A Survey on Adaptive Random Testing

Rubing Huang, Weifeng Sun, Yinyin Xu, Haibo Chen, Dave Towey, Xin Xia

Abstract—Random testing (RT) is a well-studied testing method that has been widely applied to the testing of many applications, including embedded software systems, SQL database systems, and Android applications. Adaptive random testing (ART) aims to enhance RT's failure-detection ability by more evenly spreading the test cases over the input domain. Since its introduction in 2001, there have been many contributions to the development of ART, including various approaches, implementations, assessment and evaluation methods, and applications. This paper provides a comprehensive survey on ART, classifying techniques, summarizing application areas, and analyzing experimental evaluations. This paper also addresses some misconceptions about ART, and identifies open research challenges to be further investigated in the future work.

Index Terms—Adaptive random testing, random testing, survey.

1 Introduction

Software testing is a popular technique used to assess and assure the quality of the (software) system under test (SUT). One fundamental testing approach involves simply constructing test cases in a random manner from the *input domain* (the set of all possible program inputs): This approach is called *random testing* (RT) [1]. RT may be the only testing approach used not only for operational testing, where the software reliability is estimated, but also for debug testing, where software failures are targeted with the purpose of removing the underlying bugs[1] [3]. Although conceptually very simple, RT has been used in the testing of many different environments and systems, including: Windows NT applications [4]; embedded software systems [5]; SQL database systems [6]; and Android applications [7].

RT has generated a lot of discussion and controversy, notably in the context of its effectiveness as a debug testing method [8]. Many approaches have been proposed to enhance RT's testing effectiveness, especially for failure detection. *Adaptive random testing* (ART) [9] is one such proposed improvement over RT. ART was motivated by observations reported independently by many researchers from multiple different areas regarding the behavior and patterns of software failures: Program inputs that trigger failures (*failure-causing inputs*) tend to cluster into contiguous regions (*failure regions*) [10]–[14]. Furthermore, if the failure regions are contiguous, then it follows that non-failure regions should also be adjacent throughout the input domain. Specifically: if a test case tc is a failure-causing input, then its neighbors have a high probability of also being failure-causing; similarly, if tc is not failure-causing, then its neighbors have a high probability of also not being failure-causing. In other words, a program input that is far away from non-failure-causing inputs may have a higher probability of causing failure than the neighboring test inputs. Based on this, ART aims to achieve an even spread of (random) test cases over the input domain. ART generally involves two processes: one for the random generation of test inputs; and another to ensure an even-spreading of the inputs throughout the input domain [15].

ART's invention and appearance in the literature can be traced back to a journal paper by Chen et al., published in 2001 [9]. Some papers present overviews of ART, but are either preliminary, or do not make ART the main focus [15]–[20]. For example, to draw attention to the fundamental role of diversity in test case selection strategies, Chen et al. [15] presented a synthesis of some of the most important ART research results before 2010. Similarly, Anand et al. [18] presented an orchestrated survey of the most popular techniques for automatic test case generation that included a brief report on the then state-of-the-art of ART. Roslina et al. [19] also conducted a study of ART techniques based on 61 papers. Consequently, there is currently no up-to-date, exhaustive survey analyzing both the state-of-the-art and new potential research directions of ART. This paper fills this gap in the literature.

In this paper, we present a comprehensive survey on ART covering 140 papers published between 2001 and 2017. The paper includes the following: (1) a summary and analysis of the selected 140 papers; (2) a description, classification, and summary of the techniques used to derive the main ART strategies; (3) a summary of the application and testing domains in which ART has been applied; (4) an analysis of

- *Rubing Huang is with the School of Computer Science and Communication Engineering, and also with Jiangsu Key Laboratory of Security Technology for Industrial Cyberspace, Jiangsu University, Zhenjiang, Jiangsu 212013, China.*
  E-mail: rbhuang@ujs.edu.cn.
- *Weifeng Sun, Yinyin Xu, and Haibo Chen are with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, Jiangsu 212013, China.*
  E-mail: {2211808031, 2221808040, 2221808034}@stmail.ujs.edu.cn.
- *D. Towey is with the School of Computer Science, University of Nottingham Ningbo China, Ningbo, Zhejiang 315100, China.*
  E-mail: dave.towey@nottingham.edu.cn.
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, VIC 3168, Australia.*
  E-mail: xin.xia@monash.edu.

1. According to the IEEE [2], the relationship amongst *mistake*, *fault*, *bug*, *defect*, *failure* and *error* can be briefly explained as follows: A software developer makes a *mistake*, which may introduce a *fault* (*defect* or *bug*) in the software. When a fault is encountered, a *failure* may be produced, i.e., the software behaves unexpectedly. "An *error* is the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition." [2, p.128].

the empirical studies conducted into ART; (5) a discussion of some misconceptions surrounding ART; and (6) a summary of some open research challenges that should be addressed in future ART work. To the best of our knowledge, this is the first large-scale and comprehensive survey on ART.

The rest of this paper is organized as follows. Section 2 briefly introduces the preliminaries and gives an overview of ART. Section 3 discusses this paper's literature review methodology. Section 4 examines the evolution and distribution of ART studies. Section 5 analyzes the-state-of-the-art of ART techniques. Section 6 presents the situations and problems to which ART has been applied. Section 7 gives a detailed analysis of the various empirical evaluations of ART. Section 8 discusses some misconceptions. Section 9 provides some potential challenges to be addressed in future work. Finally, Section 10 concludes the paper.

2 Background

This section presents some preliminary concepts, and provides an introduction to ART.

2.1 Preliminaries

For a given SUT, many software testing methods have been implemented according to the following four steps: (1) define the testing objectives; (2) choose inputs for the SUT (*test cases*); (3) run the SUT with these test cases; and (4) analyze the results. Each test case is selected from the set of all possible inputs that form the *input domain*. When the SUT's output or behavior when executing a test case tc is not as expected (determined by the *test oracle* [21]), then the test is considered to *fail*, otherwise it is *passes*. When a test fails, tc is called a *failure-causing input*.

Given some faulty software, two fundamental features can be used to describe the properties of the fault(s): the *failure rate* (the number of failure-causing inputs as a proportion of all possible inputs); and the *failure pattern* (the distributions of failure-causing inputs across the input domain, including their geometric shapes and locations). Before testing, these two features are fixed, but unknown.

Chan et al. [22] identified three broad categories of failure patterns: *strip*, *block* and *point*. Fig. 1 illustrates these three failure patterns in a two-dimensional input domain (the bounding box represents the input domain boundaries; and the black strip, block, or dots represent the failure-causing inputs). Previous studies have indicated that strip and block patterns are more commonly encountered than point patterns [10]–[14].

Generally speaking, failure regions are identified or constructed in empirical studies (experiments or simulations). Experiments involve real faults or mutants (seeded using mutation testing [23]) in real-life subject programs. Simulations, in contrast, create artificial failure regions using predefined values for the dimensionality d and failure rate θ, and a predetermined failure pattern type: A d-dimensional unit hypercube is often used to simulate the input domain $\mathcal{D}$ ($\mathcal{D} = \{(x_1, x_2, \cdots, x_d) | 0 \leq x_1, x_2, \cdots, x_d < 1.0\}$), with the failure regions randomly placed inside $\mathcal{D}$, and their sizes and shapes determined by θ and the selected failure pattern, respectively. During testing, when a generated test case is inside a failure region, a failure is said to be detected.

2.2 Adaptive Random Testing (ART)

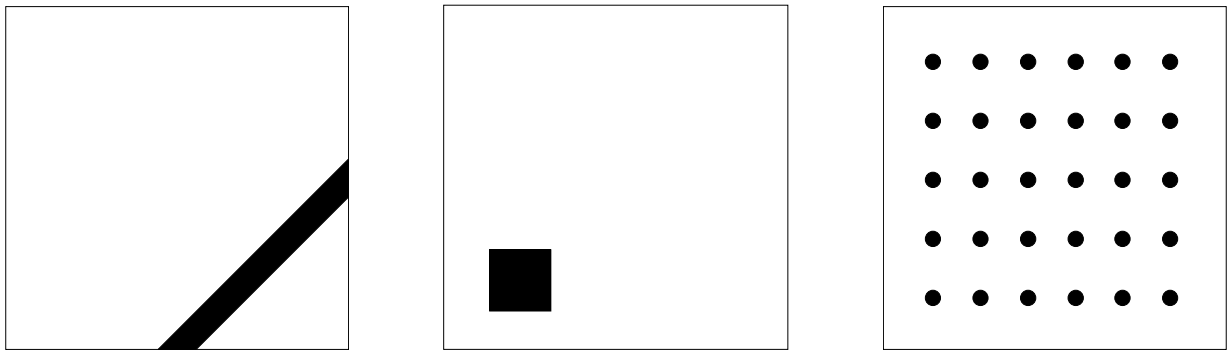
ART is a family of testing methods, with many different implementations based on various intuitions and criteria. In this section, we present an ART implementation to illustrate the fundamental principles.

The first implementation of ART was the *Fixed-Size-Candidate-Set* (FSCS) [9] version, which makes use of the concept of distance between test inputs. FSCS uses two sets of test cases: the candidate set C; and the executed set, E. C is a set of k tests randomly generated from the input domain (according to the specific distribution); and E is the set of those inputs that have already been executed, but without causing any failure. E is initially empty. The first test input is generated randomly. In each iteration of FSCS ART, an element from C is selected as the next test case such that it is farthest away from all previously executed tests (those in E). Formally, the element c' from C is chosen as the next test case such that it satisfies the following constraint:

$$\forall c \in C,\ \min_{e \in E} dist(c', e) \geq \min_{e \in E} dist(c, e), \tag{2.1}$$

where $dist(x, y)$ is a function to measure the distance between two test inputs x and y. The *Euclidean distance* is typically used in $dist(x, y)$ for numerical input domains.

Fig. 2 illustrates the FSCS process in a two-dimensional input domain: Suppose that there are three previously executed test cases, t_1, t_2, and t_3 (i.e., $E = \{t_1, t_2, t_3\}$), and



(a) Strip pattern (b) Block pattern (c) Point pattern

Fig. 1. Three types of failure patterns in the two-dimensional input domain.

