	**< 0.001**	0.3	0.987
12	7	28.57	**90.00**	**90.00**	61.43	0.00	**1**	**< 0.001**	0.43	0.743
13	6	33.33	66.67	**95.00**	61.67	28.33	**1**	**< 0.001**	**0.95**	**< 0.001**
14	2	50.00	100.00	**100.00**	50.00	0.00	**1**	**< 0.001**	**1**	**0.5**
15	4	25.00	75.00	**100.00**	75.00	25.00	**1**	**< 0.001**	**1**	**< 0.001**
16	7	28.57	82.86	**95.71**	67.14	12.86	**1**	**< 0.001**	0.61	0.175
17	7	28.57	91.43	**92.86**	64.29	1.43	**1**	**< 0.001**	0.47	0.602
18	7	28.57	91.43	**94.28**	65.71	2.86	**1**	**< 0.001**	0.51	0.482
19	7	28.57	95.71	**97.14**	68.57	1.43	**1**	**< 0.001**	0.51	0.478
20	7	42.86	**100.00**	95.71	52.85	-4.29	**1**	**< 0.001**	0.35	0.971
21	7	25.71	85.71	**98.57**	72.86	12.86	**1**	**< 0.001**	**0.67**	**0.0503**
22	7	57.14	**100.00**	94.28	37.14	-5.72	**1**	**< 0.001**	0.3	0.987
23	7	57.14	**100.00**	97.14	40.00	-2.86	**1**	**< 0.001**	0.4	0.936
24	7	57.14	**100.00**	95.71	38.57	-4.29	**1**	**< 0.001**	0.35	0.971
25	7	51.43	92.86	**94.28**	42.86	1.43	**1**	**< 0.001**	0.55	0.347
26	6	50.00	66.67	**100.00**	50.00	33.33	**1**	**< 0.001**	**1**	**< 0.001**
27	6	50.00	66.67	**98.33**	48.33	31.66	**1**	**< 0.001**	**1**	**< 0.001**
28	6	50.00	66.67	**96.67**	46.67	30.00	**1**	**< 0.001**	**1**	**< 0.001**
29	6	50.00	66.67	**100.00**	50.00	33.33	**1**	**< 0.001**	**1**	**< 0.001**
30	90	6.89	21.27	**86.33**	79.44	65.06	**1**	**< 0.001**	**1**	**< 0.001**
		Average								
		37.76	68.99	**91.71**	53.95	22.72				

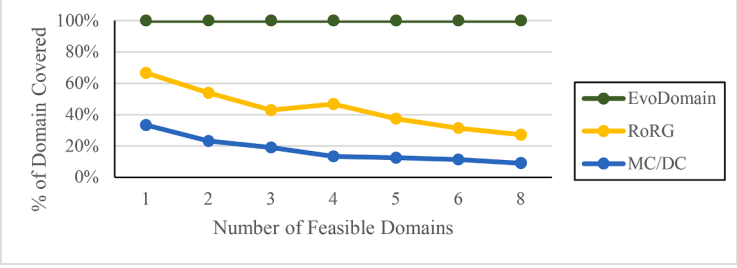


Fig. 3. The domain coverage of the selected approaches.

8). Fig. 3 compares the number of feasible subdomains identified by the proposed approach with the MC/DC and RoRG criteria. Table 6 shows that full MC/DC coverage is insufficient for test success. Domain coverage, where each subdomain is tested at least once, enhances test suite reliability. The proposed method achieved 100% coverage in all cases, while MC/DC and RoRG methods fell short due to their limitations in recognizing all feasible subdomains.

6.2. RQ2: Fault types

This section aims to evaluate the efficacy of different approaches in detecting faults across various subjects and fault types. The results of our study, depicted in Table 7, showcase the identified faults by each approach for each subject and fault type, including ROR, CSM, RVM, AOR, LOR, and SD. The analysis shows that EvoDomain is the most effective approach, outperforming other methods in 74 out of 114 cases. This approach exhibits significant enhancements over the baselines. For a more comprehensive overview, Fig. 4 presents the average number of detected faults categorized by fault type across all subjects attained by each selected approach. The findings indicate that EvoDomain surpasses other approaches in all categories, with the most noteworthy improvement observed in the AOR category. This category holds significant importance as existing logical coverage criteria struggle to capture the internal behavior of predicates. EvoDomain notably enhances MC/DC coverage from 7.78 to 68.89 and RoRG coverage from 2.22 to 66.33.

6.3. RQ3: Practical advantage

In this section, we investigate the effectiveness of fault localization techniques, specifically SBFL methods, which rely on the quality of the test data generation process. Our research focuses on three popular SBFL techniques—Ochiai, Tarantula, and Jaccard—and evaluates their performance concerning three different test suite generation approaches. To assess the effectiveness of these approaches, we analyze the Expense measurement [46]. This measurement determines the percentage of the program that must be examined to find the fault, following a rank list from the top down. The lower the measure, the better the effectiveness. It is defined as Eq. 7, where $|V|$ measures the size of executable codes in

Table 7
The average fault detection rate achieved by the selected approaches based on different fault types for each subject.

