Soft Computing
https://doi.org/10.1007/s00500-024-09680-5

APPLICATION OF SOFT COMPUTING

Dynamic domain testing with multi-agent Markov chain Monte Carlo method

Roshan Golmohammadi[1] · Saeed Parsa[1] · Morteza Zakeri-Nasrabadi[1]

Accepted: 15 January 2024

Abstract
Path testing is one of the most efficient approaches for covering a program during the test. However, executing a path with a single or limited number of test data does not guarantee that the path is fault-free, specifically in the fault-prone paths. A common solution in these cases is to extract the corresponding domain of the path constraint that covers the path and select the arbitrary numbers of test data from the extracted domain. Many approaches have been proposed to extract and partition the program input domain statically without executing the program under test. However, extracting domain based on the path constraints is inaccurate and time-consuming, specifically in high dimensional inputs. This paper presents a dynamic approach for extracting the input domain of a given path based on the modified Markov chain Monte Carlo method, multi-agent Metropolis algorithm. The algorithm uses sampling and randomly walks around symmetric proposal distribution to find a finite number of points that approximate the target domain. Our approach chooses the highest fault-prone path from the minimum-cost test paths for domain extraction to maximize the fault detection capabilities. Our experiments with ten Python programs reveal a 14% improvement in the accuracy of the extracted domains compared to the state-of-the-art domain extraction approaches. In addition, the mutation score of automatically generated test cases has improved by an average of 17.95% compared to branch and prime-path testing approaches.

Keywords Automatic test data generation · Domain extraction · Path testing · Minimum cost test path · Markov chain Monte Carlo

1 Introduction

Testing is one of the most important steps to verify program correctness during the software development life-cycle (SDLC). Automatic test data generation approaches aim to improve test quality and reduce test costs (Jain and Porwal 2019). Path testing is a white box testing technique that includes selecting paths in a program, finding input test data that executes these paths, and examining the results of executed paths against given test oracles (Khari et al. 2018). The primary goal of path testing is to generate input data for the program that covers all program paths with the least effort to find faults (Khari and Kumar 2019). The advantage of path testing is that the path coverage criterion subsumes other coverage criteria, including statement and branch coverage (Ammann and Offutt 2016). Therefore, it is more likely to find faults. However, executing a given path with single test data may not necessarily find the existing faults (Gotlieb and Petit 2006). The paths should often be executed with numerous test data to reveal latent faults. The domain extraction approach aims to discover a subdomain of input variables that executes a single path. In this way, the test engineer can execute the program paths, specifically the critical paths, with arbitrary test data from the extracted domain to verify the program under test. Therefore, domain extraction is necessary when testing the fault-prone paths of the program under test. Domain-oriented test data generation is needed to generate diverse test

✉ Saeed Parsa
parsa@iust.ac.ir

Roshan Golmohammadi
r_golmohammadi@comp.iust.ac.ir

Morteza Zakeri-Nasrabadi
morteza_zakeri@comp.iust.ac.ir

[1] School of Computer Engineering, Iran University of Science and Technology, Hengam St., Resalat Sq., Tehran 16846-13114, Iran

Published online: 04 July 2024

data that can detect and locate faults in high-quality systems. This testing approach is especially crucial for scientific and engineering (Nanthaamornphong and Carver 2017; Kelly et al. 2011), mission-critical (Dubrova 2013), and safety–critical software (Dubrova 2013; Ramler et al. 2017), where one fault can cause significant economic loss or endanger human life (Ammann and Offutt 2016; Jones and Bonsignour 2012). However, the state-of-the-art path testing method only generates one limited test data that executes the target path.

Some approaches test the path by homogeneous partitioning, assuming the same fault detection capability, and selecting a test data sample from these partitions (Hamlet and Taylor 1990; Goodenough and Gerhart 1975). An input partition or subdomain is considered covered in these approaches when one point inside the partition is selected and submitted to the program under test (Gotlieb and Petit 2010). The main assumption behind the homogeneous partitioning approaches is that test data belonging to a partition have equal fault detection capability. Goodenough and Gerhart (1975) have called this assumption "reliability", while Hamlet and Taylor (1990) have called it "homogeneity" of the partition. These approaches are inaccurate for several reasons, mainly because homogeneity is not generally a correct assumption (Hamlet and Taylor 1990). The basic principle of path testing states that testing a selected path with a point from a homogeneous subdomain is sufficient to ensure the program path is correct. Hamlet and Taylor (1990) have shown that the assumption of path correction fails when a subdomain lacks homogeneity. They have suggested that a uniform distribution across the subdomains is more proper than executing a path with a single test data since the domain homogeneity assumption could not be confirmed. Test data that execute the same path do not have the same fault detection capability, and sampling from the associated subdomain increases the likelihood of selecting the failure-causing input.

Automatic test data generation approaches that use uniform distribution sampling might generate test data that do not execute the target path. In addition, not all data points in the subdomain of a program path have the same fault detection capability. On the other hand, all input domain partitioning methods suffer from high time complexity due to frequently calling constraint solvers with high-dimensional inputs (Gotlieb and Petit 2006; Huang et al. 2015). Statically extracting path constraints with symbolic execution often does not represent a given path's dynamic behavior. Besides, there are severe problems with symbolic execution approaches, such as path explosion (Baldoni et al. 2019). Search-based test data generation approaches address the symbolic execution difficulties by executing relevant parts of programs (Mishra et al. 2019; Sahoo and Ray 2020). The aim is to maximize coverage criteria while generating tests. However, the test data covering a path are generated only in limited numbers. In contrast, many test data are necessary to detect hidden errors that are not detectable with only one or two test data. Therefore, search-based approaches rarely generate failure-causing inputs.

This paper proposes an approach that generates goal-oriented random test data for the most fault-prone path using an intelligent sampling process. The proposed approach generates the least number of data points that deviate from the target path. To this aim, the Metropolis algorithm from the Markov chain Monte Carlo (MCMC) methods' family (Changye and Robert 2020) is used, which intellectually finds samples of the input domain that execute the target path. We devise a multi-agent version of the Metropolis algorithm in which agents cooperate to achieve high-density regions in high dimensional input spaces. Cooperation between agents is performed by a scoring mechanism such that agents with higher scores direct the search toward dense regions. Not all paths in a program have the same importance and error rate when testing the program. On the other hand, it is impossible to cover all paths in a program due to the many paths. Our approach chooses the most fault-prone path from the *minimum-cost test paths* (MCTP) (Li et al. 2012) for domain extraction to maximize the fault detection capabilities. The paths used are those of the type MCTP. Hence, the program under test is executed and tested with the least number of paths and the largest number of test data per fault-prone path, improving the domain fault detection capability. The primary contributions of this paper can be summarized as follows:

(i) A dynamic domain extraction algorithm is proposed based on the Markov chain Monte Carlo (MCMC) method to accurately approximate a given path's domain. This enables multiple tests of the specific path using various test data to identify faults.
(ii) The Metropolis sampling algorithm, employing the Markov chain Monte Carlo (MCMC) method, is adapted as a multi-agent search technique to enhance the identification of high-density regions within complex input spaces. Through the utilization of multiple agents, this approach enables the exploration of diverse areas within the domain of the given path, thereby improving the generation of comprehensive test data. Sampling from varied domain regions increases the probability of detecting faults.
(iii) A dynamic domain extraction algorithm is presented. The algorithm is based on the Markov

chain Monte Carlo (MCMC) method, which accurately approximates the domain of a given path. This approach enables multiple tests to be conducted on the specific path using various test data to identify faults.

We compare our proposed approach with four state-of-the-art test data generation approaches (Huang et al. 2015; Lukasczyk et al. 2020; Monemi Bidgoli and Haghighi 2020; Huang et al. 2020) focusing on path and domain testing. Our experimental results with ten Python-based software programs show the superiority of our method in covering boundary regions and detecting faults.

The remaining sections of the paper are organized as follows. Section 2 reviews the related work on domain-oriented test data generation. In Sect. 3, we describe the proposed approach. Section 4 outlines our experiment to evaluate the proposed method. Threats to validity are discussed in Sect. 5. Finally, Sect. 6 concludes the paper and outlines future work.

2 Related works

The main idea behind domain testing in this research has been to determine regions of a program input space revealing coincidentally correct executions (Parsa 2023; Wu et al. 2022; Xue et al. 2014). In the first position, the objective of an appropriate test data generation algorithm should be to covert the fault-prone paths (Birt et al. 2004). Apparently, there is a higher chance of finding coincidentally correct executions along a fault-prone path. However, the related works show that there have been no considerations for fault proneness and coincidental correctness simultaneously. Adaptive random testing approaches tend to detect faulty execution by increasing the diversity while generating test data. In contrast, the domain-oriented test data generation methods focus on generating test data covering a specific execution path. The following two subsections further describe and evaluate adaptive random and domain-oriented test data generation techniques.

2.1 Adaptive random testing

Adaptive random testing approaches aim to maximize the diversity of generated test data considering failure patterns (Chan et al. 1996). Three types of experimentally observed failure patterns are shown in Fig. 1.

According to Chan et al. (1996), as the diversity of the input test data increases, there will be a higher chance of detecting faulty executions. Therefore, it is observed that adaptive random testing (ART) approaches aim to maximize the diversity of generated test data considering failure

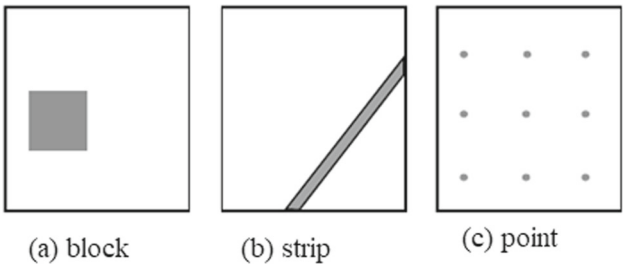

Fig. 1 Failure patterns for a two-dimensional input domain

patterns (Chan et al. 1996). Adaptive random testing (Huang et al. 2021; Chen et al. 2010) attempts to improve the failure-detection rate by generating test data that spread out as widely as possible in the program input space (Wu et al. 2020). Indeed, test cases are randomly generated but subject to additional constraints and processes to ensure the generated test data meets the "widespread" criterion. Researchers and practitioners have proposed different versions of adaptive random testing (Chan et al. 2006; Chen et al. 2004; Zhou et al. 2013) based on the preliminary work by Chen et al. (2001).

Adaptive random testing (ART) (Chan et al. 1996) focuses the testing effort on potentially more suspicious or likely-to-fail areas of the system. The basic principle of ART is to divide the input domain into subdomains and then randomly select test inputs from these subdomains to promote diversity. If a failure is detected during testing, the subdomain from which the failing test input was selected is further divided into smaller subdomains. This adaptive division allows ART to concentrate testing on the potential causes of failures.

Distance-based ART (DART) (Huang et al. 2020; Chan et al. 2006) enhances the diversity using the distance between a newly generated test case and previously executed test cases. The approach can be implemented on programs with numerical inputs (Chen et al. 2007). Although the distance of each test data from the previous one maintains the diversity of test cases, it does not guarantee to meet the test requirements and fault detection capability.

Mirror adaptive random testing (MART) is a revised version of the ART algorithm devised to improve the performance of the ART by the division and conquer approach (Chen et al. 2004). MART first divides the whole input domain into equal-sized disjoint subdomains. Then, one subdomain is selected as the source domain, while others are chosen as mirror domains. MART generates test cases in the source domain according to one existing ART algorithm and then maps each test case from the source domain into the so-called mirror test cases in the mirror domains. However, applying mirroring operators in high-dimensional input spaces is computationally intensive.

According to dynamic mirror adaptive random testing (DMART), the mirroring scheme should be dynamic instead of static to reduce the computation overhead of the MART algorithm (Huang et al. 2015). The main drawback of the MART and DMART methods is generating test cases with the same patterns, decreasing test diversity.

The ART (Huang et al. 2021; Chen et al. 2004), DART (Huang et al. 2020), MART (Chen et al. 2004), and DMART (Huang et al. 2015) approaches attempt to support the diversity of test data in the whole program, a given execution path, equal-sized partitions, and dynamic-sized partitions. None of these approaches pay any consideration for fault proneness. DART (Huang et al. 2020) is the only approach amongst the mentioned approaches that look after coincidentally correct executions by generating test data to cover a given execution path.