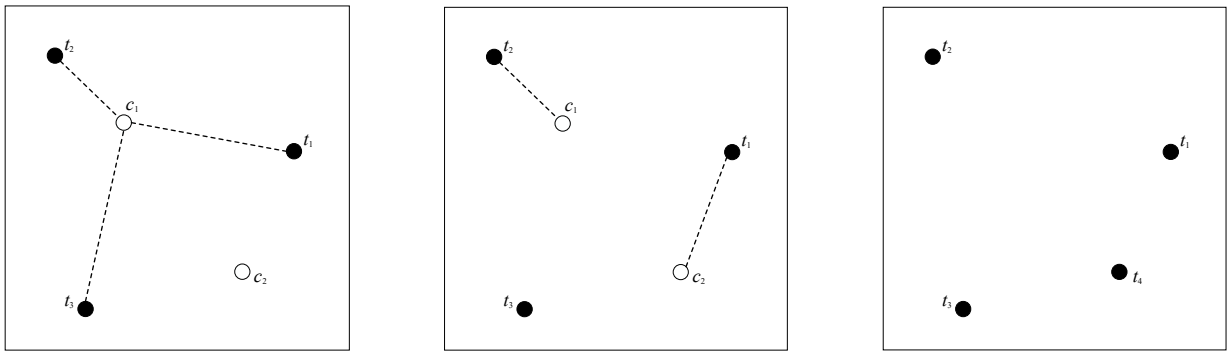


Fig. 2. Illustration of FSCS in a two-dimensional input domain.

two randomly generated test candidates, c_1 and c_2 (i.e., $C = \{c_1, c_2\}$) (Fig. 2(a)). To select the next test case from C, the distance between each candidate and each previously executed each test case in E is calculated, and the minimum value for each candidate is recorded as the *fitness value* (Fig. 2(b)). Finally, the candidate with the maximum fitness value is selected to be the next test case (Fig. 2(c)): in this example, c_2 is used for testing ($t_4 = c_2$).

As Chen et al. have explained [15], ART aims to more evenly spread randomly generated test cases than RT across the input domain. In other words, ART attempts to generate more diverse test cases than RT.

3 METHODOLOGY

Guided by Kitchenham and Charters [24] and Petersen et al. [25], in this paper, we followed a structured and systematic method to perform the ART survey. We also referred to recent survey papers on other software engineering topics, including: mutation analysis [23]; metamorphic testing [26], [27]; constrained interaction testing [28]; and test case prioritization for regression testing [29]. The detailed methodology used is described in this section.

3.1 Research Questions

The goal of this survey paper is to structure and categorize the available ART details and evidence. To achieve this, we used the following research questions (RQs):

- *RQ1: What has been the evolution and distribution of ART topics in the published studies?*
- *RQ2: What different ART strategies and approaches exist?*
- *RQ3: In what domains and applications has ART been applied?*
- *RQ4: How have empirical evaluations in ART studies been performed?*
- *RQ5: What misconceptions surrounding ART exist?*
- *RQ6: What are the remaining challenges and other future ART work?*

The answer to RQ1 will provide an overview of the published ART papers. RQ2 will identify the state-of-the-art in ART strategies and techniques, giving a description, summary, and classification. RQ3 will identify where and how ART has been applied. RQ4 will explore how the various ART studies involving simulations and experiments with real programs were conducted and evaluated. RQ5 will examine common ART misconceptions, and, finally, RQ6 will identify some remaining challenges and potential research opportunities.

3.2 Literature Search and Selection

Following previous survey studies [26], [28], [29], we also selected the following five online literature repositories belonging to publishers of technical research:

- ACM Digital Library
- Elsevier Science Direct
- IEEE Xplore Digital Library
- Springer Online Library
- Wiley Online Library

The choice of these repositories was influenced by the fact that a number of important journal articles about ART are available through Elsevier Science Direct, Springer Online Library, and Wiley Online Library. Also, ACM Digital Library and IEEE Xplore not only offer articles from conferences, symposia, and workshops, but also provide access to some important relevant journals.

After determining the literature repositories, each repository was searched using the exact phrase "adaptive random testing" and for titles or keywords containing "random test". To avoid missing papers that were not included in these five repositories, we augmented the set of papers using a search in the Google Scholar database with the phrase "adaptive random testing" as the search string[2]. The

2. The search was performed on May 1st, 2018.

TABLE 1
Initial Number of Papers for Each Search Engine

Search Engine	Studies
ACM Digital Library	29
Elsevier Science Direct	57
IEEE Xplore Digital Library	60
Springer Online Library	89
Wiley Online Library	31
Google Scholar	1010
Total	**1276**

TABLE 2
Data Collection for Research Questions

RQs	Type of Data Extracted
RQ1	Fundamental information for each paper (publication year, type of paper, author name, and affiliation).
RQ2	Motivation, description, and analysis of each technique.
RQ3	Language, function, and environment for each application.
RQ4	Simulation details, subject programs, evaluation metrics, fault types, and analysis.
RQ5	Current misconception details.
RQ6	Details of remaining challenges.

results were combined to form the candidate set of published studies shown in Table 1. Duplicates were removed, and then, to further reduce the candidate set size, we then applied the following exclusion criteria:

1) Studies not written in English.
2) Studies not related to the field of computer science.
3) Studies not related to ART.
4) Books, book chapters, or technical reports (most of which have been published as articles).
5) Master or Ph.D. theses.
6) Keynote records (because, generally, these are only extremely brief summaries or overviews — e.g., Chen's keynote at the 8th International Conference on Quality Software [30]).
7) Studies without a full-text.

Removal of duplicates and application of the exclusion criteria reduced the initial 1,276 candidate studies to 138 published papers. Finally, a snowballing process [25] was conducted by checking the references of the selected 138 papers, resulting in the addition of two more papers. In total, 140 publications (*primary studies*) were selected for inclusion in the survey.

We acknowledge the apparent infeasibility of finding all ART papers through our search. However, we are confident that we have included the majority of relevant published papers, and that our survey provides the overall trends and the-state-of-the-art of ART.

3.3 Data Extraction and Collection

All 140 primary studies were carefully read and inspected, with data extracted according to our research questions. As summarized in Table 2, we identified the following information from each study: motivation, contribution, empirical evaluation details, misconceptions, and remaining challenges. To avoid missing information and reduce error as much as possible, this process was performed by two different co-authors, and subsequently verified by the other co-authors at least twice.

4 Answer to RQ1: What has been the evolution and distribution of ART topics in the published studies?

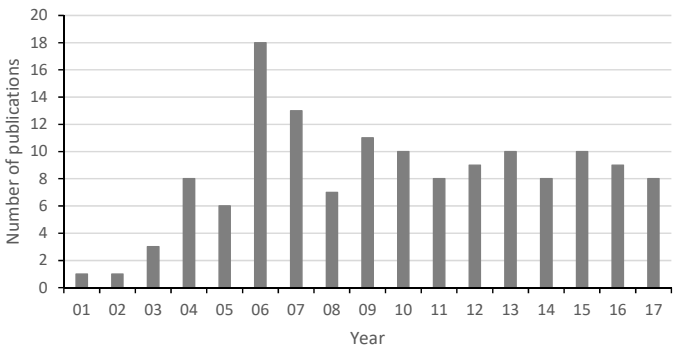
In this section, we address RQ1 by summarizing the primary studies according to publication trends, authors, venues, and types of contributions to ART.

4.1 Publication Trends

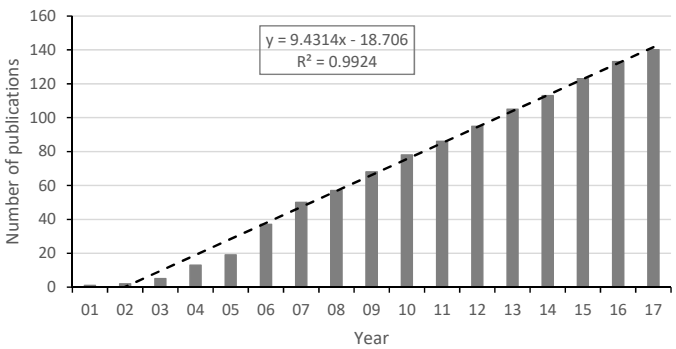
Fig. 3 presents the ART publication trends between January 1st, 2001 and December 31st, 2017, with Fig. 3(a) showing the number of publications each year, and Fig. 3(b) showing the cumulative number. It can be observed that, after the first three years there are at least six publications per year, with the number reaching a peak in 2006. Furthermore, since 2009, the number of publications each year has remained relatively fixed, ranging from seven to 10. An analysis of the cumulative publications (Fig. 3(b)) shows that a line function with high determination coefficient ($R^2 = 0.9924$) can be identified. This indicates that the topic of ART has been experiencing a strong linear growth, attracting continued interest and showing healthy development. Following this trend, it is anticipated that there will be about 180 ART papers by 2021, two decades after its appearance in Chen et al. [9].

4.2 Researchers and Organizations

Based on the 140 primary studies, 167 ART authors were identified, representing 82 different affiliations. Table 3 lists the top 10 ART authors and their most recent affiliation (with country or region). It is clear that T. Y. Chen, from Swinburne University of Technology in Australia, is the most prolific ART author, with 62 papers.



(a) Number of publications per year.



(b) Cumulative number of publications per year.

Fig. 3. ART papers published between January 1st 2001 and December 31st 2017.

TABLE 3
Top Ten ART Authors

Rank	Name	Current Affiliation	Country or Region	Papers
1.	T. Y. Chen	Swinburne University of Technology	Australia	62
2.	F.-C. Kuo	Swinburne University of Technology	Australia	35
3.	H. Liu	Victoria University	Australia	19
4.	R. G. Merkel	Monash University	Australia	15
5.	D. Towey	University of Nottingham Ningbo China	PRC	15
6.	J. Mayer	Ulm University	Germany	13
7.	K. P. Chan	The University of Hong Kong	Hong Kong	10
8.	Z. Q. Zhou	University of Wollongong	Australia	9
9.	R. Huang	Jiangsu University	PRC	8
10.	L. C. Briand	University of Luxembourg	Luxembourg	7

4.3 Geographical Distribution of Publications

We examined the geographical distribution of the 140 publications according to the affiliation country of the first author, as shown in Table 4. We found that all primary studies could be associated with a total of 18 different countries or regions, with Australia ranking first, followed by the People's Republic of China (PRC). Overall, about 41% of ART papers came from Asia; 32% from Oceania; 19% from Europe; and about 8% from America. This distribution of papers suggests that the ART community may only be represented by a modest number of countries spread throughout the world.