Predicate	Approach	Fault Types						Predicate	Approach	Fault Types					
		ROR	CSM	RVM	AOR	LOR	SD			ROR	CSM	RVM	AOR	LOR	SD
1	MC/DC	0	1.57	1	2		4	16	MC/DC	0	3.33	10			
	RoRG	0.33	1.57	1	2.17		9.5		RoRG	8	8	10			
	EvoDomain	3	5.57	10	10		10		EvoDomain	10	9	10			
2	MC/DC	0	4.22	4.5	4.91			17	MC/DC	0	3.33	10			
	RoRG	0	4.22	4.5	4.91				RoRG	9	9	10			
	EvoDomain	6.5	7.56	10	10				EvoDomain	10	8.33	10			
3	MC/DC	0	5.11	5.5	2.5			18	MC/DC	0	3.33	10			
	RoRG	0	5.11	5.5	2.5				RoRG	9	9	10			
	EvoDomain	5	6.72	10	10				EvoDomain	10	8.67	10			
4	MC/DC	0	2.67	2	3.4		6	19	MC/DC	0	3.33	10			
	RoRG	1	3.83	9	7		10		RoRG	9.33	9.67	10			
	EvoDomain	6	5.83	10	10		10		EvoDomain	10	9.33	10			
5	MC/DC	4	5.71	10	0			20	MC/DC	3.33	3.33	10			
	RoRG	4	5.71	10	0				RoRG	10	10	10			
	EvoDomain	4	5.71	10	10				EvoDomain	10	9	10			
6	MC/DC	1.43	3.31	10	1.89			21	MC/DC	0	3.33	8			
	RoRG	1.43	3.88	10	5.33				RoRG	8	8.67	10			
	EvoDomain	7.29	6.75	10	10				EvoDomain	10	9.67	10			
7	MC/DC	2.5	5.38	9	3.91	8	3	22	MC/DC	3.33	5	10		10	
	RoRG	2.5	5.38	9	3.91	8	3		RoRG	10	10	10		10	
	EvoDomain	6.5	7.23	10	10	10	10		EvoDomain	9.33	8	10		10	
8	MC/DC	6	3.33	10	3.33			23	MC/DC	3.33	5	10		10	
	RoRG	6	3.33	10	3.33				RoRG	10	10	10		10	
	EvoDomain	6	3.33	10	10				EvoDomain	10	9	10		10	
9	MC/DC	3.33	3.56	10	0.2			24	MC/DC	3.33	5	10		10	
	RoRG	3.33	3.56	10	0.2				RoRG	10	10	10		10	
	EvoDomain	4.33	4.11	10	10				EvoDomain	10	8.5	10		10	
10	MC/DC	1.67	2.73	10	0			25	MC/DC	3.33	5.5	10		5	
	RoRG	1.67	4	10	9.5				RoRG	8.33	10	10		10	
	EvoDomain	7.5	3.73	10	10				EvoDomain	10	8	10		10	
11	MC/DC	3.33	3.33	10				26	MC/DC	0	5	10		10	
	RoRG	10	10	10					RoRG	5	5	10		10	
	EvoDomain	9.33	8.67	10					EvoDomain	10	10	10		10	
12	MC/DC	0	3.33	10				27	MC/DC	0	5	10		10	
	RoRG	8.67	9	10					RoRG	5	5	10		10	
	EvoDomain	9	8.67	10					EvoDomain	10	9.5	10		10	
13	MC/DC	0	3.33	10				28	MC/DC	0	5	10		10	
	RoRG	5	6.67	10					RoRG	5	5	10		10	
	EvoDomain	9.5	9.33	10					EvoDomain	10	9	10		10	
14	MC/DC	0		10				29	MC/DC	0	5	10		10	
	RoRG	10		10					RoRG	5	5	10		10	
	EvoDomain	10		10					EvoDomain	10	8.5	10		10	
15	MC/DC	0	0	0	0			30	MC/DC	0	1.08	2	0.34		2
	RoRG	0	0	0	0				RoRG	0	2.21	5	1.56		8.5
	EvoDomain	10	10	10	10				EvoDomain	2.29	5.28	10	10		10

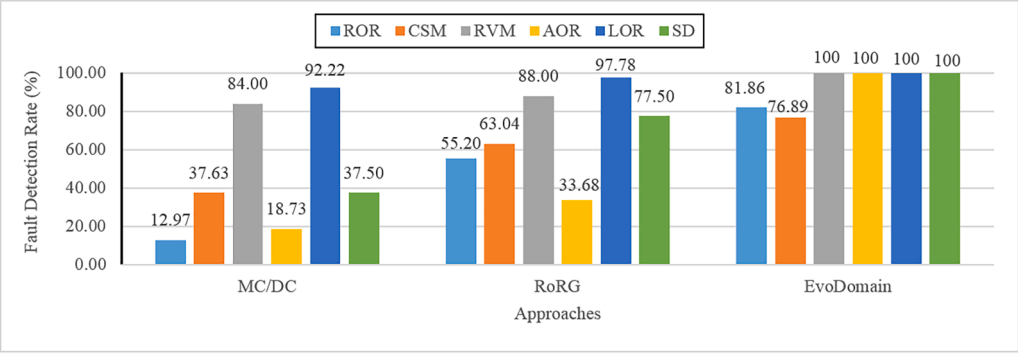


Fig. 4. The average fault detection rate of the selected approaches across all the subjects based on different fault types.

the program, and $|V_{examined}|$ measures the number of statements that must be inspected to find the fault.

$$Expense = \frac{|V_{examined}|}{|V|} * 100 \quad (7)$$

The test suites generated by EvoDomain demonstrate satisfactory behavioral coverage of the test requirements, effectively distributing tests across different subdomains and reducing the code coverage required for fault localization. This is supported by Fig. 5, which quantitatively demonstrates this observation across all subjects.