The Zhou et al. (2013) approach to test data generation determines failing inputs based on Bayesian probability. The Zhou et al. approach, one implementation of ART, utilizes Bayesian probability to estimate the likelihood of a test case revealing a fault. It maintains a probability distribution (initially uniform) over the input domain. As the test cases are executed, the observed fault-detection rate is used to update the probability distribution. Test cases with a higher likelihood of detecting faults are then given more chances of being selected for execution. Fault proneness has been considered the only criterion to evaluate the effectiveness of test cases in the approach proposed by Zhou et al. (2013).

2.2 Domain-oriented path testing

In the field of software testing, several methods have been developed to extract subdomains of program inputs that execute a specific path, which can then be utilized as advanced path testing techniques. These methods often involve the use of partitioning techniques in conjunction with automated random testing (ART) to enhance fault detection during path testing (Gotlieb and Petit 2006; Nikravan and Parsa 2019).

An early attempt to address the path domain extraction problem is the path random testing (PRT) method that generates test data by partitioning program inputs and sampling from the partitions. Gotlieb and Petit (2006) have proposed a PRT approach to generate a uniformly distributed sequence of random test data that executes a single control flow path in the program under test. The PRT method involves two main steps:

1. Derive the path conditions using backward symbolic execution and approximate the subdomain boundaries with constraint propagation and rejection.
2. Use a uniform random test data generator to extract points from the approximated subdomain.

However, one of the challenges with this method is that the constraint solver, often called during this process, can be computationally intensive and time-consuming to find accurate approximations of subdomains. The iterated partitioning PRT (IP-PRT) algorithm (Nikravan and Parsa 2019) reduces the number of constraint solver calls required by the PRT algorithm by iteratively splitting the input domain into subdomains and refuting those subdomains that are inconsistent with the path constraints. The remaining scaled-down subdomains provide all possible test data covering the desired path. The more accurate the input range, the more influential the test data (Nikravan and Parsa 2019). In path random testing approaches, the path constraints and constraint solvers are used to generate test data for the domain extraction of the path under test. The major problem with PRT methods is that they do not consider the changes made to the variables between the constraints (Gotlieb and Petit 2006; Nikravan and Parsa 2019). As a result, the boundary points of the domain may not be accurately determined in PRT.

Another family of domain extraction techniques uses heuristic methods. Huang et al. (2021) have proposed an approach for detecting faulty zones by finding boundary points. Their approach, called the search for boundary (SB), identifies additional failure-causing inputs that are as close to the boundary of the failure region of a numeric input domain as possible. SB starts with a failure-causing point in the input domain as a source or seed point. The source point is advanced in one direction with a size of L and α orientation. If the new point is in the desired region, then the process continues to explore the domain of the path. Otherwise, the search algorithm is backtracked with a size of $\frac{L}{2}$ and orientation of $-\alpha$. The authors have not compared their approach with existing works.

Marculescu and Feldt (2018) proposed an approach to identify the boundary between the valid and invalid regions of the input space by developing pairs of test sets that describe that boundary in detail. First, a test data generator, directed by a search algorithm to maximize the distance to known and valid test cases, generates test cases close to the boundary. Second, the valid test cases are changed randomly with small intervals to push them over the boundary and into the invalid part of the input space. The limited number of points cannot accurately determine the boundary of a specific path domain. Additionally, in cases where the actual domain consists of several areas with high dimensions, it is difficult to distinguish valid and invalid regions.

3 Proposed approach

This section describes our proposed approach to finding the most fault-prone path and generating test data in the subsets of input domains that execute this path. Our approach consists of a static analysis phase followed by a dynamic analysis phase, shown in Fig. 2. The input of the proposed method is the source code of the program under test (PUT), and the outputs are the fault-prone path along with the test data executing the path. In the static analysis phase, the control flow graph of the PUT is extracted, then the minimum cost test paths (MCTP) are generated to identify the fault-prone path. In addition, the PUT is instrumented to capture the required data during dynamic analysis.

The inputs of the static analysis phase are fed into the domain extraction module, which uses the Markov chain Monte Carlo method to generate test data, execute the fault-prone path, and approximate the subdomains of the input variables that correspond to this path. Our approach uses multiple Markov chains. Each chain as an agent cooperates with other agents in the search environment to find the approximation of the target domain. We discuss the details of each phase in the following sections.

3.1 Static analysis

The static analysis phase provides the artifacts required for our domain extraction approach. Extracting the domain of all execution paths in the program is a time-consuming and computationally intensive task, which may not be necessary since not all paths in the program have the same fault-proneness. Therefore, our approach focuses on extracting the domain for the most fault-prone path selected from the minimum cost test paths (MCTPs).

At first, the control flow graph of the PUT is extracted to obtain the prime paths. The advantage of prime paths is that they keep the number of test requirements down despite the execution paths, which often are infinite in the program with loops. According to Ammann and Offutt (2016), in a prime path, no node appears more than once with the path, and it does not appear as a proper subpath of any other simple path. However, the prime paths are not necessarily executable. On the other hand, an execution path may cover multiple prime paths. Therefore, it is beneficial to find a minimal set of execution paths that covers all prime paths to reduce the cost of path testing.

We use the brute force and set-covering methods proposed by Li et al. (2012) to obtain minimum-cost test paths (MCTPs). An example of a PUT and its control flow graph is shown in Fig. 3. Moreover, Table 1 shows the prime path extracted from the CFG in Fig. 3 and their corresponding MCTPs. It is observed that the Binomial method contains 13 prime paths, which can be covered through the six MCTPs. For instance, the MCTP in the first row of Table 1 covers four prime paths. We select the most fault-prone path from MCTPs with the algorithm described in Sect. 3.2.

The second task in the static analysis module is source code instrumentation, in which some codes are added to the PUT to specify the executed path and calculate the distance

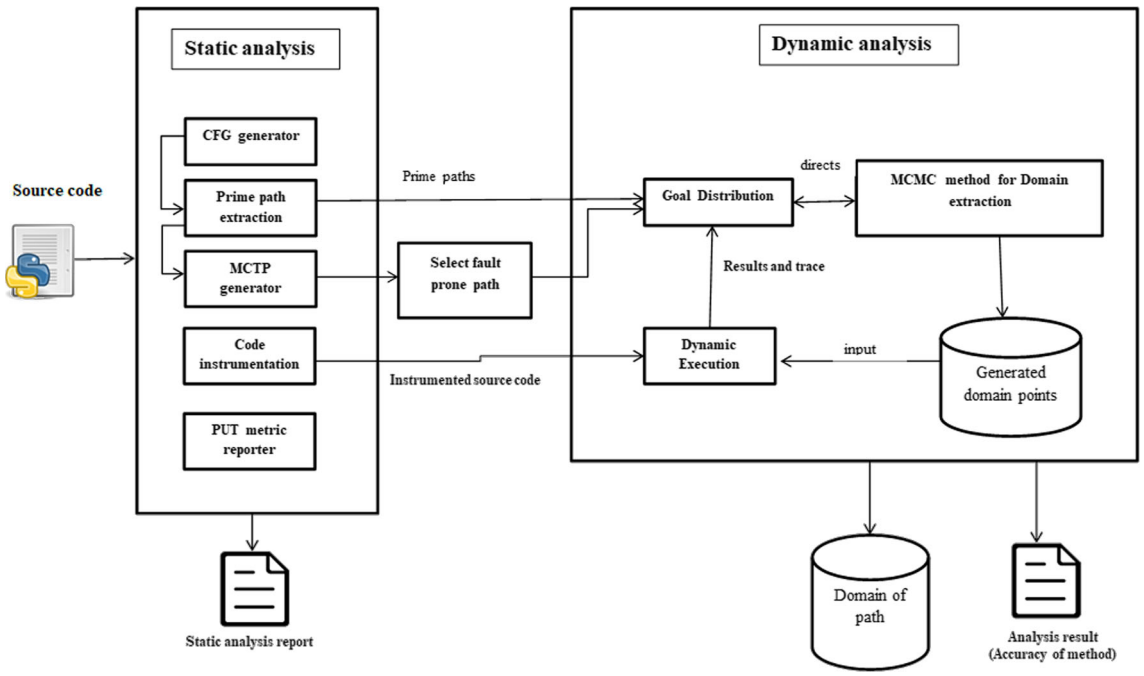

Fig. 2 Overview of the proposed domain extraction approach

Fig. 3 An example program and its CFG

```
0: def Binominal(n: int, k: int)
1:    if k > n:
2:       return 0
3:    result = 1
4:    if k > n - k:
5:       k = n - k
6:    i = 1
7:    while i <= k:
8:       result *= n
9:       result //= i
10:      n -= 1
11:      i += 1
12:   return result
```

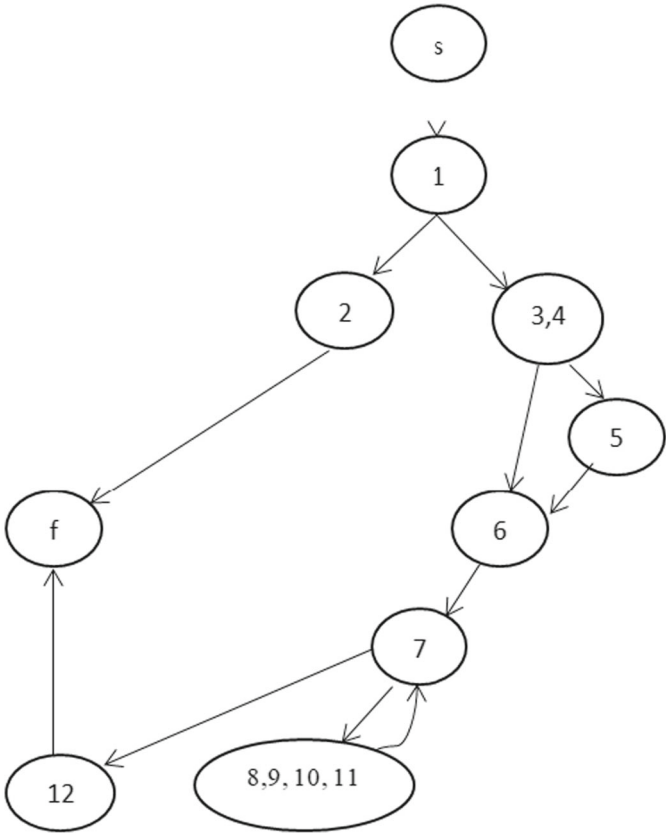

Table 1 Minimum cost test path and prime paths correspond to the CFG in Fig. 3

Minimum cost test path (MCTP)	Prime path
[s, 1, 3, 4, 6, 7, 8, 9, 10, 11, 7, 8, 9, 10, 11, 7, 12, f]	[s, 1, 3, 4, 6, 7, 8, 9, 10, 11], [8, 9, 10, 11, 7, 12, f], [7, 8, 9, 10, 11, 7], [8, 9, 10, 11, 7, 8, 9, 10, 11]
[s, 1, 2, f]	[s, 1, 2, f]
[s, 1, 3, 4, 6, 7, 12, f]	[s, 1, 3, 4, 6, 7, 12, f]
[s, 1, 3, 4, 5, 6, 7, 12, f]	[s, 1, 3, 4, 5, 6, 7, 12, f]
[s, 1, 3, 4, 6, 7, 8, 9, 10, 11, 7, 12, f]	[s, 1, 3, 4, 6, 7, 8, 9, 10, 11], [8, 9, 10, 11, 7, 12, f], [7, 8, 9, 10, 11, 7]
[s, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 7, 12, f]	[s, 1, 3, 4, 5, 6, 7, 8, 9, 10, 11], [8, 9, 10, 11, 7, 12, f], [7, 8, 9, 10, 11, 7]

of the executed path from the target MCTP. Instrumentation is used to measure code coverage, which determines which parts of the code are executed during program runs. Our proposed approach for domain extraction relies on utilizing the computed distance to assess the quality of the generated test data. The distance calculation is performed by injecting distance functions into some parts of the code, mainly condition statements. The distance functions are called when test data are executed on PUT to compute the distances based on relations in Table 2 of Sect. 3.6. The Python AST library for abstract syntax tree (AST) parsing has been used to instrument the code. It should be noted that all the tasks, as mentioned earlier, are performed statically without executing the PUT.