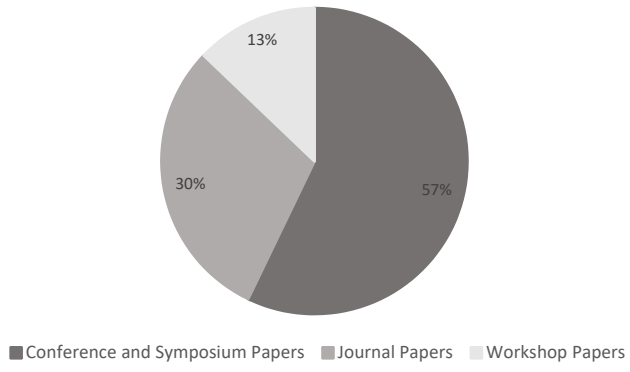
TABLE 4
Geographical Distribution of Publications

Rank	Country or Region	Papers
1.	Australia	45
2.	PRC	30
3.	Hong Kong	13
4.	Germany	13
5.	Norway	6
6.	United States	5
7.	Canada	4
8.	United Kingdom	4
9.	Malaysia	3
10.	Iran	3
11.	Indonesia	3
12.	Japan	2
13.	Switzerland	2
14.	Brazil	2
15.	Korea	2
16.	India	1
17.	Italy	1
18.	Luxembourg	1

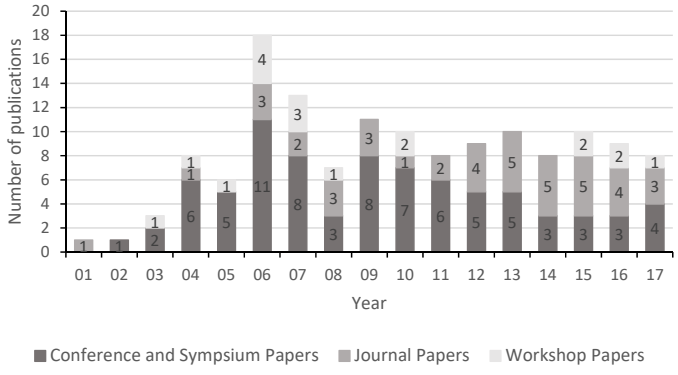
4.4 Distribution of Publication Venues

The 140 primary studies under consideration were published in 72 different venues (41 conferences or symposia, 20 journals, and 11 workshops). Fig. 4 illustrates the distribution of publication venues, with Fig. 4(a) showing the overall venue distribution, and Fig. 4(b) giving the venue distribution per year. As Fig. 4(a) shows, most papers have been published in conferences or symposia proceedings (57%), followed by journals (30%), and then workshops (13%). Fig. 4(b) shows that, between 2002 and 2012, most ART publications each year were conference and symposium papers, followed by journals and workshops. Fig. 4(b) also shows that, since 2012, this trend has changed, with the number of journal papers per year increasing, usually outnumbering the conference papers. The workshop papers generally form the least number of publications each year.

Table 5 lists the ranking of publication venues where at least three ART papers have appeared. Most of these venues



(a) Overall venue distribution.



(b) Venue distribution per year.

Fig. 4. Venue distribution for ART papers.

TABLE 5
Top Venues with a Minimum of Three ART Papers

Rank	Acronym	Full name	Papers
1.	QSIC	International Conference on Quality Software	11
2.	SEKE	International Conference on Software Engineering and Knowledge Engineering	8
3.	JSS	Journal of Systems and Software	7
4.	COMPSAC	Annual Computer Software and Applications Conference	6
5.	IEEE-TR	IEEE Transactions on Reliability	5
6.	SAC	ACM Symposium on Applied Computing	5
7.	RT	International Workshop on Random Testing	5
8.	ASE	International Conference on Automated Software Engineering	4
9.	IST	Information and Software Technology	4
10.	TSE	IEEE Transactions on Software Engineering	3
11.	TOSEM	ACM Transactions on Software Engineering and Methodology	3
12.	TOC	IEEE Transactions on Computers	3
13.	Ada-Europe	Ada-Europe International Conference on Reliable Software Technologies	3
14.	APSEC	Asia-Pacific Software Engineering Conference	3
15.	ICST	International Conference on Software Testing, Verification and Validation	3

are well-known and highly regarded in the field of software engineering or software testing.

4.5 Types of Contributions

Fig. 5 categorizes the primary studies according to main contribution (Fig. 5(a)) and research topic (Fig. 5(b))[3].

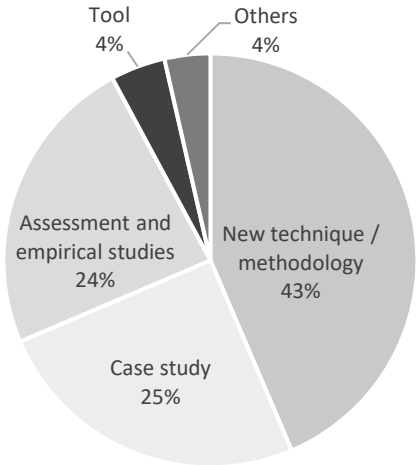
As Fig. 5(a) shows, the main contribution of 43% of the studies was to present new ART techniques or methodologies, 25% were case studies, and 24% were assessments and empirical studies. 4% of studies were surveys or overviews of ART, and the main contribution of five primary studies (4%) was to present a tool.

Fig. 5(b) shows that the primary research topic of 10% of the studies was about basic ART approaches. The effectiveness and efficiency enhancement of ART approaches were the focus of 29% and 7%, respectively; and application and assessment of ART were 27% and 23%, respectively. Finally, 4% of the papers report on techniques, achievements, and research directions.

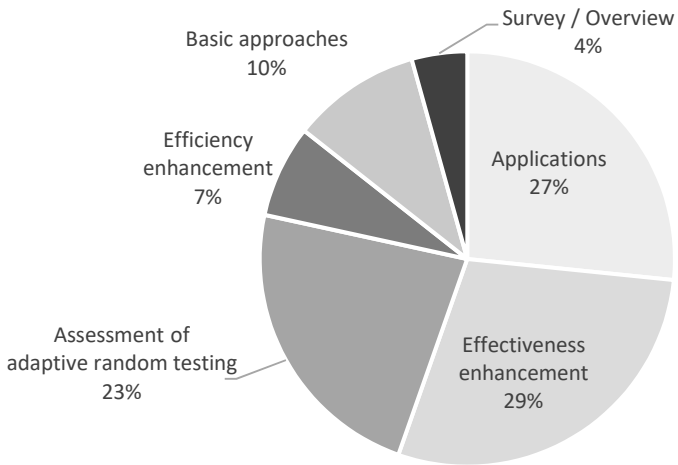
3. If a paper has multiple types of contributions or research topics, then only the main contribution or topic is identified.

Summary of answers to RQ1:

1) *ART has attracted sustained interest, with the topic showing healthy development.*
2) *Over 160 ART authors have been identified, representing more than 80 different affiliations, with T. Y. Chen being the most prolific.*
3) *Primary studies have come from 18 countries or regions, with Australia ranking first, followed by the PRC (People's Republic of China).*
4) *Most studies were published at conferences and symposia, followed by journals, and workshops.*
5) *The main contribution of most primary studies was to propose a new technique. The most popular research topics have been ART effectiveness enhancement, application, and assessment.*



(a) Type of main contribution.



(b) Research topic.

Fig. 5. Distribution of primary studies by main contribution and research topic.

5 Answer to RQ2: What different ART strategies and approaches exist?

In this section, we present the state-of-the-art of ART approaches, including a description, classification, and summary of their strengths and weaknesses.

ART attempts to spread test cases evenly over the entire input domain [15], which should result in a better failure detection ability [31]. There are two basic rationales to achieving this even spread of test cases:

Rationale 1: New test cases should be far away from previously executed, non-failure-causing test cases. As discussed in the literature [10]–[14], failure regions tend to be contiguous, which means that new test cases farther away from those already executed (but without causing failure) should have a higher probability of being failure-causing.

Rationale 2: New test cases should contribute to a good distribution of all test cases over the entire input domain. Previous studies [31] have empirically determined that better test case distributions result in better failure detection ability.

Discrepancy is a metric commonly used to measure the equidistribution of sample inputs [31], with lower values indicating better distributions. The discrepancy of a test set T is calculated as:

$$Discrepancy(T) = \max_{1 \le i \le m} \left| \frac{|T_i|}{|T|} - \frac{|\mathcal{D}_i|}{|\mathcal{D}|} \right|, \qquad (5.1)$$

where $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$ are m randomly defined subdomains of the input domain $\mathcal{D}$; and $T_1, T_2, \cdots T_m$ are the corresponding subsets of T, such that each T_i $(i = 1, 2, \cdots, m)$ is in $\mathcal{D}_i$. Discrepancy checks whether or not the number of test cases in a subdomain is proportionate to the relative size of the subdomain area — larger subdomains should have more test cases; and smaller ones should have relatively fewer.

Both **Rationale 1** and **Rationale 2** achieve a degree of *diversity* of test cases over the input domain [15]. Based on these rationales, many strategies have been proposed: *Select-Test-From-Candidates Strategy* (STFCS), *Partitioning-Based Strategy* (PBS), *Test-Profile-Based Strategy* (TPBS), *Quasi-Random Strategy* (QRS), *Search-Based Strategy* (SBS), and *Hybrid Strategies* (HSs).

5.1 Select-Test-From-Candidates Strategy

The *Select-Test-From-Candidates Strategy* (STFCS) chooses the next test case from a set of candidates based on some criteria or evaluation involving the previously executed test cases.

5.1.1 Framework

Fig. 6 presents a pseudocode framework for STFCS, showing two main components: the *random-candidate-set-construction*, and *test-case-selection*. The STFCS framework maintains two sets of test cases: the candidate set (C) of randomly generated candidate test cases; and the executed set (E) of those test cases already executed (without causing failure). The first test case is selected randomly from the input domain $\mathcal{D}$ according to a uniform distribution — all inputs in $\mathcal{D}$ have equal probability of selection. The test case is then applied to the SUT, and the output and behavior are examined to confirm whether or not a failure has been caused. Until a stopping condition is satisfied (e.g., a failure has been caused), the framework repeatedly uses the random-candidate-set-construction component to prepare the candidates, and then uses the test-case-selection component to choose one of these candidates as the next test case to be applied to the SUT.