Fig. 5 shows the effectiveness of conventional SBFL techniques using different test suite generation methods across all faulty versions. The x-

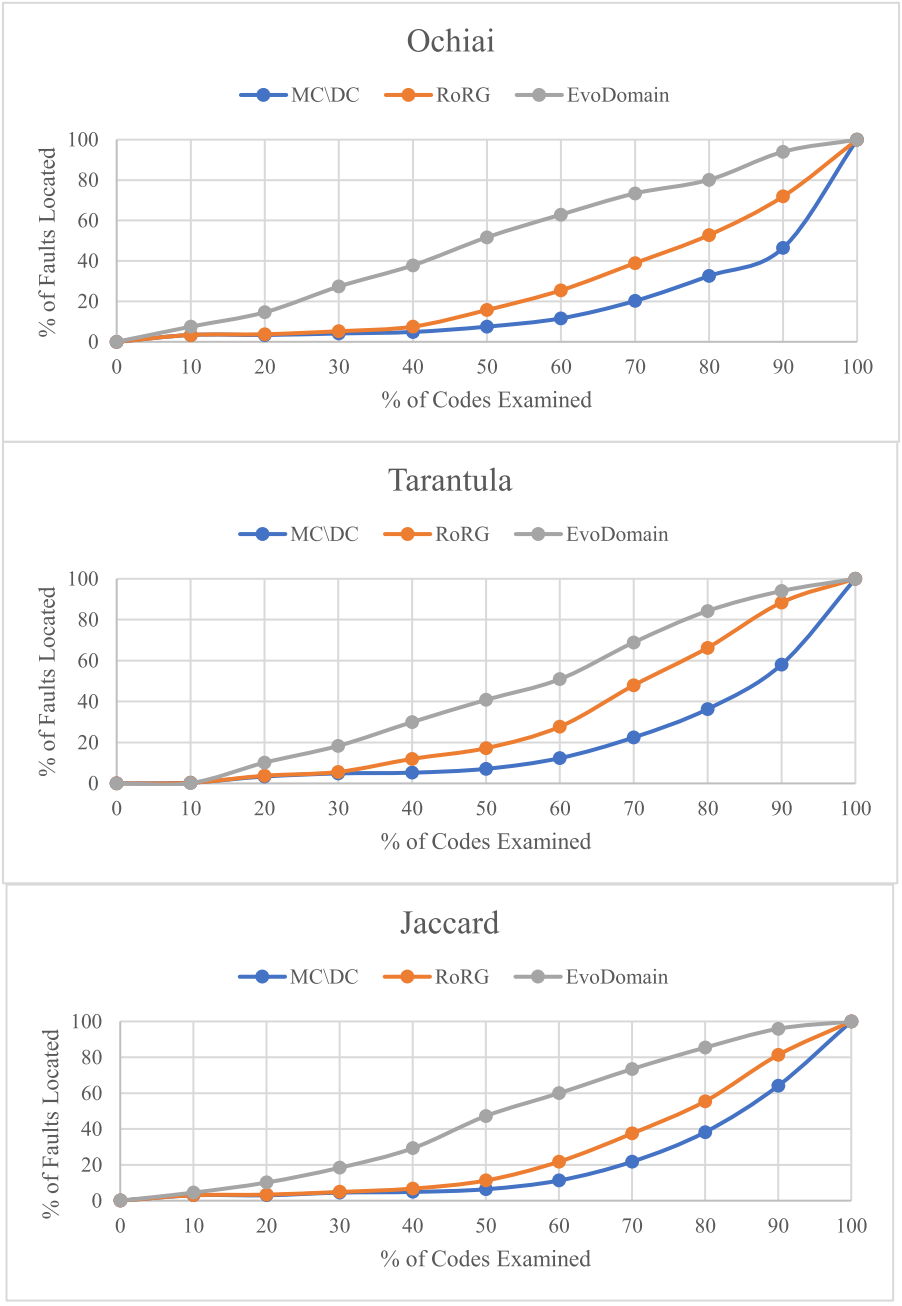


Fig. 5. Effectiveness comparison between conventional SBFL techniques using different test suite generation approaches.

axis represents the percentage of executable statements examined, and the y-axis shows the percentage of faulty versions with located faults. EvoDomain consistently outperforms other approaches, indicating better fault localization. For instance, examining less than 30% of the code, EvoDomain locates faults in 27.34% of faulty versions, compared to 4.12% for MC/DC and 5.24% for RoRG. The improvement in fault localization is consistently high for different SBFL techniques using our approach. For example, Ochiai's performance improves from 37.45 to 53.16, Tarantula from 23.59 to 47.94, and Jaccard from 38.20 to 51.68. Our approach enhances SBFL techniques by covering the internal behavior of predicates, ranking faulty statements at the forefront. This is crucial as many faults occur within specific domains or ranges, which traditional test suite generation techniques may miss.

6.4. RQ4: Performance

To demonstrate how well EvoDomain can identify the subdomain of a program being tested, we have established four criteria: F1-score, accuracy, precision, convergence, and resolution. We divided the generated test data into two classes: one within the domain and the other outside it, to measure the first two criteria. We then applied a random forest classifier to calculate the machine learning evaluation criteria—accuracy and F1-score—as illustrated in Fig. 6. Our results confirm that EvoDomain achieves a high level of accuracy and F1-score, ranging from 0.99 to 1.