3.2 Selecting the fault-prone path

Our approach for selecting fault-prone paths uses a mutation-based technique applied to the statements in the path. Minor code modifications resulting from the mutation appear as abnormal behavior of an executed test (Papadakis et al. 2019). Mutation analysis creates several mutant variants of the program under test. Mutant operators are simple syntactic rules that establish the mutant's version of

Table 2 The branch distance for different types of predicates (Afzal et al. 2009)

ID	Predicate	Branch distance
1	Boolean	if True, then 0 else K
2	$a = b$	if $abs(a-b) = 0$ then 0 else $abs(a-b) + K$
3	$a \neq b$	if $abs(a-b) - 0$ then 0 else K
4	$a < b$	if $a-b<0$ then 0 else $(a-b) + K$
5	$a \leq b$	if $a-b \leq 0$ then 0 else $(a-b) + K$
6	$a > b$	if $b-a<0$ then 0 else $(b-a) + K$
7	$a \geq b$	if $b-a \leq 0$ then 0 else $(b-a) + K$
8	$a \vee b$	$\min(\text{cost}(a), \text{cost}(b))$
9	$a \wedge b$	$\text{cost}(a) + \text{cost}(b)$
10	$\neg a$	Negation propagated over a

programs by changing the program's syntax. Mutants measure the quality of tests by observing and comparing the runtime behavior of the initial and mutated versions of programs. The mutants that exhibit different results than expected ones are called the 'killed mutants.' Otherwise, the mutant is called 'live' (Zhu et al. 2018). Our scoring mechanism to score and rank fault-prone paths of the PUT contains two steps:

- Step 1: The fault proneness score of the mutations created for the PUT is calculated.
- Step 2: The fault proneness score of the program paths is calculated based on the score of the mutants applied to the path statements.

For the first step, we adapt the learning approach proposed by Titcheu Chekam et al. (2020) to estimate the fault proneness score of mutations. This model has been trained on a dataset containing static mutation features extracted from various faulty programs. A collection of faulty programs has been prepared to create the dataset. Then, different mutations of these programs are generated, and the mutation's features are extracted for each mutant. The features used to train the model include 25 different metrics extracted from the program CFG and AST, such as mutant node depth, mutant node type, data, and control dependencies. The completed list of metrics is available in Titcheu Chekam et al. (2020). Therefore, samples in the resultant dataset are the feature vectors extracted from each mutant. In this dataset, a so-called killability metric has been devised to label each sample, i.e., mutant. The killability of a mutation is defined as a ratio of the number of test data that kill a mutation to the total number of test data in a test suite. According to this definition, increasing the number of test data detecting artificial faults made by a mutation increases the mutant killability. As the killability of a mutation increases, the probability of existing fault in the mutation, i.e., the fault-proneness score of the mutant, also increases. After labeling samples with killability scores, a machine-learning model is trained to predict the killability of the mutants. Predicting mutation's killability prevents the need for executing the mutated program with test data and highly improves the performance of scoring mutant fault proneness. We created a dataset and trained Chekam's model for Python codes.

For the second step, we present an algorithm to rank the fault proneness of the program paths based on the fault proneness score of mutants estimated by our learned model. Our algorithm relies on the collective effect of mutants' scores of the statements on a given path. Each mutation applied to a statement indicates the possibility of a fault in that statement. Therefore, the average of the total fault proneness score in different mutations applied to the same statement measures the fault proneness score of that statement. Similarly, a set of consecutive statements expresses each path in the program, and the average fault proneness score of statements in the path denotes the path's fault proneness score.

Algorithm 1 describes our fault-prone path prioritization mechanism. It receives the PUT, MCTP, and the learned machine learning model as input and produces a list of MCTPs ranked by their fault proneness score. First, the program mutations are created, and the corresponding feature vector of each mutated version is extracted (Lines 2 to 6). The Python MutPy library (Hałas et al. 2022) is used to mutate Python programs. The extracted feature vectors are then fed to the learned model to predict the fault proneness of each mutated version (Line 7). Afterward, the fault proneness score of each program path is calculated by averaging the fault proneness scores of all statements in the path (Lines 8 to 18). Finally, all MCTPs are sorted according to their fault proneness scores, and we can select the candidate path for domain extraction (Line 19).

Algorithm 1: Fault-prone path ranking

```
Input:
- Program under test
- Minimum cost test paths (MCTP)
- Learned machine learning (ML) model
Output: Fault proneness priority of input paths
# Create mutations for PUT
Mutations ← Create mutation of input program with Python Mutpy;
For i ← 1: m do
    Mu_features ← Extract_Features(Mutation[i]);
    Add MU_features to the features list;
End For
# Step 1: assign a score for each mutation based on the machine-learning model
Mutation_Scores ← ML_model.predict(features);
# Step 2: assign a score to each path
For j ← 1: n do
    sum_of_score ← 0;
    For k ← 1: MCTP[j].length() do
        For m in set of mutant applied on statement(MCTP[j][k]) do
            sum_statement_score ← Mutation_Scores(MCTP[j][k][m]);
        End For;
        statement_score ← sum_statement_score / count of mutant applied on statement k;
        sum_path_score ← sum_path_score + statement_score;
    End For;
    set path_score to (sum_path_score / MCTP[i].length());
End For;
Sort MCTP based on path_score;
```

3.3 Markov chain Monte Carlo sampling

Markov chain Monte Carlo (MCMC) sampling is a class of algorithms for systematic random sampling from distributions with high-dimensional spaces. Despite the naive Monte Carlo sampling method, which independently samples from a target distribution, e.g., Gaussian distribution, the Markov chain Monte Carlo methods obtain a sample that depends on the previous sample. The Markov chain allows sampling algorithms to sample from spaces with many random variables (high-dimensional spaces) such that the desired values are obtained from the target distribution (Brooks et al. 2011). MCMC methods combine the concept of Monte Carlo sampling with the Markovian property.

Monte Carlo sampling is a way to estimate a constant parameter by generating random numbers iteratively. Considering generated random numbers and processing them, the Monte Carlo sampling estimates a parameter that its direct calculation might be impossible or time-consuming. In other words, the Monte Carlo sampling method obtains the target distribution of all answers to a given problem. However, Monte Carlo sampling may not be effective in some situations. Two fundamental reasons for the question "why the method of producing random Monte Carlo samples cannot be effective?" are as follows:

1. The number of variables in the input space might be large. Indeed, the high input dimensions may cause inefficiency.
2. The range of the variable at each dimension might be high, such that with a limited number of samples, it is impossible to cover the whole space.

In the Markov chain, a state depends only on its previous state, also called memoryless or Markovian property. A state in a Markov chain model uses a transition function to move to the next state. In the Markov chain Monte Carlo method, a certain proposal distribution defined by a proposal function would be used as a transition function. A new instance is generated using the defined transition

function at each sampling iteration. The next state is accepted or rejected using the problem constraints to achieve a target distribution function (Hogg and Foreman-Mackey 2018).

Methods for program path domain extraction often include an unknown target distribution that does not follow conventional distributions such as the normal or uniform distribution. The path domain distribution depends on the inputs of the path and the operators inside it, as well as the conditions that make up the path. In this paper, a well-known Markov chain Monte Carlo sampling method, the Metropolis algorithm, is used to identify the input domain of the target path in the program under test.

3.4 Metropolis algorithm

The Metropolis algorithm, described in this section, is a particular case of the so-called Metropolis–Hastings algorithm, in which the proposal distribution is symmetric. In addition, transitions between states are performed by the random walk in the proposal distribution around the current state. The Metropolis–Hastings algorithm is a variant of MCMC for obtaining a sequence of random samples from a probability distribution from which direct sampling is complex. The series of random samples is considered a chain. The Metropolis algorithm changes random variables during the transition from one state of the chain to its successive state. The target distribution is approximated as a set of answers based on samples from the proposal distribution by the Metropolis algorithm (Avrachenkov et al. 2018). The sampling algorithm continues to the number of states within the chain. The steps of the Metropolis–Hastings algorithm are as follows (Luengo et al. 2020):

Step 1: Random prototype variables are generated, and t is set to 1.

Step 2: The following steps are repeated until $t ==$ the number of states within the chain:

Step 2.1: Generate a new sample based on the previous samples and proposal distribution.

Step 2.2: Calculate the acceptance probability, $p_\alpha = \min\left(1, \frac{p(\theta^*)q(\theta^*;\theta)}{p(\theta)q(\theta;\theta^*)}\right)$.

Step 2.3: Generate a uniform random number, u, between zero and one $(0 < u < 1)$.

Step 2.4: If $p_\alpha \leq u$, then the answer is accepted, and the new state is equal to θ^*.

Step 2.5: If $p_\alpha > u$, then the answer is rejected, and the new state is equal to the previous state.

Step 2.6: $t \leftarrow t + 1$

In the simplest form of the Metropolis–Hastings algorithm, the target distribution is assumed to be symmetric (Avrachenkov et al. 2018). Therefore, $q(\theta^*;\theta) = q(\theta;\theta^*)$ in acceptance probability relation in Step 2.2, and it would be simplified to Eq. (1).

$$p_\alpha = \min\left(1, \frac{p(\theta^*)}{p(\theta)}\right) \tag{1}$$

Symmetrical distributions such as normal and uniform distributions make it easy to calculate random walks based on the current point while simplifying the calculation of the acceptance probability (Mattingly et al. 2012). It is worth noting that if θ^* is less probable than θ, i.e., $p(\theta^*) < p(\theta)$, the samples are moved in a direction with a high-density distribution. On the other hand, the current state is not changed until a better sample is found. In its current form, the Metropolis algorithm suffers from the trade-off between exploration and exploitation (Robert et al. 2018), which means that it cannot simultaneously discover different regions of the answers while focusing on specific regions. Our adaption of the Metropolis algorithm to domain extraction addresses this problem using a multi-agent scheme and scoring mechanism described in the next section.

3.5 Adapting the MCMC method for domain extraction

The MCMC method has already been used in reliability assessment to search specific regions by subset simulation (Song et al. 2021; Abdollahi et al. 2021). We propose an adaption of this algorithm for testing the reliability of the most fault-prone path in a software program by sampling the test data from the path's domain. Our proposed domain extraction approach uses the Metropolis–Hastings method with a normal distribution as the proposal distribution and random walk sampling strategy. In our application, the target distribution function is the distance from the most fault-prone path, indicating the probability of executing the path with the generated test data. The shorter the distance, the more likely it is to be inside the path domain. Therefore, the lower the distance from the previously executed test data, the higher the probability of the test data covering the path. Indeed, the program statements belonging to the path domain form the target distribution function of the Metropolis sampling algorithm. We get close to the execution of a path as the distance measured by the target distribution function decreases. As a result of finding data that executes the target path, the subsequent samples are selected from the dense areas close to the current point using the Metropolis algorithm.

A major challenge is that the Metropolis algorithm fails to find all regions of a domain in cases where the path domain consists of multiple dense regions. The sampling algorithm is stuck on one region in such cases. Therefore, one region is sampled until the termination condition is

met. We have developed a multi-agent sampling approach to address this challenge. Several agents use the Metropolis algorithm to sample the search space at each iteration in the proposed method. Each agent represents an independent Markov chain. First, agents are created randomly in the search space according to the number of agents defined by the test engineer. Each agent then begins sampling independently regarding the length assigned to its chain.

Agents cooperate to find the domain of a given path. A scoring mechanism has been devised for constructive cooperation between agents during the sampling process. If an agent generates a sample that executes the given path, the agent is awarded by adding a constant to its score. After a certain period, the agent with the highest score pulls the one with the lowest score to its search region. The number of agents that an agent can pull is limited to prevent the search process from biasing toward a specific region. The proposed multi-agent approach provides two advantages: first, it provides search diversity in different regions of the answer space. Second, answer regions are more investigated than non-answer regions with the cooperation between agents.

3.6 Domain extraction algorithm

We describe our adaption of the MCMC domain extraction algorithm to find the test data executing the most fault-prone path of the PUT in detail. The proposed domain extraction algorithm uses multiple Markov chains by incorporating multiple agents cooperating to find the answer regions. As described in Sect. 3.4, we need to define a target distribution for sampling to generate the test data from regions of the input domain. The target distribution in the Monte Carlo Markov chain Metropolis algorithm is the probability distribution that we want to sample from, but we do not know its exact form, or it is difficult to sample from directly. The Metropolis algorithm simulates a Markov chain with the target distribution as its stationary distribution, meaning that the chain will converge to the target distribution after many iterations. The algorithm uses a proposal distribution that is easy to sample from and a ratio that determines the acceptance or rejection of each proposed sample. The ratio depends on the target distribution and the proposal distribution, and it ensures that the Markov chain satisfies the detailed balance condition, which is necessary for the convergence to the target distribution.