1: Set $C \leftarrow \{\}$, and $E \leftarrow \{\}$
2: Randomly generate a test case tc from $\mathcal{D}$, according to uniform distribution
3: Add tc into E, i.e., $E \leftarrow E \bigcup \{tc\}$
4: **while** The stopping condition is not satisfied **do**
5: Randomly choose a specific number of elements from $\mathcal{D}$ to form C according to the specific criterion *Random-candidate-set-construction component*
6: Find a $tc \in C$ as the next test case satisfying the specific criterion *Test-case-selection component*
7: $E \leftarrow E \bigcup \{tc\}$
8: **end while**
9: Report the result and exit

Fig. 6. Framework pseudocode of the STFCS category.

Two basic (and popular) approaches to implementing the STFCS framework are *Fixed-Size-Candidate-Set (FSCS) ART* [9], [32], and *Restricted Random Testing (RRT)* [33]–[36][4]. Clearly, there are different ways to realize the random-candidate-set-construction and test-case-selection components, leading to different STFCS implementations. A number of enhanced versions of both FSCS and RRT have also been developed.

5.1.2 Random-Candidate-Set-Construction Component

Several different implementations of the random-candidate-construction component have been developed.

1) *Uniform distribution* [9]: This involves construction of the candidate set by randomly selecting test cases according to a uniform distribution.

2) *Non-uniform distribution* [37]: When not using a uniform distribution to generate the candidates, the non-uniform distribution is usually dynamically updated throughout the test case generation process. Chen et al. [37], for example, used a dynamic, non-uniform distribution to have candidates be more likely to come from the center of $\mathcal{D}$ than from the boundary region.

3) *Filtering by eligibility* [38], [39]: Using an eligibility criterion (specified using a tester-defined parameter), this filtering ensures that candidates (and therefore the eventually generated test cases) are drawn only from the eligible regions of $\mathcal{D}$. The criterion used is that the selected candidate's coordinates are as different as possible to those of all previously executed test cases. Given a test case in a d-dimensional input domain, $(x_1, x_2, \cdots, x_d)$, filtering by eligibility selects candidates such that each i-th coordinate, x_i $(1 \le i \le d)$, is different to the i-th coordinate of every previously selected test case. This ensures that all test cases have different values for all coordinates.

4. Previous ART studies have generally considered RRT to represent an *ART by exclusion* category [18], [19]. However, both RRT and FSCS belong to the STFCS category.

4) *Construction using data pools* [40], [41]: "Data pools" are first constructed by identifying and adding both specific special values for the particular data type (such as -1, 0, 1, etc. for integers), and the boundary values (such as the minimum and maximum possible values). This method then selects candidates randomly from either just the data pools, with a probability of p, or from the entire input domain, with probability of $1 - p$. Selected candidates are removed from the data pool. Once the data pool is exhausted, or falls below a threshold size, it is then updated by adding new elements.

5) *Achieving a specific degree of coverage* [42]: This involves selecting candidates randomly (with uniform distribution) from the input domain until some specific coverage criteria (such as branch, statement, or function coverage) are met.

5.1.3 Candidate Set Size

The size of the candidate set, k, may either be a fixed number (e.g., determined in advance, perhaps by the testers), or a flexible one. Although, intuitively, increasing the size of k should improve the testing effectiveness, as reported by Chen et al. [32], the improvement in FSCS ART performance is not significant when $k > 10$: In most studies, therefore, k has been assigned a value of 10. However, when the value of k is flexible, there are different methods to design and determine its value, based on the execution conditions or environment.

5.1.4 Test-Case-Identification Component

The test-case-identification component chooses one of the candidates as the next test case, according to the specific criterion. There are generally two different implementations: *Implementation 1*: After measuring all candidates, identifying the *best* one (as implemented in FSCS); and *Implementation 2*: Checking candidates until the first *suitable* (or *valid*) one is identified (as implemented in RRT). The goal of the test-case-identification component is to achieve the even spreading of test cases over the input domain, which it does based on the *fitness* value. In other words, the *fitness function* measures each candidate c from the candidate set C against the executed set E. We next list the seven different fitness functions, $fitness(c, E)$, used in STFCS, with the first six following *Implementation 1*; and the last one following *Implementation 2*.

1) *Minimum-Distance* [9]: This involves calculating the distance between c and each element e from E ($e \in E$), and then choosing the minimum distance as the fitness value for c. In other words, the fitness function of c against E is the distance between c and its nearest neighbor in E:

$$fitness(c, E) = \min_{e \in E} dist(c, e). \tag{5.2}$$

2) *Average-Distance* [40]: Similar to *Minimum-Distance*, this also computes the distance between c and each element e in E, but instead of the minimum, the average of these distances is used as the fitness value for c:

$$fitness(c, E) = \frac{1}{|E|} \sum_{e \in E} dist(c, e). \tag{5.3}$$

3) *Maximum-Distance* [42]: This assigns the *maximum* distance as the fitness value for c. In other words, this fitness function chooses the distance between c and its neighbor in E that is farthest away.

$$fitness(c, E) = \max_{e \in E} dist(c, e). \tag{5.4}$$

4) *Centroid-Distance* [43], [44]: This uses the distance between c and the centroid (center of the gravity) of E as the fitness value for c:

$$fitness(c, E) = dist\left(c, \frac{1}{|E|} \sum_{e \in E} e\right), \tag{5.5}$$

where $\frac{1}{|E|} \sum_{e \in E} e$ returns the centroid of E.

5) *Discrepancy* [45], [46]: This involves choosing the next test case such that it achieves the lowest discrepancy when added to E. Therefore, this fitness function of c can be defined as:

$$fitness(c, E) = 1 - Discrepancy\Big(E \bigcup \{c\}\Big). \tag{5.6}$$

6) *Membership-Grade* [47]: *Fuzzy Set Theory* [48] can be used to define some *fuzzy features* to construct a *membership grade function*, allowing a candidate with the highest (or threshold) score to be selected as the next test case. Chan et al. [47] defined some fuzzy features based on distance, combining them to calculate the membership grade function for candidate test cases. Two of the features they used are: the *Dynamic Minimum Separating Distance* (DMSD), which is a minimum distance between executed test cases, decreasing in magnitude as the number of executed test cases increases; and the *Absolute Minimum Separating Distance* (AMSD), which is an absolute minimum distance between test cases, regardless of how many test cases have been executed. During the evaluation of a candidate c against E, if $\forall e \in E$, $dist(c, e) \geq DMSD$, then selection of c will be strongly favored; however, if $\exists e \in E$ such that $dist(c, e) \leq AMSD$, then c will be strongly disfavored. The candidate most highly favored (with the highest membership grade) is then selected as the next test case.

7) *Restriction* [33], [47], [49]: This involves checking whether or not a candidate c violates the pre-defined restriction criteria related to E, denoted $restriction(c, E)$. The fitness function of c against E can be defined as:

$$fitness(c, E) = \begin{cases} 0, & \text{if } restriction(c, E) \text{is true,} \\ 1, & \text{otherwise.} \end{cases} \tag{5.7}$$

The random-candidate-construction criterion successively selects candidates from the input domain (according to uniform or non-uniform distribution) until one that is not restricted is identified. Three approaches to using *Restriction* in ART are:

- Previous studies [33]–[35], [50] have implemented restriction by checking whether or not c is located outside of all (equally-sized) *exclusion regions* defined around all test cases in E. In a 2-dimensional numeric input domain $\mathcal{D}$, for example, Chan et al. [33] used circles around each already selected test case as exclusion regions, thereby defining the $restriction(c, E)$ as:

$$\forall e \in E,\ dist(c, e) < \sqrt{\frac{R \cdot |\mathcal{D}|}{\pi \cdot |E|}}, \tag{5.8}$$

where R is the *target exclusion ratio* (set by the tester [51]), and $dist(c, e)$ is the *Euclidean distance* between c and e. In fact, Eq. (5.8) relates to the DMSD fuzzy feature [47].

- Zhou et al. [49], [52], [53] designed an acceptance probability P_β based on Markov Chain Monte Carlo (MCMC) methods [54] to control the identification of random candidates. Given a candidate c, the method generates a new candidate c' according to the applied distribution (uniform or non-uniform), resulting in $restriction(c, E)$ being defined as:

$$\mathcal{U} > P_\beta = \min\left\{\frac{P(X(c')=1|E)}{P(X(c)=1|E)}, 1\right\}, \quad (5.9)$$

where $\mathcal{U}$ is a uniform random number in the interval $[0, 1.0)$, $X(c)$ is the execution output of c ($X(c) = 1$ means that c is a failure-causing input, and $X(c) = 0$ means that it is not), and $P(X(c) = 1|E)$ represents the probability that c is failure-causing, given the set E of already executed test cases. According to Bayes' rule, we have:

$$P(X(c)=1|E) = P(E|X(c)=1)P(X(c)=1)/Z, \quad (5.10)$$

where Z is a normalizing constant. Assuming all elements in E are conditionally independent for a test output of c, we then have:

$$P(E|X(c)=1) = \prod_{e\in E} P(X(e)|X(c)=1). \quad (5.11)$$

As illustrated by Zhou et al. [49], [52], [53], $P(X(e)|X(c) = 1)$ is defined as:

$$P(X(e)=1|X(c)=1) = \exp(-dist(e,c)/\beta_1), \quad (5.12)$$

and

$$P(X(e)=0|X(c)=1) = 1-\exp(-dist(e,c)/\beta_1), \quad (5.13)$$

where β_1 is a constant. If one candidate is a greater distance from the non-failure-causing test cases than another candidate, then it has a higher probability of being selected as the next test case.

- Using *Fuzzy Set Theory* [48], Chan et al. [47] applied a dynamic threshold λ to determine whether or not a candidate was acceptable, accepting the candidate if its membership grade was greater than λ. If a predetermined number of candidates are rejected for being below λ, they adjusted the threshold according to the specified principles. It should be noted that any *Implementation 1* approach to choosing candidates can be transformed into *Implementation 2* by applying a threshold mechanism.

Six of the seven fitness functions (*Minimum-Distance*, *Average-Distance*, *Maximum-Distance*, *Centroid-Distance*, *Membership-Grade*, and *Restriction*) satisfy **Rationale 1**; one (*Discrepancy*) satisfies **Rationale 2**.

5.2 Partitioning-Based Strategy

The *Partitioning-Based Strategy* (PBS) divides the input domain into a number of subdomains, choosing one as the location within which to generate the next test case. Core elements of PBS, therefore, are to partition the input domain and to select the subdomain.