Precision, convergence, and resolution are essential criteria for assessing the ability to identify boundaries with minimal samples to the SUT. This is particularly crucial when dealing with high-dimensional state spaces and limited available samples. Precision is the percentage of samples within 0.1 of performance boundaries. Convergence is the

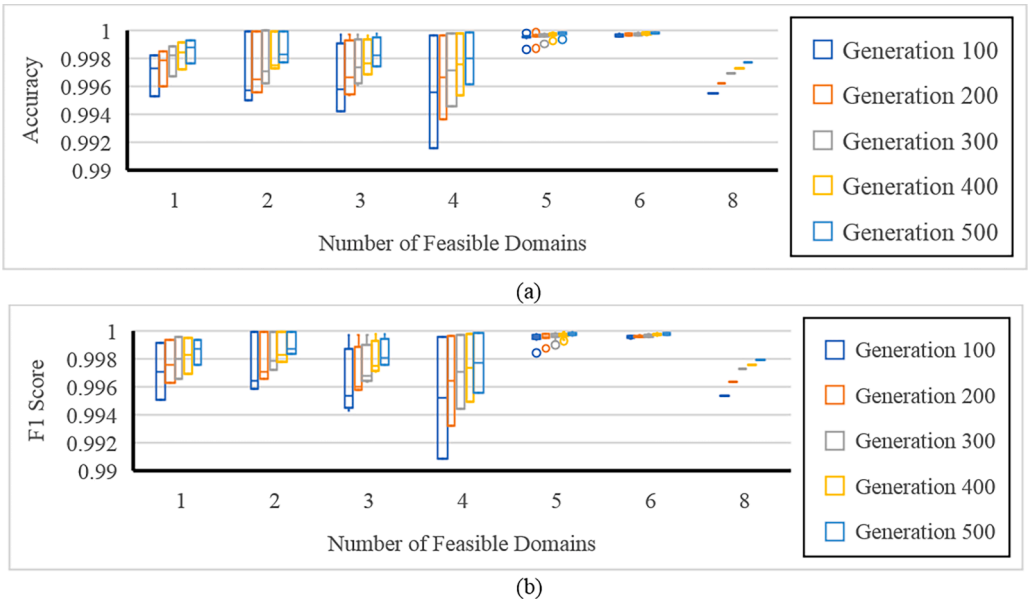


Fig. 6. The (a) accuracy and (b) F1-score achieved by EvoDomain across all the subject predicates during 500 generations.

Table 8
The average precision, convergence, resolution and run time of EvoDomain for each subject predicate.

Predicate No.	Precision	Convergence	Resolution	Run Time (hours)
1	0.45	0.55	0	1.24
2	0.73	0.27	0	1.25
3	0.6	0.4	0	1.09
4	0.89	0.11	0	1.07
5	0.58	0.42	0	1.29
6	0.82	0.18	0	1.46
7	0.96	0.04	0	1.65
8	0.67	0.33	0	1.23
9	0.79	0.21	0	1.18
10	0.54	0.46	0	1.44
11	0.92	0.08	0	1.21
12	0.89	0.11	0	1.18
13	0.95	0.05	0	0.94
14	0.97	0.03	0	0.90
15	0.91	0.09	0	0.93
16	0.83	0.17	0	1.22
17	0.85	0.15	0	1.24
18	0.9	0.1	0	1.24
19	0.94	0.06	0	1.22
20	0.89	0.11	0	1.23
21	0.91	0.09	0	1.43
22	0.88	0.12	0	1.44
23	0.86	0.14	0	1.64
24	0.79	0.21	0	1.51
25	0.9	0.1	0	1.47
26	0.86	0.14	0	1.55
27	0.91	0.09	0	1.49
28	0.97	0.03	0	1.51
29	0.97	0.03	0	1.60
30	0.34	0.66	0	1.65

number of inquiries needed to sample the boundary instances. Resolution measures the distance from the closest boundary. Table 8 shows average values and runtime, with higher precision and lower convergence and resolution being preferable. Results are analyzed using clause complexity from Table 5.

Precision averages around 0.80, ranging from 0.34 to 0.97. The highest precision (0.97) occurs at CC values of 0.6 and 3, while the lowest (0.34) is at 8.4, indicating higher complexity challenges precision. For a CC of 3, precision varies from 0.58 to 0.97, warranting further exploration of influencing factors. Convergence values range from 0.03 to 0.66, with the highest (0.66) at a CC of 8.4 and the lowest (0.03) at CC values of 0.6 and 3. Higher complexity generally requires more sampling to find all performance boundaries. For a CC of 3, convergence varies from 0.03 to 0.42. All samples have a resolution value of zero because the nearest point to the domain is on the domain, making these points identifiable in the EvoDomain method. CC and runtime have a nuanced relationship. Moderate CC values around 3 show varied runtimes (1.18 to 1.64 hours), suggesting other influencing factors. The highest CC value (8.4) does not have the longest runtime, hinting at optimization algorithms' roles. Lower CC values, like 0.6, generally have shorter

Table 9
Comparison of relative accuracy between static and dynamic domain-oriented test suite generation approaches.

Predicate No.	EvoDomain	COSMOS	Improvement
1	**91**	Not Found	91
2	**87**	82	5
3	**88**	Not Found	88
4	**71**	Not Found	71
5	**95**	90	5
6	82	**86**	-4
7	**80**	Not Found	80
8	89	**92**	-3
9	73	**83**	-10
10	**57**	46	11
11	**82**	77	5
12	**83**	69	14
13	**77**	56	21
14	**69**	37	32
15	**54**	42	12
16	**87**	85	2
17	**85**	67	18
18	**91**	89	2
19	86	**91**	-5
20	**88**	85	3
21	**91**	73	18
22	76	**80**	-4
23	**88**	**88**	0
24	**85**	72	13
25	**87**	81	6
26	**87**	86	1
27	**86**	78	8
28	**89**	84	5
29	**84**	79	5
30	**92**	Not Found	92
Average	**83**	76	19

runtimes. Thus, CC alone is not a definitive runtime predictor, highlighting the need to consider multiple factors.