The target distribution for the search space is defined by Eq. 2, indicating the closeness of the executed path to the target path and the number of covered prime paths in the MCTP (Monemi Bidgoli and Haghighi 2020). In Eq. 2, NC is the number of covered prime paths of the fault-prone path, FPP, by a given test data, TD, and APP represents the number of all prime paths in FPP. SEP is the test data distance from the predicates in the fault-prone path, calculated according to Table 2. The value of the target distribution defined by Eq. 2 is between zero and one. Any test data that minimize Eq. 2 is more likely to execute the target path.

$$\begin{aligned} &\text{targetdistribution}(\text{TD}, \text{FPP}) \\ &= \frac{1}{2} \times \left(\left(1 - \frac{\text{NC}(\text{TD})}{\text{APP}(\text{FPP})} \right) + \frac{\text{SEP}}{\text{SEP} + 1} \right) \end{aligned} \tag{2}$$

The number of covered prime paths in the MCTP is an important metric that indicates how well the MCTP can exercise the program under test. A prime path is a simple path that is not a proper subpath of any other simple path in the program. A simple path is a path that does not contain any repeated nodes or edges. The MCTP is a set of test paths that covers all the edges and nodes in the program with the minimum cost, where the cost is defined as the sum of the execution frequencies of the edges in the test paths. The more prime paths are covered by the MCTP, the more likely it is to reveal faults in the program. Therefore, we aim to generate test data that can execute as many prime paths as possible in the MCTP.

Algorithm 2 shows our proposed MCMC domain extraction approach. The algorithm receives the most fault-prone path of the PUT, the PUT, and the test budget as inputs. It returns a set of test data that executes the fault-prone path, approximating the path's domain as the output. The details of the algorithm are described in the rest of this section.

All agents are initialized randomly in the search space at the beginning of the algorithm (Line 3). The main operations of the algorithm, *i.e.*, generating test data and updating the agents' states, are repeated for all agents according to the test budget, Tr (Lines 5 to 22). At each iteration, t, the test data, $\text{Agent}[i] \cdot \theta^*$(a sample in the search space) is generated from the normal distribution around the current state, $\text{Agent}[i] \cdot \theta[t-1]$, to create the random walk search with step, σ, for the agent (Line 8).

To find how the generated test data are appropriate, the ratio of the target distribution value of the current state, *i.e.*, the generated test data and the previous one, is computed based on Eq. 2. The minimum of this value and one is then set as acceptance probability, p_α (Line 9).

In the next step, the number, u, between zero and one, is randomly selected for the uniform distribution (Line 10). Then, if the condition $p_\alpha < u$ is satisfied, the generated test data, $\text{Agent}[i].\theta^*$, is set as the new state of the Markov chain (Lines 11 to 14). The generated test data, $\text{Agent}[i].\theta^*$, is also added as valid test data to the test suite. Otherwise, the state of the Markov chain is not changed (Lines 16 and 17). The scoring mechanism for cooperating and directing agents towards dense regions is applied in Lines 13 and 17.

Algorithm 2: MCMC domain extraction

```
1:  Input:
    - Fault-prone path (FPP)
    - Program under test (PUT)
    - Test budget (TB)
    Output: A set of test data in the domain of the fault-prone path
    (TestSuite)
2:  TestSuite ← { };
3:  InitializeAgents();
4:  t = 0;
5:  While t + 1 < TB do:
6:      t = t + 1;
7:      For i=1 to N:
8:          Agent[i].θ* ← normal distribution (Agent[i].θ[t− 1], σ);
9:          p_α ← min(1, p(Agent[i].θ*, FPP) / p(Agent[i].θ[t−1], FPP));
10:         u ← uniform distribution (0, 1);
11:         If p_α ≤ u do:
12:             Agent[i].θ[t] ← Agent[i].θ*;
13:             Agent[i].score ← Agent[i].score +1;
14:             TestSuite ← TestSuite ∪ Agent[i].θ[t]
15:         Else:
16:             Agent[i].θ[t] ← Agent[i].θ[t− 1];
17:             Agent[i].score ← Agent[i].score + 0.5;
18:     End For;
19:     If t % Pr == 0 do:
20:         UpdateAgents ();
21:     End If;
22: End While;
```

```
1:  Procedure 1: UpdateAgents
2:  Find an agent with a maximum and minimum score as Maxagent, Minagent;
3:  If Maxagent.threshold < Tr do:
4:      Minagent.θ[t] ← Maxagent.θ[t];
5:  Else:
6:      Set the agent with the second maximum score as Maxagent;
7:  End If;
```

In the scoring mechanism, agents who continue to search in the right direction would be highly scored. A low score is also given to the agents with weak functionality to maintain the interval between the agents and avoid biasing all agents to one search region. As a result, the search process would be expanded in all directions of the search spaces.

The cooperation between agents is performed in pre-defined periods such that agents with low scores have a chance to explore their neighborhood spaces thoroughly (Lines 19 and 20). Procedure 1 describes the details of the scoring mechanism. The agent with the highest score pulls the agent with the lowest score to their area. On the other hand, a high-score agent would direct a limited number of

Table 3 The parameters and notations used in the proposed MCMC algorithm

Parameter or notation	Description
TB	Test budget that means the length of each chain
N	Number of agents or chains
Tr	An agent could conduct a number of agents to move to a new area
Pr	Number of iterations wait to update agent positions
Agent[i]	An object that acts such as a chain
Agent[i].θ^*	Test data generated from normal distribution randomly
Agent[i].θ	List of test data assigned to the chain at previous iterations
Agent[i].score	The score for conducting other agents to a dense area
Agent[i].threshold	Agents conducted by agent[i] to change the search area
p	Target distribution value for test data that shows distance from goal path
σ	Step size for random walk approach in the normal distribution

the low-score agents to the region specified by the threshold parameter. All the parameters used in this algorithm are described in Table 3.

4 Evaluation

In this section, various experiments are performed to answer four research questions regarding the effectiveness and applicability of the proposed path domain extraction approach, MCMC-DOM. Python programming language was used to implement the proposed approach, related work algorithms, and required simulations. We compare our approach with state-of-the-art domain extraction and test data generation approaches regarding various evaluation metrics, including accuracy, boundary coverage, and mutation score. First, we describe our research questions, evaluation metrics, and experimental setup. Then, we discuss the results of our findings.

4.1 Research questions

We conducted our experiment to answer the following research questions:

- RQ1. *How accurate are the test data generated for covering the most fault-prone path?*
- RQ2. *What is the path complexity's impact on the accuracy of the extracted domain?*
- RQ3. *How many of the boundary points of the target domain are covered by the proposed domain extraction approach?*
- RQ4. *What is the fault detection capability of the selected test data from the extracted domain by our approach?*

The following strategies are used to answer the research questions:

Strategy to answer RQ1: First, the path domain is obtained by running different domain extraction tools on given programs. Then, the ratio of test data executing the fault-prone path to all test data generated by the domain extraction tool is calculated and reported as the tool's accuracy.

Strategy to answer RQ2: First, we compute two complexity metrics, described in Sect. 4.3.2, for each fault-prone path. Then, the accuracy of extracted domains is analyzed regarding the values of these complexity metrics.

Strategy to answer RQ3: First, the boundary points of the actual domain are obtained according to the presented method by Huang et al. (2021) for computing boundary points. Then, the overlaps of these points with the points obtained by the proposed domain extraction approach are computed as a new boundary coverage metric. The results are also compared with existing domain extraction approaches.

Strategy to answer RQ4: The effectiveness of generated test data by our tool corresponds to the extracted domain of the most fault-prone path examined using mutation analysis. The mutation score is computed and reported, indicating the fault detection capability of test data. We also compare the results with test data generated by regular test data generation approaches, where the goal is to cover all branches or paths with a minimum number of test data. In this way, we examine our hypothesis that some faults in a

path are detected only by testing the path with numerous test data generated from the path's domain.

4.2 Evaluation metrics

Accuracy It is necessary to check how many of the generated test data execute the target path to evaluate the performance of the proposed domain extraction method. The number of points executing the target path relative to the total points generated by the domain extraction algorithm is defined as the accuracy of the method.

Boundary coverage Most program faults occur at the boundary points of the path's domain (Dobslaw et al. 2020). Therefore, it is necessary to check how many of the generated domain points execute the boundary of the target path. To this aim, the overlap of the points generated by the domain extraction approach and the boundary points obtained by the Huang et al. (2021) approach is computed. Due to floating point precision, the pair of boundary points may not be entirely equal. Therefore, we consider an acceptable error margin, Δ, when computing the overlap of each pair of points. A generated test data are considered a boundary point if the Euclidean distance between test data and an actual boundary point is less than Δ.

Mutation score The mutation analysis technique obtains the mutation score by generating artificial faults in the program under test and executing the mutated versions with the test data. The output of mutated versions is then compared with the expected output to determine the mutation score. The mutant survives if the expected result is achieved at the output. Otherwise, the mutant is called killed. The mutation score is computed by Eq. (3). The higher values of this score mean fault detection ability are more appropriate in the mutant programs. In other words, the faults in the program are detected accurately, indicating the effectiveness of generated test data according to their fault detection capability.

$$\text{Mutation score} = \frac{\text{number of the killed mutant}}{\text{total number of mutant (killed and survived)}} \times 100 \tag{3}$$

4.3 Experiments setup

4.3.1 Studied programs

We selected ten software programs with various input parameters and internal structures for our experiments. Eight of these are relatively small Python programs, while two are real-world programs with non-trivial structure and complexity. Table 4 shows the selected programs, functionalities, and some relevant information for each program. The #Units column indicates the number of functions in the program source code. The #Input column denotes the number of inputs of the PUT. The #PP and #MCTP columns show the number of prime paths and minimum cost test paths in each PUT. We used two source code metrics to measure the path complexity: (1) path cyclomatic complexity (PCYC) and (2) path clause complexity (PCC). The former indicates the structural complexity of the path, while the latter indicates the arithmetic complexity. The value of these metrics for the most fault-prone path is shown in Table 4.

Cyclomatic complexity is used to determine the complexity of a method. It is defined as the number of independent control flow graph paths. If the number of nodes and edges for CFG of the function under the test are respectively represented by n and e, the cyclomatic complexity for a function equals $e - n + 2$ (Ebert et al. 2016). We used the same definition at the path level to compute the path cyclomatic complexity (PCYC) for the most fault-prone path. The PCYC metric only considers the number of nodes and edges within the target path instead of the entire CFG.

The clause complexity (CC) metric is used to determine the difficulty degree of the operators in the program clauses. Mao (2014) has mentioned that the operator dependency of a clause must satisfy the difficulty degree. The clause has been classified into four groups based on the common programming operators, including equality ($==$), non-equality ($!=$), inequality ($\leq$, $\geq$, $<$, and $>$), and Boolean (AND, OR, XOR, XNOR) operators. The list of complexity degrees for each programming operator is shown in Table 5.

The summation of all complexity degrees existing in a program determines the clause complexity of the program. We use the same definition at the path level to compute the path clause complexity (PCC) for the most fault-prone path.

4.3.2 Parameters' settings

The proposed algorithm contains several configurable parameters, specifically, chain length, the number of search agents, and the number of iterations that could be set to different values in experiments. In addition, the period of updating agents' status and the maximum step distance in random walking could be varied. The programs in our experiments, listed in Table 4, consist of inputs with

Table 4 Selected case studies for experiments

#Prog	Program name	Description	#Units	#Inputs	#PP	#MCTP	PCYC	PCC
P1	Armstrong-numbers	Find numbers that are equal to the sum of cubes of its digits between zero to integer input	1	2	13	4	3	4.2
P2	bisection	Approximation method to find the roots of the given function	1	2	21	9	5	4.8
P3	Decimaltoany	Convert decimal number to given any base	1	2	31	15	10	11.5
P4	reminder	The remainder of the first argument is divided by the second argument	3	2	49	29	14	7.8
P5	expint	Exponential integral function	3	2	48	25	16	7.9
P6	gammaq	Gamma Function	3	2	67	10	13	8.3
P7	gcd	Greatest divisor of two number	1	2	11	6	6	2.4
P8	CheckTriangle	Find the type of triangle	1	3	3	3	3	4.5
P9	Scipy	Open-source Python library used for scientific and statistical computing	25	2, 3, 4, 5	437	290	107	367.6
P10	mpmath	Open-source Python library for complex floating-point arithmetics	30	2, 3, 4	653	383	131	433.2

Table 5 Complexity degree of clause operator (Mao 2014)

No	Operator	Weight
1	==	0.9
2	! =	0.2
3	$\leq, \geq, <, >$	0.6
4	Boolean operators	0.5

different dimensions and ranges. We set the parameters of our proposed MCMC algorithm according to the input dimensions and the range of each input for the program under test. The values of configurable parameters for each PUT in our proposed approach are shown in Table 6. Depending on the complexity of the PCYC and PCC, parameters are assigned for each PUT. Accordingly, the more complex a program is, the more agents and repetitions of Markov chains it will take into account. However, due to the complexity of the paths, larger random steps are considered, and the updates are made in shorter time intervals so that the agents can approach the areas where the paths are executed.