1: Set $E \leftarrow \{\}$
2: Randomly generate a test case tc from $\mathcal{D}$, according to uniform distribution
3: Add tc into E, i.e., $E \leftarrow E \bigcup \{tc\}$
4: **while** The stopping condition is not satisfied **do**
5: **if** The partitioning condition is triggered **then**
6: Partition the input domain $\mathcal{D}$ into m disjoint subdomains $D_1, D_2, \cdots, D_m$, according to the specific criterion *Partitioning-schema component*
7: **end if**
8: Choose a subdomain D_i according the specific criterion *Subdomain-selection component*
9: Randomly generate the next test case tc from D_i, according to uniform distribution
10: $E \leftarrow E \bigcup \{tc\}$
11: **end while**
12: Report the result and exit

Fig. 7. Framework pseudocode of the PBS category.

5.2.1 Framework

Fig. 7 presents a pseudocode framework for PBS, showing two main components: the *partitioning-schema*, and *subdomain-selection*. The partitioning-schema component defines how to partition the input domain into subdomains, and the subdomain-selection component defines how to choose the target subdomain where the next test case will be generated.

After partitioning, the input domain $\mathcal{D}$ will be divided into m disjoint subdomains $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$ $(m > 1)$, according to the partitioning-schema criteria: $\mathcal{D}_i \bigcap \mathcal{D}_j = \emptyset$ $(1 \leq i \neq j \leq m)$, and $\mathcal{D} = \mathcal{D}_1 \bigcup \mathcal{D}_2 \bigcup \cdots \bigcup \mathcal{D}_m$. Next, based on the subdomain-selection criteria, PBS chooses a *suitable* subdomain within which to generate the next test case.

5.2.2 Partitioning-Schema Component

Many different criteria can be used to partition the input domain, which can be achieved using either *static* or *dynamic* partitioning. Static partitioning [55]–[58] means that the input domain is divided before test case generation, with no further partitioning required once testing begins. Dynamic partitioning involves dividing the input domain dynamically, often at the same time that each new test case is generated. There have been many dynamic partitioning schemas proposed, including *random partitioning* [59], *bisection partitioning* [59], [60], and *iterative partitioning* [61], [62].

1) *Static partitioning* [58]: Static partitioning divides the input domain into a fixed number of equally-sized subdomains, with these subdomains then remaining unchanged throughout the entire testing process. This is simple, but influenced by the tester: testers need to divide the input domain before testing, and different numbers of subdomains may result in different ART performance.

2) *Random partitioning* [59]: Random partitioning uses the generated test case tc as the breakpoint to divide the input (sub-)domain $\mathcal{D}_i$ into smaller subdomains. This partitioning usually results in the input domain $\mathcal{D}$ being divided into subdomains of unequal size.

3) *Bisection partitioning* [59], [60]: Similar to static partitioning, bisection partitioning divides the input domain into equally-sized subdomains. However, unlike static partitioning, bisection partitioning *dynamically* bisects the input domain whenever the partitioning condition is triggered. There are a number of bisection partitioning implementations. Chen et al. [59], for example, successively bisected dimensions of the input domain; whenever the i-th bisection of a dimension resulted in 2^i parts, the $(i+1)$-th bisection was then applied to another dimension. Chow et al. [60] bisected all dimensions at the same time, with the input domain $\mathcal{D}$ being divided into 2^{i*d} subdomains (where d is the dimensionality of $\mathcal{D}$) after the i-th bisection. Bisection partitioning does not change existing partitions during bisection, only bisecting the subdomains in the next round.

4) *Iterative partitioning* [61], [62]: In contrast to bisection partitioning, iterative partitioning modifies existing partitions, resulting in the input domain being divided into equally-sized subdomains. Each round of iterative partitioning divides the entire input domain $\mathcal{D}$ using a new schema. After the i-th round of partitioning, for example, Chen et al. [61] divided the input domain into i^d subdomains, with each dimension divided into i equally-sized parts. Mayer et al. [62], however, divided only the largest dimension into equally-sized parts, leaving other dimensions unchanged, resulting in a dimension with j parts being divided into $j+1$ parts.

Although random partitioning may divide the input domain into subdomains with different sizes, the other three partitioning approaches result in equally-sized subdomains.

5.2.3 Subdomain-Selection Component

After partitioning the input domain $\mathcal{D}$ into m subdomains $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$, the next step is to choose the subdomain where the next test case will be generated. The following criteria can be used to support this subdomain selection process:

1) *Maximum size* [59]: Given the set T of previously generated test cases, among those subdomains $\mathcal{D}_i$ $(1 \leq i \leq m)$ without any test cases, the largest one is selected for generation of the next test case: $\forall j \in \{1, 2, \cdots, m\}$ satisfying $\mathcal{D}_i \bigcap T = \mathcal{D}_j \bigcap T = \emptyset$, and $|\mathcal{D}_i| \geq |\mathcal{D}_j|$.

2) *Fewest previously generated test cases* [59], [60]: Given the set T of previously generated test cases, this criterion chooses a subdomain $\mathcal{D}_i$ containing the fewest test cases: $\forall j \in \{1, 2, \cdots, m\}$, $|\mathcal{D}_i \bigcap T| \leq |\mathcal{D}_j \bigcap T|$.

3) *No test cases in target or neighbor subdomains* [58], [61], [62]: This ensures that the selected subdomain not only contains no test cases, but also does not neighbor other subdomains containing test cases.

4) *Proportional selection* [63]: Proportional selection uses two dynamic probability values, p_1 and p_2, to represent the likelihood that some (or all) elements of the failure region are located in the edge or center regions, respectively. Kuo et al. [63], for example, used two equally-sized subdomains in their proportional selection implementation, with each test case selected from either the edge or center region based on the value of p_1/p_2.

Three criteria (*fewest previously generated test cases*, *no test cases in target or neighbor subdomains*, and *proportional selection*) require that all subdomains be the same size; only one (*maximum size*) has no such requirement. Furthermore, two criteria (*maximum size* and *no test cases in target or neighbor subdomains*) generally select one test case per subdomain; the other two (*fewest previously generated test cases* and *proportional selection*) may select multiple test cases from each subdomain.

Intuitively speaking, three criteria (*maximum size*, *fewest previously generated test cases*, and *proportional selection*) follow **Rationale 2**. The *maximum size* criterion chooses the largest subdomain without any test cases as the target location for the next test case — test selection from a larger subdomain may have a better chance of achieving a good distribution of test cases. Similarly, the *fewest previously generated test cases*, and *proportional selection* criteria ensure that subdomains with fewer test cases have a higher probability of being selected. The third criterion (*no test cases in target or neighbor subdomains*) follows both **Rationale 1** and **Rationale 2**, choosing a target subdomain without (and away from) any test cases, thereby achieving a good test case distribution. Furthermore, because this criterion also avoids subdomains neighboring those containing test cases, the subsequently generated test cases generally have a minimum distance from all others.

5.3 Test-Profile-Based Strategy

The *Test-Profile-Based Strategy* (TPBS) [64] generates test cases based on a well-designed test profile (different from the uniform distribution), dynamically updating the profile after each test case selection.

5.3.1 Framework

Fig. 8 presents a pseudocode framework for TPBS. Because TPBS generates test cases based on the test profile, the core part of TBPS focuses on how to design the dynamic test profile. A test profile can be considered as the selection probability distribution for all test inputs in the input domain $\mathcal{D}$, with test cases in different locations having different probabilities. When a test case is executed without causing failure, its selection probability is then assigned a value of 0.

```
1: Set E ← {}
2: Randomly generate a test case tc from D, according to uniform distribution
3: Add tc into E, i.e., E ← E ⋃ {tc}
4: while The stopping condition is not satisfied do
5:     Adjust the test profile based on already selected test cases from E [Test-profile-adjustment component]
6:     Randomly generate the next test case tc based on adjusted test profile
7:     E ← E ⋃ {tc}
8: end while
9: Report the result and exit
```

Fig. 8. Framework pseudocode of the TPBS category.

5.3.2 Test-Profile-Adjustment Component

Based on the intuitions underlying ART [9], a test profile should be adjusted to satisfy the following [64]:

- The closer a test input is to the previously executed test cases, the *lower* the selection probability that it is assigned should be.
- The farther away a test input is from previously executed test cases, the *higher* the selection probability that it is assigned should be.
- The probability distribution should be dynamically adjusted to maintain these two features.

A number of test profiles exist to describe the probability distribution of test cases, including the *triangle profile* [65], *cosine profile* [65], *semicircle profile* [65], and *power-law profile* [66]. Furthermore, the *probabilistic ART* implementation [50] uses a similar mechanism to TPBS.

Because the test profiles use the location of non-failure-causing test cases when assigning the selection probability of each test input from the input domain, TPBS obviously follows **Rationale 1**: If a test input is farther away from non-failure-causing test cases than other candidates, it has a higher probability of being chosen as the next test case.

5.4 Quasi-Random Strategy

The *Quasi-Random Strategy* (QRS) [67], [68] applies *quasi-random sequences* to the implementation of ART. Quasi-random sequences are point sequences with low discrepancy and low dispersion: As discussed by Chen et al. [31], a set of points with lower discrepancy and dispersion generally has a more even distribution. Furthermore, the computational overheads incurred when generating n quasi-random test cases is only $O(n)$, which is similar to that of pure RT. In other words, QRS can achieve an even-spread of test cases with a low computational cost.

5.4.1 Framework

Fig. 9 presents a pseudocode framework for QRS, showing the two main components: *quasi-random-sequence-selection* and *randomization*. QRS first takes a quasi-random sequence to construct each point, then randomizes it to create the next test case according to the specific criterion. The main motivation for involving randomization in the process is that quasi-random sequences are usually generated by deterministic algorithms, which means that the sequences violate a core principle of ART: the randomness of the test cases.

```
1: Set E ← {}
2: while The stopping condition is not satisfied do
3:     Generate the next element ts from a given quasi-random sequence
       [Quasi-random-sequence-selection component]
4:     Randomize ts as the test case tc according to the specific criterion [Randomization component]
5:     E ← E ⋃ {tc}
6: end while
7: Report the result and exit
```

Fig. 9. Framework pseudocode of the QRS category.

5.4.2 Quasi-Random-Sequence-Selection Component

A number of quasi-random sequences have been examined, including Halton [69], Sobol [70], and Niederreiter [71]. In this section, we only describe some representative sequences for quasi-random testing.