6.5. RQ5: Comparison with COSMOS

COSMOS is an advanced and efficient static domain-oriented test suite generation method capable of handling various data types, structures, and logic. COSMOS can solve linear and non-linear constraints more accurately than other constraint solvers. COSMOS uses a simple sampling approach based on random search to compute the domain of function arguments, which may affect its accuracy. Moreover, COSMOS depends heavily on a hyperparameter to partition the initial domain. On the other hand, EvoDomain uses a heuristic search to find feasible subdomains, making it more likely to accurately detect domains than COSMOS. Table 9 provides a quantitative comparison of the relative accuracy of domain identification in EvoDomain and COSMOS, confirming EvoDomain's superiority. Relative accuracy is computed by dividing the number of selected test data from the suggested domain rejected by the underlying predicate by the size of the requested test suite. Although COSMOS outperforms EvoDomain in five out of 25 instances, it could not find a solution using the default configuration setup in five out of 30 cases. In 24 instances, EvoDomain is superior to COSMOS, improving it by up to 32%.

6.6. RQ6: Search effectiveness

In this section, we compare the search effectiveness of EvoDomain and a random search algorithm in identifying feasible domains. To make a fair comparison, we allocated twice the time budget to the random search algorithm. The random search algorithm used the same DBSCAN clustering algorithm as EvoDomain to detect disconnected sub-domains. We evaluated the conformance of EvoDomain and the random search algorithm to the hypercubes representing each subdomain. The results indicate that both approaches consistently detected the subdomains. Fig. 7 shows that generating a domain-oriented test suite is challenging, as the random search algorithm took up to four times the test budget to reach solutions. Therefore, EvoDomain outperforms the random search algorithm in terms of execution time.

6.7. Threats to validity

This section discusses the threats to the study's validity and the steps taken to mitigate them. The threats are categorized as internal, construct, conclusion, and external.

Our study's accuracy and reliability face several threats to validity. We used the Memetic optimization algorithm with recommended hyperparameters, but configuring these could introduce additional threats. Randomness in EvoDomain is another internal threat, so we ran each experiment 10 times to mitigate this. Although 30 repetitions are standard [47], resource constraints and the extensive time required (around 35 weeks) made this impractical. We believe 10 repetitions still provide a robust and reliable dataset for analysis.

Threats to internal validity can arise from the study's conduct. Although we carefully tested our framework to minimize faults, testing alone cannot guarantee the absence of defects. To address this, we repeated the selection process multiple times in our randomized algorithms.

Measuring performance in terms of mutation score, execution time, and expense score could introduce threats to construct validity. Random variations can threaten conclusion validity; therefore, we repeated the experiments multiple times to minimize their impact. We used the Wilcoxon rank-sum test for statistical significance and Vargha and Delaney's $\hat{A}_{12}$ statistics for the effect size measure.

External validity can be threatened if the results are not generalizable beyond the specific programs. To address this, we considered 30 subject predicates extensively used in the literature and compared the results with four diverse state-of-the-art test generation baselines. We also utilized a diverse set of six mutation operators during mutation testing. Lastly, we selected the Expense score as a well-known metric to evaluate the fault localization effectiveness of our proposed approach and the baselines, addressing the external threat related to the fault proneness criterion.

7. Related work

This section reviews MC/DC-based test data generation approaches and summarizes software fault localization techniques, emphasizing the impact of effective test data generation.

MC/DC-based Test Data Generation. Significant progress has been made in software testing, particularly in generating test data with an emphasis on the MC/DC coverage criterion. Pandita et al. [48] introduced a method using test-generation tools for comprehensive input creation, though its effectiveness varies with program complexity. Wu et al. [49] enhanced this approach by employing Control Flow Graphs and condition vectors, significantly increasing coverage rates. A novel technique leveraging Abstract Syntax Trees was proposed to detect faults in Boolean expressions, showing promising results [50].

BOOMPizer [51] emerged as a technique aimed at optimizing MC/DC test case creation and reduction, demonstrating substantial improvements in test efficiency and fault detection. Yu et al.'s study [52] underscored the importance of MC/DC compliance in airborne software, emphasizing the necessity of advanced testing criteria for enhancing software quality and safety.

Search-based software engineering (SBSE) techniques, including Simulated Annealing, Hill Climbing, and Genetic Algorithms, have proven effective in generating high-quality test data for MC/DC coverage [53-55]. Ghani et al. [53] developed a framework utilizing SBSE to meet desired coverage criteria. This showcases the potential of

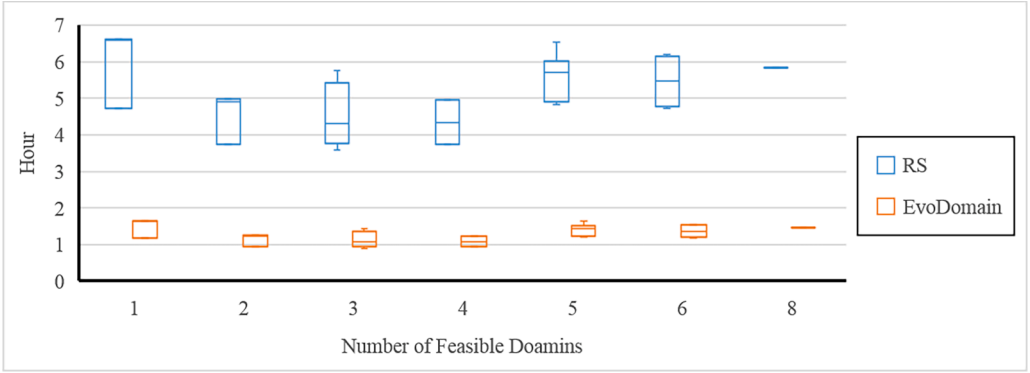


Fig. 7. The comparison of the search effectiveness between EvoDomain and RS.

these methods in raising testing standards. Awedikian et al. [54] introduced a genetic algorithm-based approach for MC/DC test input generation, indicating its superiority over branch distance adaptation.