Due to the stochastic nature of the proposed approach, the results may vary each time the algorithm is executed on a program under test. We repeated all experiments 30 times and averaged the results to ensure the reliability of the results and minimize the randomness effects. Indeed, the reliability of results is improved by increasing the number of experiments, such as the number of iterations or runs.

Table 6 Different values for configurable parameters have been used in the experiments

Parameter	Values	Program under test
Chain length or test budget (TB)	100	P1, P2, P7, P8
	150	P3, P4, P5, P6
	200	P9, P10
Number of agents (N)	40	P1, P2, P7, P8
	50	P3, P4, P5, P6
	60	P9, P10
Sigma (σ) for random walk steps	2	P1, P2, P7, P8
	4	P3, P4, P5, P6, P9
	5	P10
Update period for agents' status (Pr)	10	P3, P5, P9, P10
	15	P1, P2, P4, P6, P7, P8

The following are two main characteristics of repeating executions:

- The MCMC algorithm minimizes the impact of randomness by incorporating multiple sources of randomness, such as the initial state and random walk. A more stable estimate of the algorithm's performance can be obtained by running the algorithm multiple times to reduce the influence of random fluctuations. As a result, the search process is less likely to be misled by lucky or unlucky initial populations or random variation.
- The MCMC algorithm can be run multiple times to collect statistical data on its performance. Analyzing these data across multiple runs makes it possible to assess the statistical significance of the observed results. This is especially important when comparing different algorithm variations or evaluating an algorithm's performance against a baseline.

4.3.3 Experiment environment

Both the quality and quantity of generated test data with automatic test data generation algorithms are highly affected by the processing power of the machine used for test data generation. The experiment environment should be controlled to achieve reliable results and fair comparisons. We performed all experiments on a system containing a Core® i7™ 8550 CPU with a 1.8 GHz processing rate, 12 GB of RAM, and a Debian Linux operating system have been performed. The system was continuously monitored to ensure no other resource-consuming process was executed during the tests to maintain consistency in the results.

4.4 Results

4.4.1 Results for RQ1: method accuracy

In the first experiment, we compared our proposed domain extraction approach (MCMC-DOM) with PRT (Gotlieb and Petit 2006), IP-PRT (Nikravan and Parsa 2019), DART (Huang et al. 2020), DMART (Huang et al. 2015) regarding the accuracy of the extracted domain. Due to the static partitioning technique, PRT and IP-PRT do not directly generate test data. Indeed, the output of these approaches is the union of intervals. Therefore, we randomly selected test data from the generated interval according to the sampling effort used in the MCMC-DOM. The sampling effort is computed by multiplying the test budget by the number of agents. In this way, we ensured the fairness of our comparison. The performance of the PRT and IP-PRT methods in finding the points that can execute the path correctly is close to the proposed method because these two methods consider the entire path. On the other hand, due to the high accuracy of the division and conquer methods, the points of the space that execute the path are discovered correctly, and the main problem of these methods is in determining the boundary points of the space, which are less accurate because they do not determine them accurately. However, the DART (Huang et al. 2020) DMART (Huang et al. 2015) methods find data points with the approach of the distance from the path. As a result of the random search and lack of intelligence, some attempts using this method fail to execute the path. As a result of being more intelligent and better at detecting boundary points and utilizing the distance from the path, the proposed method has achieved greater accuracy.

Figure 4 shows the extracted domains' accuracy by the proposed method, MCMC-DOM, and the PRT, IP-PRT, DMART, and DART approaches for the studied programs. It is observed that the accuracy of the samples selected by MCMC-DOM is more than both the PRT and IP-PRT approaches in all studied programs. Figure 5 shows the average accuracy for each domain extraction approach in ten experiments corresponding to all studied programs. The average accuracy of our approach is close to 85%, revealing a 14% improvement compared to the IP-PRT method.

The input dimension of a PUT is an important factor that affects the accuracy. The high dimensional inputs make the partitioning difficult, resulting in sparsity and poor accuracy of the detected regions. However, the proposed approach achieves high accuracy due to the Metropolis algorithm's effective sampling and the sampling's dynamicity. We conclude that the MCMC-DOM performs better than the static partitioning-based approaches, such as PRT and IP-PRT, in finding the accurate domain of a path under test.

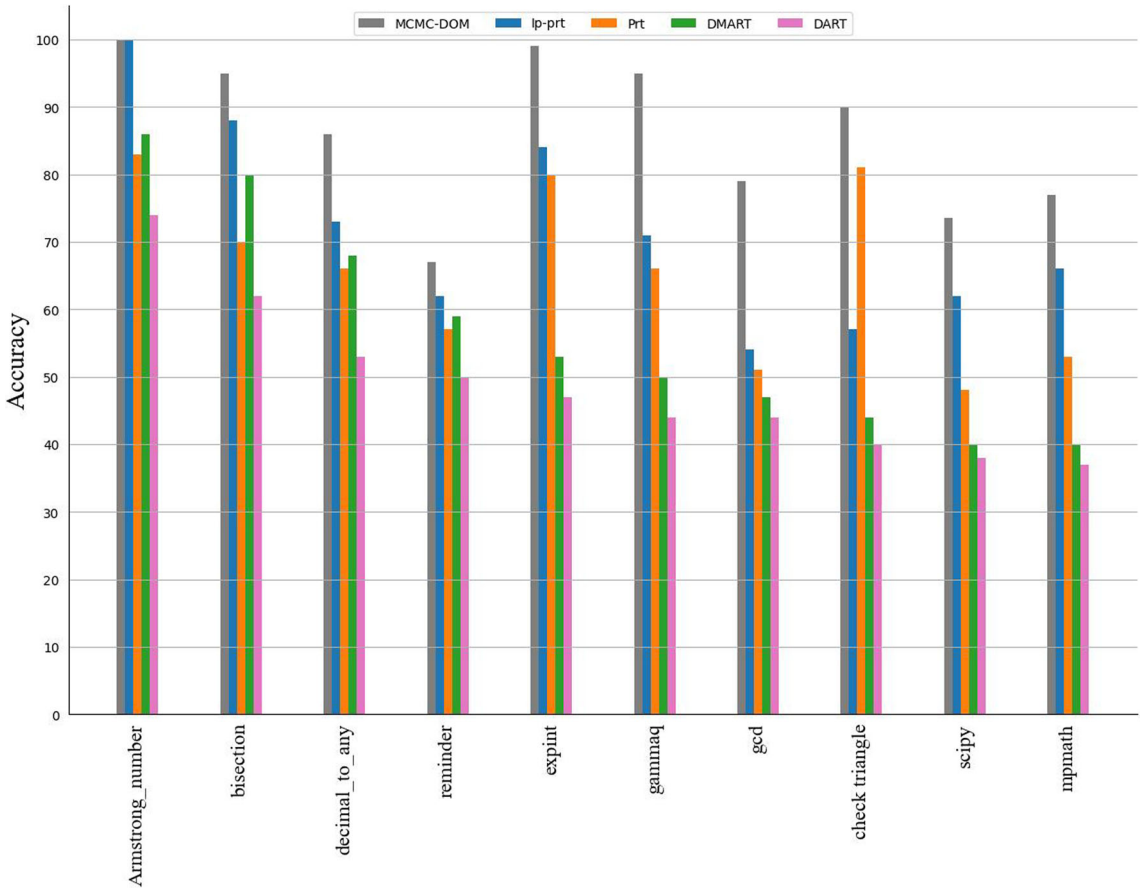

Fig. 4 The accuracy of different domain extraction approaches on each method per case study

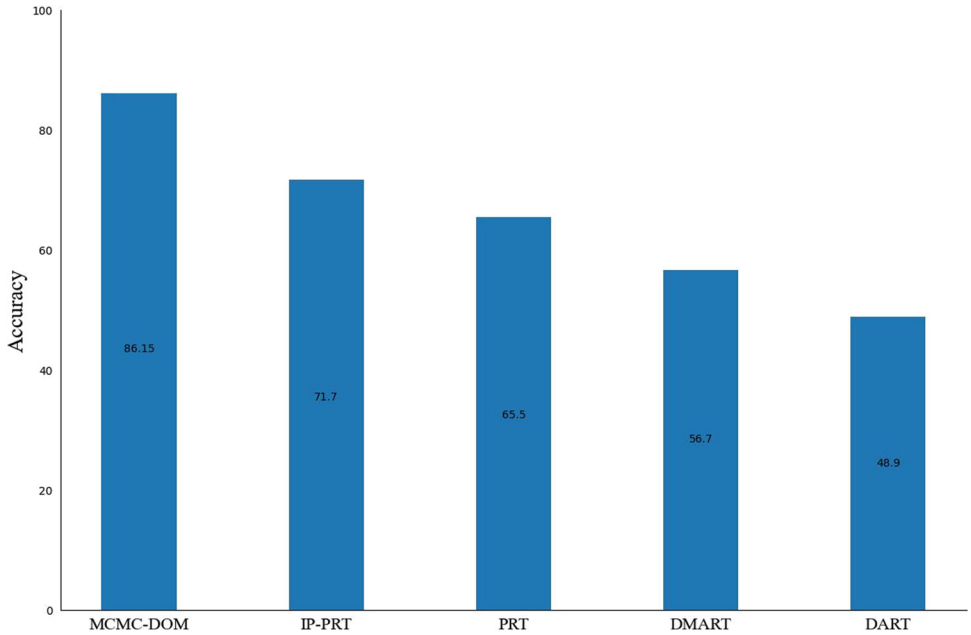

Fig. 5 The average accuracy of different domain extraction approaches on all case studies

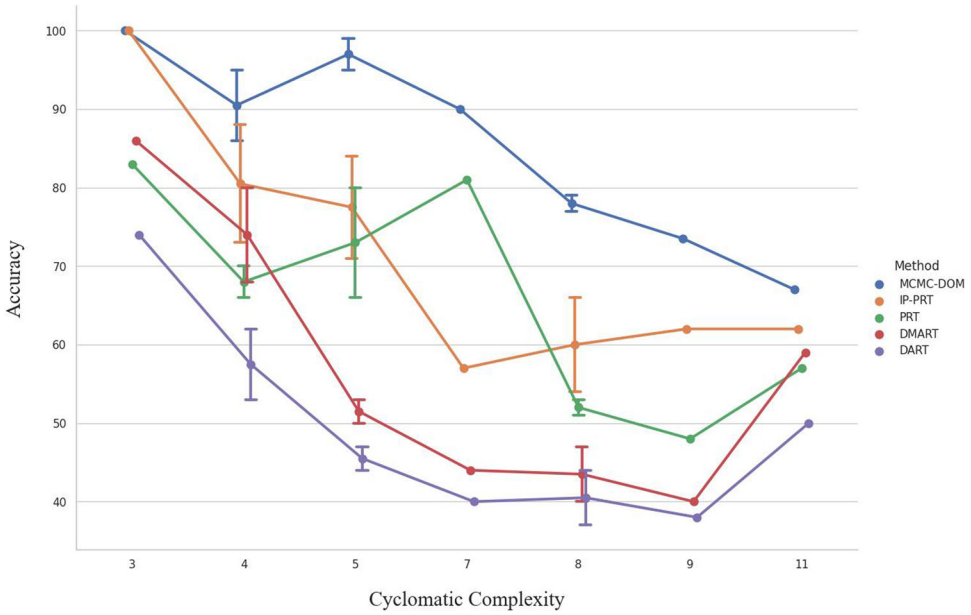

Fig. 6 Accuracy of extracted domains with different approaches versus the cyclomatic complexity