1) *Halton sequence* [69]: The Halton sequence can be considered the d-dimensional extension of the Van der Corput sequence, a one-dimensional quasi-random sequence [72] defined as:

$$\phi_b(i) = \sum_{j=0}^{\omega} i_j b^{-j-1}, \tag{5.14}$$

where b is a prime number, $\phi_b(i)$ denotes the i-th element of the Van der Corput sequence, i_j is the j-th digit of i (in base b), and ω denotes the lowest integer for which $\forall j > \omega,\ i_j = 0$ is true. For a d-dimensional input domain, therefore, the i-th element of the Halton sequence can be defined as $(\phi_{b_1}(i), \phi_{b_2}(i), \cdots, \phi_{b_d}(i))$, where the bases, $b_1, b_2, \cdots, b_d$, are pairwise coprime. Previous studies [72], [73] have used the Halton sequence to generate test cases.

2) *Sobol sequence* [70]: The Sobol sequence can be considered a permutation of the binary Van der Corput sequence, $\phi_2(i)$, in each dimension [72], and is defined as:

$$Sobol(i) = \underset{j=1,2,\cdots,\omega}{\text{XOR}}(i_j \delta_j), \tag{5.15}$$

$$\delta_j = \underset{k=1,2,\cdots,r}{\text{XOR}}(\frac{\beta_k \delta_{j-k}}{2^j}) \oplus \frac{\delta_{j-r}}{2^{j+r}}, \tag{5.16}$$

where $Sobol(i)$ represents the i-th element of the Sobol sequence, i_j is the j-th digit of i in binary, ω denotes the number of digits of i in binary, and $\beta_1, \beta_2, \cdots, \beta_r$ come from the coefficients of a degree r primitive polynomial over the finite field. Previous studies [72]–[74] have used this sequence for test case generation.

3) *Niederreiter sequence* [71]: The Niederreiter sequence may be considered to provide a good reference for other quasi-random sequences: because all the other approaches can be described in terms of what Niederreiter calls (t, d)-sequences [71]. As discussed by Chen and Merkel [67], the Niederreiter sequence has lower discrepancy than other sequences. Previous investigations [67], [73] have used Niederreiter sequences to conduct software testing.

5.4.3 Randomization Component

The randomization step involves randomizing the quasi-random sequences into actual test cases. The following three representative methods illustrate this.

1) *Cranley-Patterson Rotation* [75], [76]: This generates a random d-dimensional vector $V = (v_1, v_2, \cdots, v_d)$ to permute each coordinate of the i-th point $T_i = (t_i^1, t_i^2, \cdots, t_i^d)$ to a new i-th point $P_i = (p_i^1, p_i^2, \cdots, p_i^d)$, where

$$p_i^j = \begin{cases} t_i^j + v_j, & \text{if } t_i^j + v_j < 1, \\ t_i^j + v_j - 1, & \text{if } t_i^j + v_j \geq 1. \end{cases} \tag{5.17}$$

2) *Owen Scrambling* [77]: Owen Scrambling applies the randomization process to the Niederreiter sequence (a (t, d)-sequence in base b). The i-th point in the sequence can be written as $T_i = (t_i^1, t_i^2, \cdots, t_i^d)$, where $t_i^j = \sum_{k=1}^{\infty} a_{ijk} b^{-k}$. The permutation process is applied to the parameter a_{ijk}

for each point according to some criteria. Compared with Cranley-Patterson rotation, Owen Scrambling more precisely maintains the low discrepancy and low dispersion of quasi-random sequences [68].

3) *Random Shaking and Rotation* [74], [78]: This first uses a non-uniform distribution (such as the *cosine distribution*) to shake the coordinates of each item in the quasi-random sequence into a random number within a specific value range. Then, a random vector based on the non-uniform distribution is used to permute the coordinates of all points in the sequence.

QRS generates a list of test cases (a quasi-random sequence) with a good distribution (including discrepancy [31]), indicating that it follows **Rationale 2**.

5.5 Search-Based Strategy

The *Search-Based Strategy* (SBS), which comes from *Search Based Software Testing* (SBST) [79], [80], uses search-based algorithms to achieve the even-spreading of test cases over the input domain. In contrast to other ART strategies, SBS aims to address the question: Given a test set E, of size N ($|E| = N$), due to limited testing resources, how can E achieve an even spread of test cases over the input domain, thereby enhancing its fault detection ability? SBS needs to assign a parameter (the number of test cases N) before test case generation begins.

5.5.1 Framework

Fig. 10 shows a pseudocode framework for SBS. Because ART requires that test cases that have some randomness, SBS generates an initial test set population PT (of size ps) where each test set (of size N) is randomly generated. A search-based algorithm is then used to iteratively evolve PT into its next generation. Once a stopping condition is satisfied, the best solution from PT is selected as the final test set. Two core elements of SBS, therefore, are the choice of search-based algorithm for evolving PT, and the evaluation (*fitness*) function for each solution. Because the fitness function is also involved in the evolution process (to evaluate the PT updates), we do not consider it a separate SBS component.

1: Set the number of test cases N
2: $E \leftarrow \{\}$
3: Generate an initial population of test sets $PT = \{T_1, T_2, \cdots, T_{ps}\}$, each of which is randomly generated with size N according to uniform distribution, where ps is the population size
4: **while** The stopping condition is not satisfied **do**
5: Evolve PT to construct a new population of test sets PT' by using a given search-based algorithm [*Evolution component*]
6: $PT \leftarrow PT'$
7: **end while**
8: $E \leftarrow$ the best solution of PT
9: Report the result and exit

Fig. 10. Framework pseudocode of the SBS category.

5.5.2 Evolution Component

A number of search-based algorithms have been used to evolve ART test sets, including the following:

1) *Hill Climbing* (HC) [81]: HC makes use of a single initial test set T, rather than a population of test sets PT (i.e., $ps = 1$). The basic idea behind HC is to calculate the fitness of T, and to shake it for as long as the fitness value increases. One HC fitness function is the minimum distance between any two test cases in T, where the distance is a specific Euclidean distance [82]:

$$fitness(T) = \min_{tc_i \neq tc_j \in T} dist(tc_i, tc_j). \tag{5.18}$$

2) *Simulated Annealing* (SA) [83]: Similar to HC, SA also only uses a single test set T ($ps = 1$). During each iteration, SA constructs a new test set T' by randomly selecting input variables from T with a mutation probability and modifying their values. The fitness values of both T and T' are then calculated. If the fitness of T' is greater than that of T, then T' is accepted as the current solution for the next iteration. If the fitness of T' is *not* greater than that of T, then the acceptance of T' is determined by a controlled probability function using random numbers and the temperature parameter adopted in SA. Bueno et al. [83] defined the fitness function of T as the sum of distances between each test case and its nearest neighbor:

$$fitness(T) = \sum_{tc_i \in T} \min_{tc_j \neq tc_i \in T} dist(tc_i, tc_j). \tag{5.19}$$

3) *Genetic Algorithm* (GA) [83]: GA uses a population of test sets rather than just a single one, and three main operations: *selection*, *crossover*, and *mutation*. GA first chooses the test sets for the next generation by assigning a selection probability — Bueno et al. [83] used a selection probability proportional to the fitness of T (calculated with Eq. (5.19)). The crossover operation then generates *offspring* by exchanging partial values of test cases between pairs of test sets, and then through mutation by randomly changing some partial values.

4) *Simulated Repulsion* (SR) [84]: Similar to GA, SR makes use of a population of test sets, PT, with each solution $T_i \in PT$ $(1 \leq i \leq ps)$ evolving independently from, and concurrently to, the other test sets. In each SR iteration, each test case from each solution T_i updates its value based on Newton mechanics with electrostatic force from Coulomb's Law. The principle of moving a test case $tc \in T_i$ is as follows:

$$tc_{new} = tc + (\overrightarrow{RF}(tc)/m), \tag{5.20}$$

where m is a constant (the mass of all test cases), and $\overrightarrow{RF}(tc)$ is the resultant force of tc, defined as:

$$\overrightarrow{RF}(tc) = \sum_{tc' \neq tc \in T_i} \frac{Q^2}{dist(tc, tc')^2}, \tag{5.21}$$

where Q is the current value of electric charge for the test cases.

5) *Local Spreading* (LS) [85]: Similar to HC and SA, LS also only uses a single initial test set T. LS successively moves each point $tc \in T$ that is allowed to move according to the following: tc's first and second nearest neighbors in T, tc_f and tc_s, are identified, and the corresponding distances, d_f

and d_s, are calculated. A direction of movement is identified related to tc_f. Then, tc is moved a small distance (related to $d_s - d_f$) in the identified direction, slightly increasing the minimum distance from tc to its nearest neighbor (the distance between tc and tc_f). These steps are repeated until there are no points remaining that can still move. LS effectively attempts to increase the minimum distance among all test cases in T, thereby producing a more evenly-spread test set.

6) *Random Border Centroidal Voronoi Tessellations* (RBCVT) [73]: RBCVT uses an initial test set T of size N, and makes use of Centroidal Voronoi Tessellations (CVT) [86] to achieve an even spread of the N test cases over the input domain, $\mathcal{D}$. It constructs N disjoint cells around the initial N test cases using a Voronoi tessellation with random border point set, $V_1, V_2, \cdots, V_N$, satisfying $i \neq j$, $V_i \bigcap V_j = \emptyset$; and $\bigcup_{i=1}^{N} V_i = \mathcal{D}$, where $1 \leq i, j \leq N$. Each cell V_i corresponds to a point $tc \in T$ such that

$$V_i = \{x \in \mathcal{D} | \forall tc' \neq tc \in T : dist(x, tc) < dist(x, tc')\}. \tag{5.22}$$

RBCVT then calculates the centroid of each Voronoi region to obtain N new points for the next generation and evolution.

Since SBS achieves an even spread of the N test cases over the input domain, many studies have used test suites generated by other ART approaches to replace the random test suites, to speed up the evolution process. Shahbazi et al. [73], for example, used RBCVT to improve the quality of test suites obtained from STFCS and QRS. Huang et al. [85] have also argued that it would be better to use adaptive random test suites than random test suites as the input for LS.

5.6 Hybrid Strategies

Hybrid Strategies (HSs) aim at improving the testing effectiveness (such as fault detection capability) or efficiency (such as test generation cost) by combining multiple ART approaches.

5.6.1 STFCS + PBS

The STFCS + PBS hybrids aim to enhance the effectiveness of either STFCS or PBS.