Offutt et al. [3] proposed enhancements to logic-based test design, advocating for the adoption of minimal-MUMCUT over MC/DC because of its superior fault detection capabilities. Paul et al. [56] conducted a comprehensive review of MC/DC variants, introducing a new form and highlighting the importance of achieving 100% MC/DC coverage through automated tools.

Ayav et al. [57] presented an affordable method for prioritizing MC/DC test cases using Fourier analysis, providing a cost-effective solution for enhancing software testing procedures. Vilkomir et al. [58] provided definitions and instructions for assessing MC/DC in black-box testing, aiming to improve software testing quality and productivity.

In model-based testing (MBT), applying MC/DC coverage has shown significant increases in code coverage and fault detection [59]. However, achieving full MC/DC coverage remains challenging due to the complexity of test case generation. Research suggests that exploring efficient metaheuristic algorithms could optimize MC/DC test case generation [60,61].

Zafar et al. [62] found MBT-generated test suites combined with manual testing to be effective in achieving MC/DC adequacy. Čegiň et al. [63] explored reinforcement learning as a new method for generating test data, potentially solving path explosion problems faced by symbolic methods. Sartaj et al. [64] suggested reusing previously generated test data for achieving MC/DC coverage through a search-based strategy, indicating greater efficiency compared to generating new test data.

Fault localization. Software fault localization is crucial for debugging, aiming to pinpoint faults in a program. With software systems becoming more complex, effective fault localization techniques are vital for maintaining quality and reducing debugging time. Key techniques include program spectrum-based methods, slice-based methods, model-based approaches, and statistical debugging [65]. Spectrum-based methods analyze execution traces of test cases to identify suspicious code segments by comparing passing and failing test cases. Slice-based methods create program slices, which are subsets of the program affecting values at a specific point, to narrow down fault locations. Model-based approaches use formal models to identify discrepancies between expected and actual behavior. Statistical debugging employs statistical techniques to correlate program behaviors with failures, identifying likely fault locations.

Recent studies highlight that test suites enhance fault localization efficiency. Adjusting test case weighting improves suspiciousness scores, and smaller, randomly selected subsets can outperform larger test suites [66]. The passing test discrimination (PTD) metric, which evaluates the contribution of passing test cases to fault suspicion, further enhances fault localization when integrated with existing techniques [67].

Although this paper proposes a test data generation approach, it provides a preliminary analysis of the advantages of domain-oriented test suite generation for fault localization compared to the mentioned studies, showing a promising avenue for future research.

8. Conclusion and future works

EvoDomain is a dynamic approach for generating domain-oriented test suites using a memetic algorithm that combines genetic and hill-climbing algorithms. It aims to enhance the fault detection capabilities of MC/DC. Evaluations using 30 classic and industrial case studies show promising results, with a 74.44% increase in fault detection rate compared to MC/DC and a 65.06% increase compared to RoRG. EvoDomain also improves support for different fault types of MC/DC by up to 68.89% and RoRG by up to 66.33%. Additionally, it demonstrates significant practical benefits in improving fault localization techniques, achieving enhancements of up to 53.16% for Ochiai, 47.97% for Tarantula, and 51.68% for Jaccard. In domain identification, EvoDomain shows desirable accuracy, F1-score, precision, and convergence, and finds feasible subdomains in less than one-third the time compared to random search. It also shows a 32% improvement in convergence effectiveness compared to COSMOS.

While the evaluation results are promising, future work could refine the memetic algorithm in EvoDomain, explore its benefits on other SBFL approaches, and investigate its impact on other testing techniques like t-way testing. Additionally, assessing EvoDomain's performance across a broader range of case studies would help evaluate its robustness and versatility.

CRediT authorship contribution statement

Akram Kalaee: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Resources, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Saeed Parsa:** Writing – review & editing, Validation, Supervision, Project administration. **Zahra Mansouri:** Visualization, Software, Resources, Investigation, Data curation.

Declaration of competing interest

The authors declare that they have no competing interests as defined by Springer or other interests that might be perceived to influence the results and discussion reported in this paper.

Data availability

We have shared the link to our code and data in the manuscript file.

Funding declaration

No funding was received for this paper.