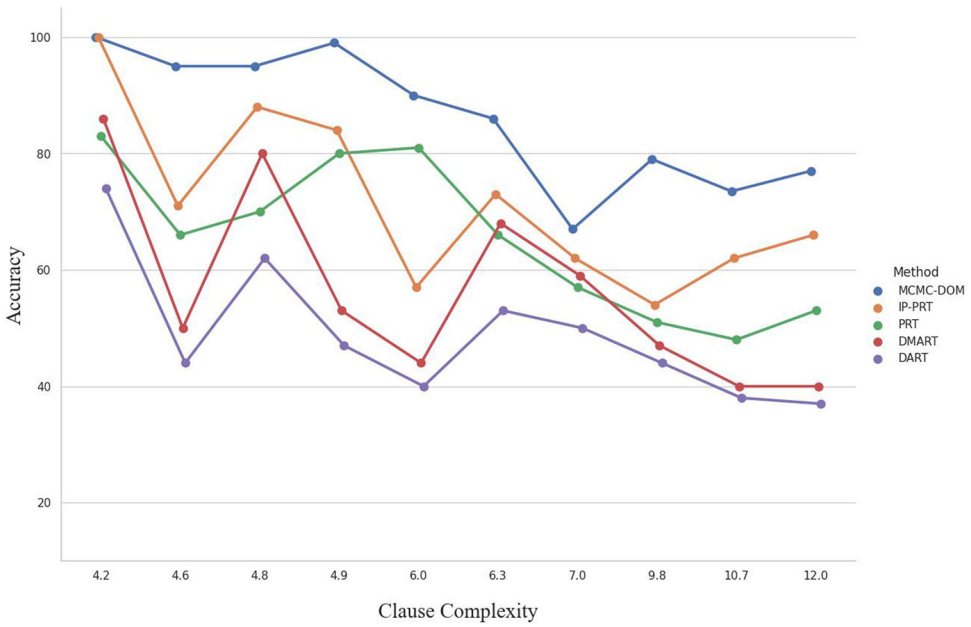

Fig. 7 Accuracy of extracted domains with different approaches versus clause complexity

RQ1. *How accurate are the test data generated for covering the most fault-prone path?*
Summary for RQ1. *Our proposed MCMC-DOM approach archives an average accuracy of 85%, which is the highest accuracy among two domain extraction approaches, PRT and IP-PRT. Results show a 14% improvement in accuracy compared to the IP-PRT approach.*

4.4.2 Results for RQ2: path complexity analysis

In RQ2, we study the impacts of path complexity on the performance of our proposed domain extraction approach. Figures. 6 and 7 and show the accuracy of the extracted domains of the studied programs with different approaches per cyclomatic and closure complexity metrics. It is observed that the overall accuracy of all domain extraction approaches decreases as the program's complexity increases. Nevertheless, the accuracy of our proposed MCMC-DOM approach is superior to the PRT (Gotlieb and Petit 2006), IP-PRT (Nikravan and Parsa 2019), DART (Huang

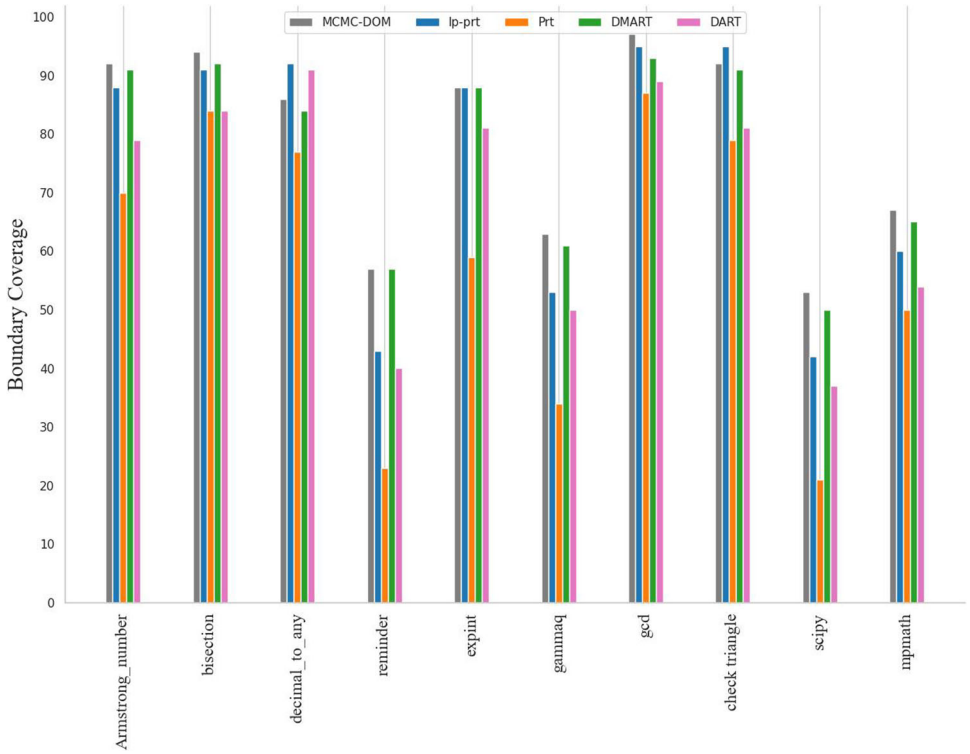

Fig. 8 The boundary coverage is achieved by each method per case study

et al. 2020), and DMART (Huang et al. 2015) approaches. Our approach is highly effective for programs with high cyclomatic and closure complexities. The fluctuations in the accuracy values observed in Figs. 6 and 7 and are primarily due to the input dimensions and the search space size in different cases. We conclude that the complexity of a path, the input dimensions, and the range of each dimension are the three main factors that affect the accuracy of domain extraction approaches.

boundary coverage metric discussed in Sect. 4.2. Figure 8 shows the boundary coverage of our domain extraction approach, MCMC-DOM, along with PRT (Gotlieb and Petit 2006) and IP-PRT (Nikravan and Parsa 2019). It is observed that the performance of MCMC-DOM in covering boundary points of the path domain is higher than the PRT and IP-PRT approaches in most studied programs. Moreover, it is observed that in only one of the studied programs, CheckTriangle, IP-PRT has performed better than MCMC-DOM in detecting the boundary points. The evaluated path in the CheckTriangle program consists of simple constraints with low complexities such that IP-PRT could accurately find its boundary points. However, as the complexity of the fault-prone path increases, the boundary coverage obtained by MCMC-DOM tends to be greater than that obtained by IP-PRT.

RQ2. *What is the path complexity's impact on the accuracy of the extracted domain?*

Summary for RQ2. *Our experiments reveal that increasing the program's complexity (both clause and cyclomatic complexities) decreases the accuracy of domain extraction approaches. However, the decreasing rate of MCMC-DOM accuracy is lower than PRT and IP-PRT accuracy.*

4.4.3 Results for RQ3: boundary points coverage

To answer RQ3, the coverage of the boundary points of the domain extraction approaches in previous experiments is compared to the fixed-orientation search for boundary (FSB) method introduced by Huang et al. (2021). The FSB generates the boundary data to discover the fault zone. As discussed in the related work, it receives a point that causes a fault as the initial seed, and the boundary points are identified based on this point. We executed the FSB method on our studied programs and computed the

RQ3. *How many of the boundary points of the target domain are covered by the proposed domain extraction approach?*

Summary for RQ3: *The average boundary points' coverage of the proposed MCMC-DOM approach is about 78.9%. It covers more boundary points than PRT and IP-PRT domain extraction approaches.*

4.4.4 Results for RQ4: fault detection capability

Our experiments in this section aim to compare the actual fault detection capability of the proposed approach with the state-of-the-art test data generation approaches that do not consider the path domain directly. To this aim, the fault detection rate of the generated test data by different test data generation approaches is investigated using mutation analysis. A test data generation approach targeting prime path coverage along with an approach maximizing branch coverage is compared with our MCMC domain extraction approach.

ACO-AR-SP approach (Monemi Bidgoli and Haghighi 2020) has been selected for the prime path coverage, which is the best method to cover the prime paths to the best of our knowledge. Likewise, the Pynguin tool (Lukasczyk et al. 2020) is used as a test data generation tool that aims to maximize the branch coverage of the program under test. Pynguin is an implementation of EvoSuite (Fraser and Arcuri 2013), the well-known test data generation tool for the Java programming language, for testing Python codes. It should be noted that only data that execute the target path is selected from the test data generated by ACO-AR-SP to be used in our experiment. For the Pynguin tool, the test data that execute the branches of the target path are selected. This way, we can compare path and branch coverage testing approaches with the domain testing approaches.

Figure 9 demonstrates the mutation score obtained by branch, path, and domain testing approaches for studied programs. It is observed that the mutation score of our domain extraction approach is higher than ACO-AR-SP (Monemi Bidgoli and Haghighi 2020) and Pynguin (Lukasczyk et al. 2020). As discussed earlier, the reason is that not all faults in a path are revealed by executing the path one time or with a small number of test data. For instance, a statement in a path that computes a division only fails when the divisor is set to zero. A higher mutation score means a higher fault detection capability.

To minimize the randomness and ensure the reliability of our results, we performed the test data generation and

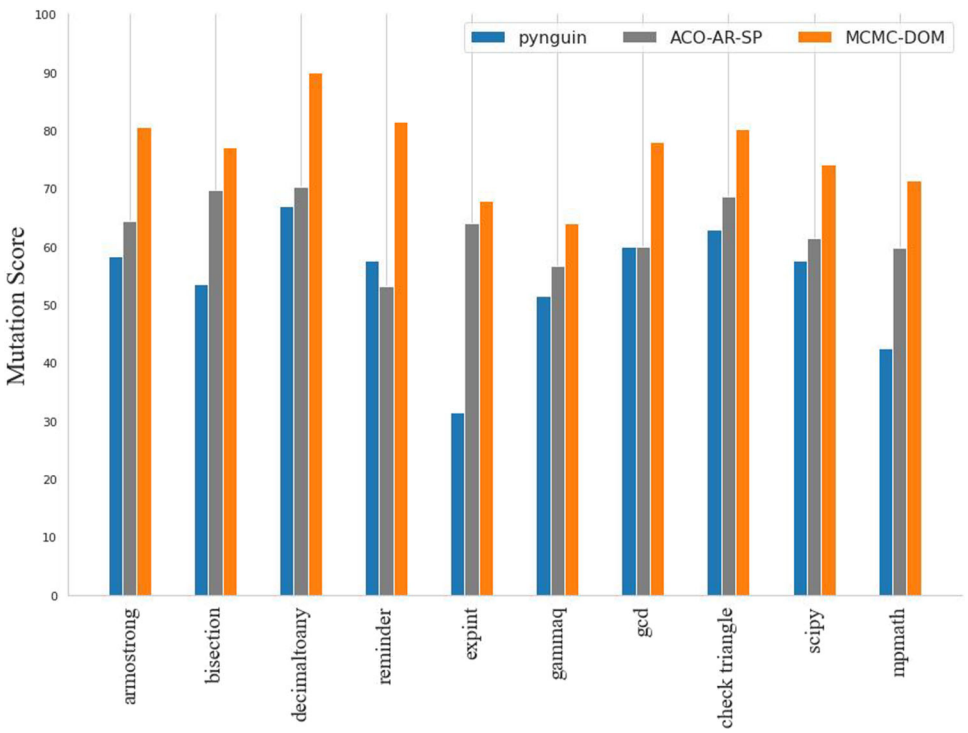

Fig. 9 Mutation score obtained by branch, path, and domain testing approaches for studied programs

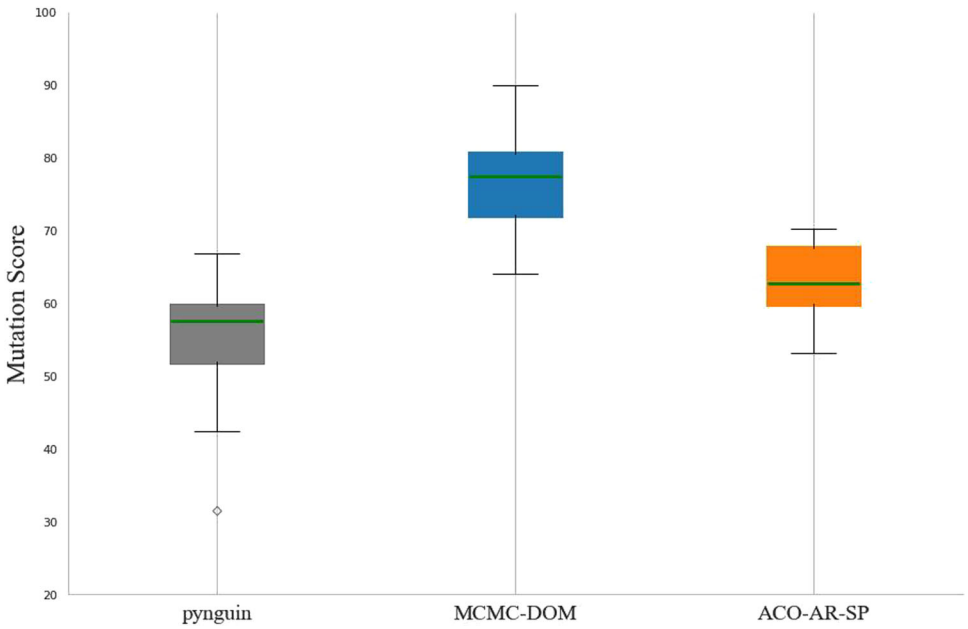

Fig. 10 The mutation score for the domain extraction, prime paths, and branch coverage methods