From the perspective of STFCS enhancement, Chen et al. [87], [88], when generating the m-th test case, divided the input domain $\mathcal{D}$ into m disjoint, equally-sized subdomains, $\mathcal{D}_1, \mathcal{D}_2, \cdots, \mathcal{D}_m$, from the edge to the center of $\mathcal{D}$, such that: $\mathcal{D} = \bigcup_{i=1}^{m} \mathcal{D}_i$; $D_i \bigcap D_j = \emptyset$ for $1 \leq i \neq j \leq m$; and $|\mathcal{D}_1| = |\mathcal{D}_2| = \cdots = |\mathcal{D}_m|$. Next, for the STFCS random-candidate-construction component, they generated random test cases in those subdomains not already containing test cases. Mayer [89] used bisection partitioning to control the STFCS test-case-identification component, only checking the distance from a candidate c to points in its neighboring regions, instead of to all points. These methods could significantly reduce the STFCS computational overheads, for both FSCS and RRT. Mao et al. [90] proposed a similar method, *distance-aware forgetting*, to reduce the FSCS computational overheads, but they used static, rather than bisection, partitioning. Chow et al. [60] proposed a new efficient and effective method called *ART with divide-and-conquer* that independently applies STFCS to each subdomain (using bisection partitioning). Previous studies [55]–[57] have combined STFCS with static partitioning, using the concept of *mirroring* to reduce the computational costs. Enhancements to mirroring have included a revised distance metric [91], and dynamic partitioning with new mirroring functions [92]. Chan et al. [47] applied bisection partitioning to each dimension of the input domain, then checking the amount of executed test cases in each subdomain: candidates in subdomains with fewer executed test cases were then more likely to be selected.

Regarding the enhancement of PBS, Chen and Huang [93] applied the test-case-identification component to improving the effectiveness of PBS with random partitioning, selecting test cases based on the principle of Minimum-Distance and Restriction. Mayer [94] used a similar mechanism to improve the effectiveness of PBS with bisection partitioning. Mao and Zhan [95] also used this mechanism to enhance PBS by bisection partitioning, but instead of Euclidean distance, they used the coordinate distance to boundaries (*boundary distance*). Similarly, Mayer [96], [97], for PBS with random and bisection partitioning, used exclusion regions in a possible subdomain to generate a new test case. Mao [98], to overcome the drawbacks of random partitioning, proposed a new partitioning schema, *two-point partitioning*, based on the STFCS test-case-identification component: When generating a new test case tc, it randomly chooses two candidates from the subdomain that needs to be partitioned, and then uses the midpoint of tc and the farthest candidate as the break point to partition the subdomain.

In addition to the hybrid methods listed above, other, new ART techniques have been proposed based on other combinations of different concepts. Chen et al. [99], for example, introduced a new test-case-identification criterion (identifying the test case that is more adjacent to the subdomain centroid), and combined it with PBS with bisection partitioning to form a new technique: *ART by balancing*. Mayer [100] proposed a new approach, *lattice-based ART*, that uses bisection partitioning to divide the input domain for lattice generation. It then generates test cases by permuting the lattices within a restricted region. Chen et al. [101] enhanced Mayer's lattice-based ART [100] by refining the restricted regions for each lattice. Sabor and Mohsenzadeh [102], [103] proposed an enhanced version of Chen and Huang's method [93] by including an *enlarged input domain* [104].

5.6.2 STFCS + SBS

The STFCS + SBS hybrids either enhance STFCS, or represent new methods. Tappenden and Miller [105] proposed *Evolutionary ART*, a new method that aims to construct an evolved test set for the STFCS test-case-identification. The method initially generates a fixed-size random test set, according to a uniform distribution. Until a stopping condition is satisfied (for example, 100 generations have been completed [105]), each iteration uses an evolutionary algorithm, a *Genetic Algorithm* (GA), to evolve the test set. The fitness function used during the evolution stage is the

same as Eq. (5.2), and the candidate with the highest fitness value is then selected as the next test case.

Iqbal et al. [106] combined STFCS with SBS to produce a hybrid that initially uses GA to generate test cases, but if no fitter test cases are found after running a number of iterations, then the algorithm switches to FSCS to generate the following test cases.

5.6.3 TPBS + PBS or STFCS

The TPBS + PBS or STFCS hybrids aim to augment the TPBS test profiles using PBS or STFCS principles. Liu et al. [107] proposed three methods to design test profiles, based on STFCS restriction (*exclusion*), and on PBS subdomain-selection criteria (*maximum size* [59] and *least number of previously generated test cases* [59], [60]). Using *exclusion*, all points inside the exclusion regions should have no chance of being selected as test cases: their probability of selection is 0. Using the *maximum size* [59], all points inside the largest subdomain have a probability to be chosen as test cases, and all other points (those in other subdomains) have no chance. Similarly, when using the *least number of previously generated test cases* [59], [60], all points within the subdomains with the least number of previously generated test cases have a chance to be selected, and all others have no chance.

5.7 Strengths and Weaknesses

Previous studies [18] have confirmed that ART is more effective than RT in general, according to several different evaluations. As discussed by Chen et al. [108], however, both favorable and unfavorable conditions exist for ART. In this section, therefore, we summarize the strengths and weaknesses of ART.

5.7.1 Strengths

ART outperforms RT from the following perspectives:

1) *Test case distribution*: ART generally delivers a more even distribution of test cases across the input domain than RT. All ART approaches deliver better test case dispersion [31] than RT [31], [39], [45], [46], [64], [74], [78], [107]. When the input domain dimensionality d is low (d is equal to 1 or 2), all ART approaches generate test cases with a more even spread over the input domain than RT, in terms of test case discrepancy [31]. However, as the dimensionality increases, some ART approaches have worse performance than RT, including FSCS [31], [39], [45], [46], RRT [31], [107], and TPBS [64]. Nevertheless, even when the dimensionality is 3 or 4, a number of ART approaches do still have better discrepancy than RT, including PBS [31], [107] and QRS [74].

2) *Fault detection capability*: It is natural that ART should have better fault detection ability than RT when the failure region is clustered — ART was specifically designed to make use of this clustering information. Chen et al. [108] investigated the factors impacting on ART fault detection ability, identifying a number of favorable conditions for ART, including: (a) when the failure rate is small; (b) when the failure region is compact; (c) when the number of failure regions is low; and (d) when a predominant region exists among all the failure regions. When any of these conditions are satisfied, ART generally has better fault detection performance than RT. Even when none of the conditions are satisfied, ART can achieve comparable fault detection to RT.

3) *Code coverage*: Studies have shown that ART achieves greater *structure-based code coverage* than RT for the same number of test cases [83], [84], [109], [110]. Bueno et al. [83], [84] have observed that SBS outperforms RT for *data-flow coverage* [111] (including *all-uses coverage* and *all-du-paths coverage*). Chen et al. [109], [110] have reported that FSCS is more effective than RT for both *control-flow coverage* [112] (including *block coverage* and *decision coverage*), and data-flow coverage (*c-uses coverage* and *p-uses coverage* [111]).

4) *Reliability estimation and assessment*: For the same number of test cases, ART has greater code coverage than RT [83], [84], [109], [110]. It has also been observed that coverage can be used to improve the effectiveness of software reliability estimation [113]. Compared with RT, therefore, ART should enable better software reliability estimation, and higher confidence in the reliability of the SUT, even when no failure is detected. Unfortunately, this characteristic (strength) of ART was obtained from the perspective of theoretical results rather than empirical studies, which means that no ART studies have yet investigated the reliability estimation and assessment.

5) *Cost-effectiveness*: The cost-effectiveness of testing considers both effectiveness (e.g., fault detection) and efficiency (including test case generation and execution time). ART cost-effectiveness has often been examined using the *F-time* [61], which is the amount of computer execution time required to detect the first failure (including the time for both generation and execution of test cases). Because ART involves additional computation to evenly spread the test cases over the input domain [17], [18], ART should naturally take more time than RT to generate the same number of test cases, suggesting that it may have worse cost-effectiveness than RT. However, studies [18], [92], [114]–[116] have shown that compared with RT, ART typically requires less time to identify the first failure (*F-time*) — therefore, ART can be more cost-effective than RT. In general, three main conditions can result in ART achieving a better cost-effectiveness than RT: (a) ART using fewer test cases than RT to detect the first failure (*F-measure*); (b) the computational overhead of the ART approach being acceptable (comparable or slightly higher than that of RT); or (c) the combined program execution and test setup time being more than the time required by ART to generate a test case.

5.7.2 Weaknesses

There are three main challenges associated with some ART approaches: *boundary effect*, *computational overheads*, and *high dimension problem*.

1) *Boundary effect* [117]: Some ART approaches tend to generate more test cases near the boundary than near the input domain center, a situation known as the *boundary effect*. One reason for the boundary effect, as explained in the context of RRT [33]–[36], is that test cases cannot be generated outside the boundary, thus reducing the number of sources of restriction from close to boundary regions. Both FSCS and RRT have been shown to suffer from the boundary effect, especially when the failure rate and dimensionality are high [117].

A number of attempts have been made to address the boundary effect. Some studies increased the selection probability of candidates from the input domain center over those from the boundary [37]–[39], [63], [87], [88]. Other studies have removed the boundaries themselves, either by joining boundaries [82], [118], [119], or by extending the input domain beyond the original boundaries [104], [117], [120]. Chen et al. [99] preferred to choose test cases close to the input domain centroid. Mayer [121] initially selected test cases within a small region around the center of the input domain, extending the region if no failures were identified.

2) *Computational overheads* [17], [18]: ART approaches typically incur heavier computational costs than RT to generate test cases. This is particularly the case for some ART approaches, such as FSCS, and RRT. When testing, the actual test execution time can be an important factor that may mitigate the computational overheads: When the test execution time is very long, for example, the ART computational costs may be more acceptable. However, because the test execution time depends mainly on characteristics of the SUT, and not on the test generation, we do not discuss it here.