References

[1] K. Naik, P. Tripathy, Software testing and quality assurance: theory and practice, John Wiley & Sons, 2011.
[2] P. Ammann, J. Offutt, Introduction to software testing, Cambridge University Press, 2016.
[3] G. Kaminski, P. Ammann, J. Offutt, Improving logic-based testing, Journal of Systems and Software 86 (8) (2013) 2002–2012.
[4] A. Gotlieb, M. Petit, A uniform random test data generator for path testing, Journal of Systems and Software 83 (12) (2010) 2618–2626.
[5] E. Nikravan, S. Parsa, Path-oriented random testing through iterative partitioning (IP-PRT), Turkish Journal of Electrical Engineering and Computer Sciences 27 (4) (2019) 2666–2680.
[6] A.J. Offutt, Z. Jin, J. Pan, The dynamic domain reduction procedure for test data generation, Software: Practice and Experience 29 (2) (1999) 167–193.
[7] E. Nikravan, S. Parsa, Improving dynamic domain reduction test data generation method by Euler/Venn reasoning system, Software Quality Journal 28 (2) (2020) 823–851.
[8] A. Kalaee, S. Parsa, N. Fathi, COSMOS: A comprehensive framework for automatically generating domain-oriented test suite, Inf. Softw. Technol. 154 (2023) 107091.
[9] Haonan Li, Yu Hao, Yizhuo Zhai, Zhiyun Qian, Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach, in: Proceedings of the ACM on Programming Languages, 2024, pp. 474–499, 8, no. OOPSLA1.
[10] K. Khan, S.U. Rehman, K. Aziz, S. Fong, S. Sarasvady, February. DBSCAN: Past, present and future, in: The fifth international conference on the applications of digital information and web technologies (ICADIWT 2014), IEEE, 2014, pp. 232–238.
[11] H.A. De Souza, M.L. Chaim, F. Kon, arXiv preprint, 2016.
[12] P. McMinn, Search-based software test data generation: a survey, Software testing, Verification and reliability 14 (2) (2004) 105–156.
[13] S.N. Sivanandam, S.N. Deepa, Genetic algorithms, Springer, Berlin Heidelberg, 2008, pp. 15–37.
[14] N. Tracey, J. Clark, K., Mander, and J. McDermid. "An automated framework for structural test-data generation." Proceeding 13th IEEE International Conference on Automated Software Engineering, Hawaii, HI, USA, pp. 285–288, 1998.
[15] C. Wilt, J. Thayer, W. Ruml, A comparison of greedy search algorithms, Proceedings of the International Symposium on Combinatorial Search 1 (1) (2010) 129–136.
[16] C. Cotta, L. Mathieson, P. Moscato, Memetic algorithms. Handbook of Heuristics, Springer, 2017, pp. 1–32.