Table 7 Mutation score for the methods examined

Program name	Domain testing (MCMC-DOM)	Path testing (ACO-AR-SP)	Branch testing (Pynguin)
Armstrong-numbers	80.65 (92.21, 72.05)	64.42 (70.36, 57.22)	58.28 (34.69, 56.37)
bisection	77.00 (83.34, 74.02)	69.66 (72.30, 63.48)	53.50 (72.30, 63.48)
Decimaltoany	89.90 (93.60, 83.36)	70.24 (68.98, 71.06)	66.90 (61.38, 76.79)
reminder	81.50 (86.66, 68.06)	53.18 (50.30, 55.98)	57.63 (53.64, 63.68)
expint	67.8 (64.34, 76.60)	63.96 (57.31, 66.38)	31.50 (22.12, 38.34)
gammaq	64.02 (59.33, 72.83)	56.56 (53.87, 61.28)	51.41 (44.94, 54.82)
gcd	78.00 (86.74, 73.48)	60.00 (57.84, 64.19)	60.00 (53.45, 62.54)
CheckTriangle	80.12 (86.49, 74.19)	68.66 (73.30, 62.74)	62.95 (57.97, 64.49)
Scipy	74.09 (77.81, 71.33)	61.39 (55.67, 63.79)	57.62 (53.96, 59.94)
mpmath	71.43 (74.59, 63.73)	59.81 (53.63, 67.70)	42.43 (37.42, 44.73)

mutation analysis 30 times for each selected program and reported the average values. We also conducted statistical tests to measure the significance and effect size of the differences between our approach and the existing methods. Figure 10 shows the box-whisker plot of the mutation score distribution for evaluated test data generation approaches. It is observed that, on average, the mutation score of our domain extraction approach is higher than the path testing approach, i.e., ACO-AR-SP (Monemi Bidgoli and Haghighi 2020). Moreover, the mutation score of the path testing approach is higher than the branch testing approach, i.e., Pynguin (Lukasczyk et al. 2020).

Table 7 shows the average mutation score of each test data generation approach on the selected program after 30 experiments. Table 7 also reports the maximum and minimum values of mutation scores for each program. As a result of the experiments conducted for different programs, the maximum value obtained by the proposed method is higher than that obtained by the other two methods. Due to multiple tests of the program with the test data, we have obtained more accurate boundary values by applying the proposed method. Similarly, the test data have been obtained due to the more accurate coverage of the domain space. Consequently, the minimum value of the results for the proposed method is better than those for the other two methods for these reasons. Based on our findings, the difference between the highest and lowest mutation scores of the proposed method is generally less than the difference between the highest and lowest mutation scores of the other two methods, indicating that the proposed method is more reliable and that random factors have a less substantial impact on the results.

Table 8 Statistical results (p value) of mutation score analysis by Wilcoxon test

Porgram under test	MCMC-DOM vs. Pynguin	MCMC-DOM vs. ACO-AR-SP
Armstrong-numbers (++)	1.00×10^{-3}	1.12×10^{-3}
bisection (±)	1.80×10^{-3}	5.20×10^{-2}
decimaltoany (++)	8.20×10^{-3}	1.00×10^{-3}
reminder (++)	1.00×10^{-3}	1.00×10^{-3}
expint (±)	2.40×10^{-3}	3.90×10^{-1}
gammaq (±)	1.00×10^{-3}	8.00×10^{-1}
gcd (++)	7.90×10^{-3}	3.30×10^{-3}
CheckTriangle (++)	6.40×10^{-3}	1.00×10^{-3}
Scipy (++)	1.30×10^{-2}	2.00×10^{-3}
Mpmath (++)	1.00×10^{-3}	3.00×10^{-3}

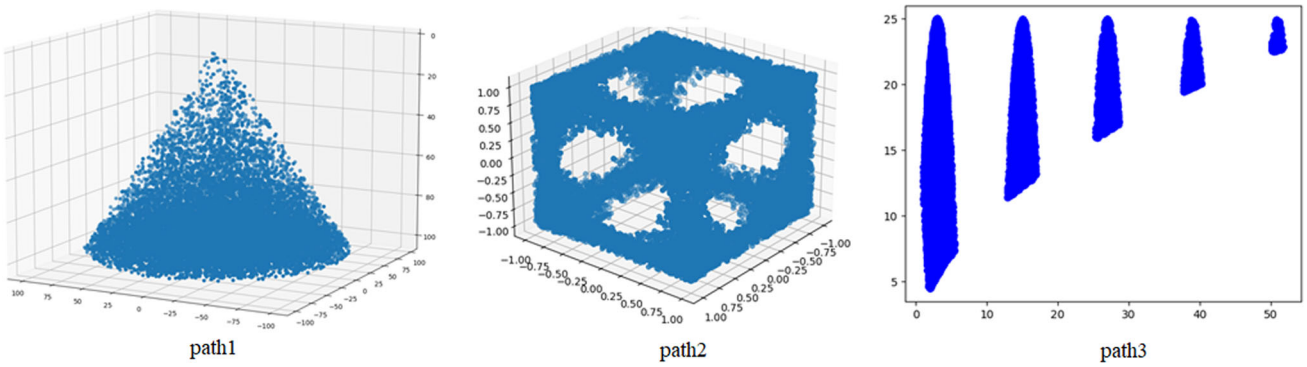

Fig. 11 Illustration of the domains extracted for paths in Listing 1

The proposed method performs better than the two methods based on path and branch coverage in terms of averages, maximum, and minimum values. Statistical tests are used to demonstrate that the proposed method performs significantly better than the other two methods (Meléndez et al. 2020). A statistical test such as a t-test or Wilcoxon is used to compare the results of two groups of experiments. Normal distributions are tested with a t-test, which is a parametric test. The Wilcoxon test is a non-parametric test used when the distribution is not normal. The Kolmogorov–Smirnov test determines the normality of a data series (Berger and Zhou 2014). Based on the Kolmogorov–Smirnov test, it has been determined that the p value for mutation scores obtained by different methods is equal to 0.025, indicating that this test is accepted and that the distribution of the data is not normal. Therefore, the Wilcoxon test is used to compare the results of experiments with different methods defined as distinct groups.

The Wilcoxon test was used to pairwise compare the ACO-AR-SP (Monemi Bidgoli and Haghighi 2020) and Pynguin test effectiveness with our proposed method. Table 8 shows the p values of the Wilcoxon test on each program under test. A p value less than 0.05 shows a significant difference between the results of the two methods. A "+" symbol at the first position in the sequence of signs presented beside each program name means that the mutation score of our approach is statically different from the Pynguin mutation score. A "+" symbol at the second position means the same for the ACO-AR-SP approach. A "−" symbol means the opposite for each program. Overall, the mutation score distribution of our proposed approach is significantly higher than those of the ACO-AR-SP (Monemi Bidgoli and Haghighi 2020) and Pynguin (Lukasczyk et al. 2020) approaches.

RQ4. *What is the fault detection capability of the selected test data from the extracted domain by our approach?*

Summary for RQ4: *The proposed domain extraction approach has the highest fault detection capability compared to the branch and path-based testing approaches. The average mutation score of the proposed method on ten Python programs is 76.50%, which is significantly higher than the average mutation score of ACO-AR-SP* [21] *and Pynguin test data generation tools* [20].

4.4.5 Visualization of the extracted domains

Visualization helps better understand the regions of the inputs that correspond to a given path domain, specifically when the input dimensions and their range are not too large. Listing 1 describes the simplified form of three paths of the programs we tested in our experiments. The input dimensions for these paths are two and three. Therefore, we can visualize the paths' domains with 2D and 3D plots. Figure 11 shows the extracted domains by our proposed approach for each path in Listing 1. The domain of path1 has a three-dimensional distribution representing a cone. Path2 is defined in a small range but with a manifold structure, which increases the possibility of error for the domain extraction method. The domain of path3 is a multi-piece domain, which also challenges the exploitation rate of the proposed method. We observe that the domain of selected paths is extracted with relatively high accuracy.

Listing 1. Example paths illustrated for domain extraction

```
// Path 1: input: x, y, z
k = x ** 2;
if k > 0:
    H = y ** 2;
    L = k + H ** 0.5;
    return true;
// Path 2: input: x, y, z
k = x ** 2 + y ** 2
if k < 5:
    h = pow(z, 2);
     if h + k < 1.4:
         return true;
// Path 3: input: x, y
if x >= 0 and y >= 0:
    k = y**2;
    h = math.sin(math.radians(x * 30)) * 25;
    l = math.cos(math.radians(x * 40)) * 15;
      if k >= x*10 and y <= h and y >= l:
         return true;
```

5 Threats to validity

Several factors may threaten the internal and external validity of our results. The main threat regarding the internal validity of our evaluation results is the stochastic nature of the proposed algorithm and algorithms in related works. Indeed, IP-PRT, PRT, and MCMC algorithms start randomly, and some operators with random behavior have been used to implement these methods. The operators with random behavior affect the performance and accuracy of the methods. We tried to reduce the effect of random behavior with the design test scenario by executing each program under test in our experiments 30 times. The average value has been considered for these experiments.

The generalization of our approach to other software systems and programs written in other programming languages threatens the external validity reported in our study. We selected eight programs with different complexities and structures often appearing in real-world programs. Most of these programs have been used in test data generation literature. We also evaluated our approach on complex modules from two industrial Python packages, 'mpmath' and 'scipy,' widely used for data analysis and scientific computing. However, the lack of standard case studies is a major challenge in test data generation evaluation and comparison. It is necessary to create real-world benchmarks that can assess and compare different approaches in a fair and rigorous manner.

6 Conclusion

Executing a faulty statement affecting the execution result does not necessarily cause a faulty result or outcome. Such coincidentally correct executions often require more than one test to reveal themselves. By prioritizing the selection of test data from the domain of inputs for minimum cost test paths (MCTPs), the time and cost of finding these types of hidden faults are highly reduced, and the probability of success in finding them increases. The domain extraction approach proposed in this paper facilitates the detection of the domain of inputs for a given execution path. As the size of the input space increases, the probability of finding all the regions of the input space covering a desired execution path reduces. The proposed multi-agent Markov chain Monte Carlo (MCMC) sampling method assigns concurrent agents to look for appropriate test data as points in different regions of the input space. The agents cooperate to exploit the boundaries of the regions in the input space covering a desired MCTP while exploring new regions.

The evaluation of our approach on ten Python programs indicates that it can achieve high accuracy and effectiveness in fault detection. The average accuracy of the extracted domain with our proposed method is 85%, which significantly improves the accuracy of the state-of-the-art domain testing methods by 14%. Our domain extraction approach also finds nearly 78.9% of the boundary points of the domains for given paths, which are often more fault-prone than the internal points of a domain. Therefore, our approach can generate compelling test suites that reveal more difficult faults in the program under test. The test suites generated by our method achieve an average mutation score of 76.50% on ten Python programs, which is significantly higher than state-of-the-art branch and path-based testing tools by 22.28% and 13.72%, respectively. Our experiments also indicate that a path's complexity negatively impacts the accuracy of the domain extraction algorithm due to the changes in the shape and sparsity of valid regions.

Further investigations are required to improve the performance of domain extraction approaches. One possible direction for future work is to refactor the program under test to simplify conditional logic and remove long parameter lists before applying the domain extraction algorithms, which can reduce the complexity and dimensionality of the problem. Another direction for future work is to use the extracted domain in program debugging to improve the fault localization performance by generating adequate test data for the faulty path detected in testing. Generating many test data for a path during debugging can help to narrow down the search space and identify the location of a detected fault more accurately. Finally, machine learning techniques can be used to train a predictive model that predicts program outputs during the MCMC sampling process. This model can then be used instead of the actual program to guide the sampling process without executing the program, thereby increasing the efficiency and scalability of the proposed MCMC-DOM approach.

Funding This study has received no funding from any organization.

Data availability The datasets generated and analyzed during the current study are available in the MCMC-DOM GitHub repository, https://github.com/roshangol/Domain_coverage.

Declarations

Conflict of interest All the authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Abdollahi A, Azhdary Moghaddam M, Hashemi Monfared SA, Rashki M, Li Y (2021) Subset simulation method including fitness-based seed selection for reliability analysis. Eng Comput 37(4):2689–2705. https://doi.org/10.1007/s00366-020-00961-9

Afzal W, Torkar R, Feldt R (2009) A systematic review of search-based testing for non-functional system properties. Inf Softw Technol 51(6):957–976. https://doi.org/10.1016/j.infsof.2008.12.005

Ammann P, Offutt J (2016) Introduction to software testing. Cambridge University Press, Cambridge. https://doi.org/10.1017/9781316771273

Avrachenkov K, Borkar VS, Kadavankandy A, Sreedharan JK (2018) Revisiting random walk based sampling in networks: evasion of burn-in period and frequent regenerations. Comput Soc Netw 5(1):4. https://doi.org/10.1186/s40649-018-0051-0

Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I (2019) A survey of symbolic execution techniques. ACM Comput Surv 51(3):1–39. https://doi.org/10.1145/3182657

Berger VW, Zhou Y (2014) Kolmogorov–Smirnov test: overview. In: Wiley StatsRef: statistics reference online. Wiley.https://doi.org/10.1002/9781118445112.stat06558

Birt JR, Sitte R (2004) Optimizing testing efficiency with error-prone path identification and genetic algorithms. In: Proceedings 2004 Australian software engineering conference. IEEE, pp 106–115. https://doi.org/10.1109/ASWEC.2004.1290463

Brooks S, Gelman A, Jones Galin L, Meng X (2011) Handbook of Markov chain Monte Carlo. CRC Press, Boca Raton

Chan FT, Chen TY, Mak IK, Yu YT (1996) Proportional sampling strategy: guidelines for software testing practitioners. Inf Softw Technol 38(12):775–782. https://doi.org/10.1016/0950-5849(96)01103-2

Chan KP, Chen TY, Towey D (2006) Restricted random testing: adaptive random testing by exclusion. Int J Softw Eng Knowl Eng 16(04):553–584. https://doi.org/10.1142/S0218194006002926

Changye W, Robert CP (2020) Markov chain Monte Carlo algorithms for Bayesian computation, a survey and some generalisation, pp 89–119. https://doi.org/10.1007/978-3-030-42553-1_4

Chen TY, Tse TH, Yu YT (2001) Proportional sampling strategy: a compendium and some insights. J Syst Softw 58(1):65–81. https://doi.org/10.1016/S0164-1212(01)00028-0

Chen TY, Kuo F-C, Merkel RG, Ng SP (2004) Mirror adaptive random testing. Inf Softw Technol 46(15):1001–1010. https://doi.org/10.1016/j.infsof.2004.07.004

Chen TY, Eddy G, Merkel R, Wong PK (2004) Adaptive random testing through dynamic partitioning. In: Proceedings fourth international conference on quality software, 2004. QSIC 2004. IEEE, pp 79–86. https://doi.org/10.1109/QSIC.2004.1357947

Chen TY, Huang D, Tse TH, Yang Z (2007) An innovative approach to tackling the boundary effect in adaptive random testing. In: 2007 40th annual Hawaii international conference on system sciences (HICSS'07). IEEE, p 262a. https://doi.org/10.1109/HICSS.2007.67

Chen TY, Kuo F-C, Merkel RG, Tse TH (2010) Adaptive random testing: the ART of test case diversity. J Syst Softw 83(1):60–66. https://doi.org/10.1016/j.jss.2009.02.022

Dobslaw F, de Oliveira Neto FG, Feldt R (2020) Boundary value exploration for software analysis. In: 2020 IEEE international conference on software testing, verification and validation workshops (ICSTW). IEEE, pp 346–353. https://doi.org/10.1109/ICSTW50294.2020.00062

Dubrova E (2013) Fault-tolerant design. Springer Publishing Company, Incorporated, Berlin

Ebert C, Cain J, Antoniol G, Counsell S, Laplante P (2016) Cyclomatic complexity. IEEE Softw 33(6):27–29. https://doi.org/10.1109/MS.2016.147

Fraser G, Arcuri A (2013) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291. https://doi.org/10.1109/TSE.2012.14

Goodenough JB, Gerhart SL (1975) Toward a theory of test data selection. IEEE Trans Softw Eng 1(2):156–173. https://doi.org/10.1109/TSE.1975.6312836

Gotlieb A, Petit M (2006) Path-oriented random testing. In: Proceedings of the 1st international workshop on random testing—RT '06. ACM Press, New York 2006, p 28. https://doi.org/10.1145/1145735.1145740

Gotlieb A, Petit M (2010) A uniform random test data generator for path testing. J Syst Softw 83(12):2618–2626. https://doi.org/10.1016/j.jss.2010.08.021

Hałas K, Hossner P, Myint S, Mueller A (2022) Mutpy [Online]. https://github.com/mutpy/mutpy. Accessed 25 Jan 2022

Hamlet D, Taylor R (1990) Partition testing does not inspire confidence (program testing). IEEE Trans Softw Eng 16(12):1402–1411. https://doi.org/10.1109/32.62448

Hogg DW, Foreman-Mackey D (2018) Data analysis recipes: using Markov chain Monte Carlo. Astrophys J Suppl Ser 236(1):11. https://doi.org/10.3847/1538-4365/aab76e

Huang R, Liu H, Xie X, Chen J (2015) Enhancing mirror adaptive random testing through dynamic partitioning. Inf Softw Technol 67:13–29. https://doi.org/10.1016/j.infsof.2015.06.003

Huang R, Cui C, Sun W, Towey D (2020) Poster: is Euclidean distance the best distance measurement for adaptive random testing? In: 2020 IEEE 13th international conference on software testing, validation and verification (ICST), IEEE, pp 406–409. https://doi.org/10.1109/ICST46399.2020.00049.

Huang R, Sun W, Chen TY, Ng S, Chen J (2021) Identification of failure regions for programs with numeric inputs. IEEE Trans Emerg Top Comput Intell 5(4):651–667. https://doi.org/10.1109/TETCI.2020.3013713

Huang R, Sun W, Xu Y, Chen H, Towey D, Xia X (2021) A survey on adaptive random testing. IEEE Trans Softw Eng 47(10):2052–2083. https://doi.org/10.1109/TSE.2019.2942921

Jain N, Porwal R (2019) Automated test data generation applying heuristic approaches—a survey, pp 699–708. https://doi.org/10.1007/978-981-10-8848-3_68

Jones C, Bonsignour O (2012) The economics of software quality. Addison-Wesley [Online]. https://books.google.com/books?id=_t5l5Cn0NBEC

Kelly D, Gray R, Shao Y (2011) Examining random and designed tests to detect code mistakes in scientific software. J Comput Sci 2(1):47–56. https://doi.org/10.1016/j.jocs.2010.12.002

Khari M, Kumar P (2019) An extensive evaluation of search-based software testing: a review. Soft Comput 23(6):1933–1946. https://doi.org/10.1007/s00500-017-2906-y

Khari M, Kumar P, Burgos D, Crespo RG (2018) Optimized test suites for automated testing using different optimization techniques. Soft Comput 22(24):8341–8352. https://doi.org/10.1007/s00500-017-2780-7

Li N, Li F, Offutt J (2012) Better algorithms to minimize the cost of test paths. In: 2012 IEEE fifth international conference on software testing, verification and validation. IEEE, pp 280–289. https://doi.org/10.1109/ICST.2012.108

Luengo D, Martino L, Bugallo M, Elvira V, Särkkä S (2020) A survey of Monte Carlo methods for parameter estimation. EURASIP J Adv Signal Process 2020(1):25. https://doi.org/10.1186/s13634-020-00675-6

Lukasczyk S, Kroiß F, Fraser G (2020) Automated unit test generation for python, pp 9–24. https://doi.org/10.1007/978-3-030-59762-7_2

Mao C (2014) Harmony search-based test data generation for branch coverage in software structural testing. Neural Comput Appl 25(1):199–216. https://doi.org/10.1007/s00521-013-1474-z

Marculescu B, Feldt R (2018) Finding a boundary between valid and invalid regions of the input space. In: 2018 25th Asia-Pacific software engineering conference (APSEC). IEEE, pp 169–178. https://doi.org/10.1109/APSEC.2018.00031

Mattingly JC, Pillai NS, Stuart AM (2012) Diffusion limits of the random walk Metropolis algorithm in high dimensions. Ann Appl Probab. https://doi.org/10.1214/10-AAP754

Meléndez R, Giraldo R, Leiva V (2020) Sign, Wilcoxon and Mann–Whitney tests for functional data: an approach based on random projections. Mathematics 9(1):44. https://doi.org/10.3390/math9010044

Mishra DB, Mishra R, Das KN, Acharya AA (2019) Test case generation and optimization for critical path testing using genetic algorithm, pp 67–80. https://doi.org/10.1007/978-981-13-1595-4_6

Monemi Bidgoli A, Haghighi H (2020) Augmenting ant colony optimization with adaptive random testing to cover prime paths. J Syst Softw 161:110495. https://doi.org/10.1016/j.jss.2019.110495

Nanthaamornphong A, Carver JC (2017) Test-driven development in scientific software: a survey. Softw Qual J 25(2):343–372. https://doi.org/10.1007/s11219-015-9292-4

Nikravan E, Parsa S (2019) Path-oriented random testing through iterative partitioning (IP-PRT). Turk J Electr Eng Comput Sci 27(4):2666–2680. https://doi.org/10.3906/elk-1801-18

Papadakis M, Kintis M, Zhang J, Jia Y, Le Traon Y, Harman M (2019) Mutation testing advances: an analysis and survey, pp 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

Parsa S (2023) Coincidentally correct executions. In: Software testing automation: testability evaluation, refactoring, test data generation and fault localization. Springer International Publishing, Cham, pp 365–408. https://doi.org/10.1007/978-3-031-22057-9_9

Ramler R, Wetzlmaier T, Klammer C (2017) An empirical study on the application of mutation testing for a safety-critical industrial software system. In: Proceedings of the symposium on applied computing. ACM, New York, pp 1401–1408. https://doi.org/10.1145/3019612.3019830

Robert CP, Elvira V, Tawn N, Wu C (2018) Accelerating MCMC algorithms. Wires Comput Stat. https://doi.org/10.1002/wics.1435

Sahoo RR, Ray M (2020) PSO based test case generation for critical path using improved combined fitness function. J King Saud Univ Comput Inf Sci 32(4):479–490. https://doi.org/10.1016/j.jksuci.2019.09.010

Song K, Zhang Y, Zhuang X, Yu X, Song B (2021) Reliability-based design optimization using adaptive surrogate model and importance sampling-based modified SORA method. Eng Comput 37(2):1295–1314. https://doi.org/10.1007/s00366-019-00884-0

Titcheu Chekam T, Papadakis M, Bissyandé TF, Le Traon Y, Sen K (2020) Selecting fault revealing mutants. Empir Softw Eng 25(1):434–487. https://doi.org/10.1007/s10664-019-09778-7

Wu H, Nie C, Petke J, Jia Y, Harman M (2020) An empirical comparison of combinatorial testing, random testing and adaptive random testing. IEEE Trans Softw Eng 46(3):302–320. https://doi.org/10.1109/TSE.2018.2852744

Wu Y, Liu Y, Wang W, Li Z, Chen X, Doyle P (2022) Theoretical analysis and empirical study on the impact of coincidental correct test cases in multiple fault localization. IEEE Trans Reliab 71(2):830–849. https://doi.org/10.1109/TR.2022.3165126

Xue X, Pang Y, Namin AS (2014) Trimming test suites with coincidentally correct test cases for enhancing fault localizations. In: 2014 IEEE 38th annual computer software and applications

conference. IEEE, pp 239–244. https://doi.org/10.1109/COMPSAC.2014.32

Zhou B, Okamura H, Dohi T (2013) Enhancing performance of random testing through Markov chain Monte Carlo methods. IEEE Trans Comput 62(1):186–192. https://doi.org/10.1109/TC.2011.208

Zhu Q, Panichella A, Zaidman A (2018) A systematic literature review of how mutation testing supports quality assurance processes. Softw Test Verif Reliab 28(6):e1675. https://doi.org/10.1002/stvr.1675