[17] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in Computers, 112, Elsevier, 2019, pp. 275–378.
[18] Software considerations in airborne systems and equipment certification. Washington, RTCA, Inc., December 1992.
[19] King, K.N., Offutt, J., 1991. A Fortran language system for mutation-based software.
[20] J.A. Jones, M.J. Harrold, J. Stasko, Visualization of test information to assist fault localization, in: Proceedings of the 24th international conference on Software engineering, 2002, pp. 467–477.
[21] R. Abreu, P. Zoeteweij, A.J. Van Gemund, On the accuracy of spectrum-based fault localization, in: Testing: Academic and industrial conference practice and research techniques-MUTATION, IEEE, 2007, pp. 89–98. TAICPART-MUTATION 2007.
[22] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: Problem determination in large, dynamic internet services, in: Proceedings International Conference on Dependable Systems and Networks, IEEE, 2002, pp. 595–604.
[23] M. Harman, P. McMinn, A theoretical and empirical analysis of evolutionary testing and hill climbing for structural test data generation, in: Proceedings of the 22nd International Conference on Software Engineering, ICSE, 2010.
[24] P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. Caltech Concurrent Computation Program, C3P Report 826 (1989).
[25] D.E. Goldberg, K. Sastry, A practical schema theorem for genetic algorithm design and tuning, in: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO), 2001, pp. 328–335.
[26] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Transactions on Evolutionary Computation 6 (2) (2002) 182–197.
[27] N. Srinivas, K. Deb, Multiobjective optimization using nondominated sorting in genetic algorithms, Evol. Comput. 2 (3) (1994) 221–248.
[28] N. Kosmatov, B. Legeard, F. Peureux, M. Utting, Boundary coverage criteria for test generation from formal models, in: Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE), 2004, pp. 139–150, pages.
[29] F. Dobslaw, F.G. de Oliveira Neto, R. Feldt, Boundary value exploration for software analysis, in: Proceedings of the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2020, pp. 346–353, pages.
[30] J.A. Briem, J. Smit, H. Sellik, P. Rapoport, G. Gousios, M. Aniche, OffSide: Learning to Identify Mistakes in Boundary Conditions, in: Proceedings of the 2nd Workshop on Testing for Deep Learning and Deep Learning for Testing (DeepTest), 2020.
[31] S Katoch, SS Chauhan, V. Kumar, A review on genetic algorithm: past, present, and future, Multimed. Tools. Appl. 80 (5) (2021) 8091–8126.
[32] A. Parmar, R. Katariya, V. Patel, A review on random forest: An ensemble classifier, in: International conference on intelligent data communication technologies and Internet of things (ICICI) 2018, Springer International Publishing, 2019, pp. 758–763.
[33] D. Anguita, L. Ghelardoni, A. Ghio, L. Oneto, S. Ridella, The'K'in K-fold Cross Validation. ESANN, 2012, pp. 441–446.
[34] MutMut, "Mutmut 1.9.0: Python mutation tester," 2024. [Online]. Available: https://mutmut.readthedocs.io/en/latest/. [Accessed: 13- September- 2023].
[35] Software-artifact Infrastructure Repository. (2023) 'Tcas', available at: https://sir.csc.ncsu.edu/portal/bios/tcas.php (Accessed: 26 October 2023).
[36] H. Nowacki, Modelling of design decisions for CAD. Computer Aided Design Modelling, Systems Engineering, CAD-Systems, CREST Advanced Course Darmstadt (2005) 177–223, 8.–19September 1980.
[37] C. Mao, Harmony search-based test data generation for branch coverage in software structural testing, Neural Comput. Appl. 25 (1) (2014) 199–216.
[38] P. Ammann, J. Offutt, H. Huang, Coverage criteria for logical expressions, in: Proceedings of the 14th International Symposium on Software Reliability Engineering, IEEE Computer Society Press, Denver, CO, 2003, pp. 99–107. November.
[39] H. Alibrahim, S.A. Ludwig, Hyperparameter Optimization: Comparing Genetic Algorithm against Grid Search and Bayesian Optimization, IEEE Congress on Evolutionary Computation (CEC), Kraków, Poland, 2021, 2021.
[40] I.D. Raji, H. Bello-Salau, I.J. Umoh, A.J. Onumanyi, M.A. Adegboye, A. T. Salawudeen, Simple Deterministic Selection-Based Genetic Algorithm for Hyperparameter Tuning of Machine Learning Models, Applied Sciences 12 (3) (2022) 1186.
[41] S.A. Ludwig, H. Alibrahim, Hyperparameter Tuning in Machine Learning Models, North Dakota State University, 2021.
[42] P. Sarang, DBSCAN: Density-Based Spatial Clustering of Applications with Noise. Thinking Data Science, Springer, 2023, pp. 209–234.
[43] M. Schonlau, Random Forest Algorithm: Parameters and Performance. Applied Statistical Learning, Springer, 2023, pp. 183–204.
[44] J.A. Capon, Elementary statistics for the social sciences: Study guide, Wadsworth Publishing Company, Belmont, CA, USA, 1991.
[45] Vargha, H.D. Delaney, A critique and improvement of the cl common language effect size statistics of McGraw and Wong, Journal of Educational and Behavioral Statistics 25 (2) (2000) 101–132.
[46] Y. Li, C. Liu, Effective fault localization using weighted test cases, J. Softw. 9 (8) (2014) 2112–2119.
[47] Andrea Arcuri, Lionel Briand, A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering, Software Testing, Verification and Reliability 24 (3) (2014) 219–250.
[48] R. Pandita, T. Xie, N. Tillmann, J. De Halleux, Guided test generation for coverage criteria, in: 2010 IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–10.
[49] T. Wu, J. Yan, J. Zhang, Automatic test data generation for unit testing to achieve MC/DC criterion, in: 2014 Eighth International Conference on Software Security and Reliability (SERE), IEEE, 2014, pp. 118–126.
[50] T.K. Paul, M.J.M. Chowdhury, M.F. Lau, A new disjunctive literal insertion fault detection strategy in boolean specifications, Journal of Software: Evolution and Process 33 (5) (2021) e2336.
[51] S.K. Barisal, S.P.S. Chauhan, A. Dutta, S. Godboley, B. Sahoo, D.P. Mohapatra, Boompizer: Minimization and prioritization of concolic based boosted mc/dc test cases, Journal of King Saud University-Computer and Information Sciences 34 (10) (2022) 9757–9776.
[52] Y.T. Yu, M.F. Lau, A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions, Journal of Systems and Software 79 (5) (2006) 577–590.
[53] K. Ghani, J.A. Clark, Automatic test data generation for multiple condition and MCDC coverage, in: Software Engineering Advances, 2009. ICSEA'09. Fourth International Conference on, IEEE, 2009.
[54] Z. Awedikian, K. Ayari, G. Antoniol, Mc/dc automatic test input data generation, in: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, ACM, 2009, pp. 1657–1664.
[55] J. Minj, Feasible Test Case Generation Using Search Based Technique, Int. J. Comput. Appl. 70 (28) (2013) 4.
[56] T.K. Paul, M.F. Lau, A systematic literature review on modified condition and decision coverage, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 1301–1308.
[57] T. Ayav, Prioritizing MCDC test cases by spectral analysis of Boolean functions, Software Testing, Verification and Reliability 27 (7) (2017) e1641.
[58] S. Vilkomir, J. Baptista, G. Das, Using mc/dc as a black-box testing technique, in: 2017 IEEE 28th Annual Software Technology Conference (STC), IEEE, 2017, pp. 1–7.
[59] S.S. Arefin, H. Hemmati, H.W. Loewen, Evaluating specification-level mc/dc criterion in model-based testing of safety critical systems, in: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, 2018, pp. 256–265.
[60] A. Haque, K.Z. Zamli, Search Based Test Data Generation Strategy for Safety Critical Software using MC/DC Criterion, in: 2015 National Postgraduate Conference, IEEE, 2015, pp. 1–10.
[61] A. Haque, An Experimental Study of Neighbourhood Based Metaheuristic Algorithms for Test Case Generation Satisfying the Modified Condition /Decision Coverage Criterion. faculty of computer systems and software engineering, Universiti malaysia pahang, Kuantan, 2018.
[62] M.N. Zafar, W. Afzal, E. Enoiu, Evaluating system-level test generation for industrial software: A comparison between manual, combinatorial and model-based testing, in: Proceedings of the 3rd ACM/IEEE International Conference on Automation of Software Test, 2022, pp. 148–159.
[63] J. Čegiň, K. Rástočný, Test Data Generation for MC/DC Criterion using Reinforcement Learning, in: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), IEEE, 2020, pp. 354–357.
[64] H. Sartaj, M.Z. Iqbal, A.A.A. Jilani, M.U. Khan, A search-based approach to generate mc/dc test data for ocl constraints, in: Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Springer International Publishing, Tallinn, Estonia, 2019, pp. 105–120. August 31–September 1, 2019Proceedings 11.
[65] W.Eric Wong, T.H. Tse, Handbook of software fault localization: foundations and advances, John Wiley & Sons, 2023.
[66] Yihan Li, Chao Liu, Effective fault localization using weighted test cases, J. Softw. 9 (8) (2014) 2112–2119.
[67] Yan Lei, Chengnian Sun, Xiaoguang Mao, Zhendong Su, How test suites impact fault localisation starting from the size, IET Software 12 (3) (2018) 190–205.