The time complexity of FSCS and RRT are of the order of $O(n^2)$ and $n^2 \log n$, respectively (where n is the number of generated test cases) [32], [122]. The PBS and QRS techniques of ART are significantly more efficient than others (such as STFCS and SBS). Many hybrid approaches have been developed to reduce the overheads incurred by FSCS and RRT. Other techniques for ART overhead reduction have also been explored. Chan et al. [36], [123], for example, used a square exclusion region version of RRT to reduce the distance calculation overheads (though not the algorithm's complexity, $O(n^2)$). Chen and Merkel [124] applied Voronoi diagrams [125] to reduce the FSCS distance calculations, lowering the complexity from $O(n^2)$ to $O(n^{\frac{4}{3}})$. *Mirroring* [55]–[57], [92] has been used to directly generate *mirror test cases* of a *source test case* based on a principle of symmetry of subdomains. Although the time complexity for FSCS mirroring can still be $O(\frac{n^2}{m^2})$ [55]–[57] (where m is the number of subdomains), an enhanced mirroring technique can reduce this to $O(n)$ [92]. Shahbazi et al. [73] have also proposed a linear-order ($O(n)$) ART approach: a fast search algorithm for RBCVT (RBCVT-Fast).

While the overhead-reduction approaches listed above can only be applied to numeric input domains, *forgetting* [126], which reduces overheads by omitting some previous test cases from calculations, has no such limitation. Based on the *Category-Partition Method* [127], Barus et al. [128] have also recently introduced a linear-order FSCS algorithm using the test-case-identification with *Average-Distance* for nonnumeric inputs.

3) *High dimension problem* [31], [129]: It has been observed that the effectiveness of some ART approaches may decrease when the number of dimensions of the input domain increases, due to the *curse of dimensionality* [130]. Because the center of a high dimensional input domain has a higher probability of being a failure region than the boundary [99], [121], it has been suggested that the boundary effect may impact (or even cause) the high dimension problem. Approaches for addressing the boundary effect [37]–[39], [63], [82], [87], [88], [99], [104], [117]–[121], [131], therefore, may also help to alleviate the high dimension problem. However, because their effectiveness is not constant across dimensions, although they may alleviate, current approaches do not *solve* the dimensionality problem [81]. Furthermore, finding a solution with consistent effectiveness across all dimensions seems unlikely [81], so some decrease in effectiveness in higher dimensions may need to be tolerated. Nevertheless, Schneckenburger and Schweiggert [81] have combined *Hill Climbing* with *continuous distance* [122] to produce a search-based ART approach that (slightly) reduces the dependency on dimensionality.

Summary of answers to RQ2:

1) *Based on different concepts for the even-spreading of test cases, the various ART approaches can be classified into the following six categories: Select-Test-From-Candidates Strategy (STFCS); Partitioning-Based Strategy (PBS); Test-Profile-Based Strategy (TPBS); Quasi-Random Strategy (QRS); Search-Based Strategy (SBS); and Hybrid Strategies (HSs).*
2) *For each of the first five categories, a framework has been presented showing the basic steps involved.*
3) *Compared with RT, ART generally performs better when certain conditions hold (identified as "favorable conditions" for ART), according to test case distribution, fault detection capability, code coverage, reliability estimation and assessment, and cost-effectiveness.*
4) *ART suffers from three main weaknesses: boundary effect, computational overheads, and the high dimension problem.*

6 Answer to RQ3: In what domains and applications has ART been applied?

Of the 140 papers in our survey, 80 involved application of ART to specific testing problems. Fig. 11 presents the distribution of the ART applications, showing that 76% of studies focused on various testing environments (or programs), and 24% involved application of ART to other testing techniques.

6.1 Application Domains

Based on the classification shown in Fig. 11, numeric programs (47%) have been the most popular domains for ART application, followed by object-oriented programs (7%), embedded systems (6%), web services and applications (4%), configurable systems (4%), and simulations and modelling (3%). 5% of the papers revealed other application domains, including mobile applications and aspect-oriented programs.

6.1.1 Numeric Programs

Chen et al. [32] applied ART to the testing of twelve open-source numeric analysis programs from Numerical Recipes [132] and ACM Collected Algorithms [133], written in C or C++. These programs have also been widely used in other ART studies, and, in addition to C and C++, have also been implemented in Java. Zhou et al. [52] applied ART to three other numeric programs from the Numerical

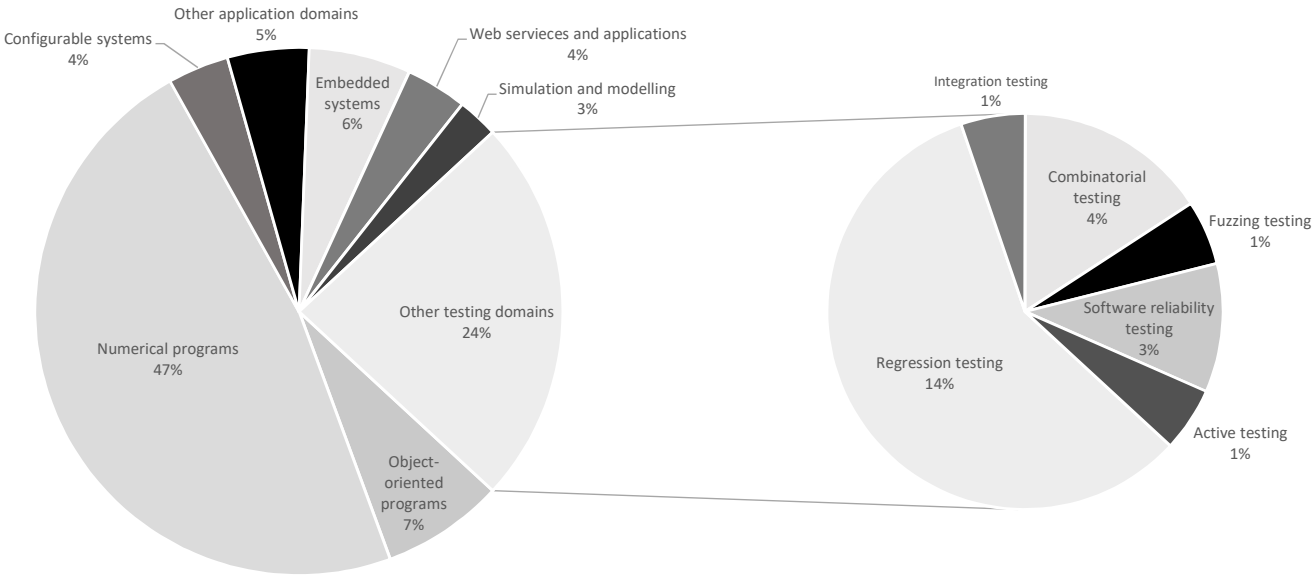


Fig. 11. Application distribution of ART.

Recipes [132]. Arcuri and Briand [134] conducted experiments on a further nine numeric programs for basic algorithms [135] and for mathematical routines from the Numerical Recipes [132]. Chen et al. [109] used ART with two numeric programs from the GNU Scientific Library [136], and one from the Software-artifact Infrastructure Repository (SIR) [137]. Walkinshaw and Fraser [138] investigated six units within the Apache Commons Math framework [139] and two units within JodaTime [140].

6.1.2 Object-Oriented Programs

Ciupa et al. [40], [141] defined a new similarity metric for object-oriented (OO) inputs, and integrated it into ART (ARTOO). They compared ARTOO with RT using eight real-life, faulty OO programs from the EiffelBase Library [142]. Lin et al. [114] used their ART approach on six OO programs containing manually-seeded faults — five of these programs were from the Apache Common library [143], and one was a wide-area event notification system, Siena. Chen et al. [144] also proposed a new similarity metric for OO inputs, using it to apply ART to 17 OO programs (five C++ libraries and 12 C# programs) from the following open-source repositories: Codeforge [145], Sourceforge [146], Codeplex [147], Codeproject [148], and Github [149]. Putra and Mursanto [44] compared two ART techniques applied to eight OO programs, written in Java, from the Apache Common library [143]. Jaygarl et al. [150] evaluated different RT and ART techniques applied to four open-source OO programs (written in Java).

6.1.3 Configurable Systems

Chen et al. [110] compared the code coverage achieved by ART and RT using ten UNIX utility programs that can be considered configurable systems — they are influenced by different configurations or factors, obtained using the *Category-Partition Method* (CPM) [127]. Huang et al. [151] applied ART to *combinatorial input domains* (*configurable input domains*), testing five small C programs [152], and four versions of another configurable system, Flex (a fast lexical analyzer), from the SIR [137]. They also identified the configurable input domains using CPM. Barus et al. [128] proposed an efficient ART approach, and applied it to 14 configurable systems and programs — seven Siemens programs from the SIR [137]; six UNIX utilities; and one regular expression processor from the GNU Scientific Library [136].

6.1.4 Web Services and Applications

Tappenden and Miller [153] applied an evolutionary ART algorithm to cookie collection testing, applying it to six open-source web applications written in C# and PHP. Selay et al. [115] used ART in image comparisons when testing a set of real-world, industrial web applications. Chen et al. [154] developed a system to test web service vulnerabilities that generates ART test cases based on *Simple Object Access Protocol* (SOAP) messages: twenty web services (both open-source and specifically written services) were examined in their study.

6.1.5 Embedded Systems

Hemmati et al. [155]–[157] applied ART to two industrial embedded systems: a core subsystem of a video-conference system, implemented in C; and a safety monitoring component of a safety-critical control system written in C++. Arcuri et al. [106], [158] compared RT, ART, and search-based testing using a real-life real-time embedded system. This very large and complex seismic acquisition system, implemented in Java, interacts with many sensors and actuators.

6.1.6 Simulations and Models

Matinnejad et al. [159] applied an ART approach to test three Stateflow models of mixed discrete-continuous controllers: two industrial models from Delphi [160], Supercharger Clutch Controller (SCC) and Auto Start-Stop Control (ASS); and one public domain model, Guidance Control System (GCS), from Mathworks [161]. Sun et al. [162] proposed an

enhanced ART approach for testing *Architecture Analyze and Design Language* (AADL) models [163], reporting on a case study applying it to an Unmanned Aerial Vehicle (UAV) cruise control system, which includes three sensor devices (radar, GPS, and speed devices), and two subsystems (read data calculation, and flight control systems).

6.1.7 Other Domains

Parizi and Ghani [164] conducted a preliminary study of ART for aspect-oriented programs, identifying some potential research directions. Shahbazi and Miller [165] investigated the application of ART to programs with string inputs, comparing the performance of different ART approaches on 19 open-source programs. Liu et al. [116] used ART to test mobile applications, providing a new distance metric for event sequences. Their study examined six real-life mobile applications implemented in Java. Koo and Park [166] investigated ART for SDN (*Software-Defined Networking*) OpenFlow switches, aiming to generate test packages for switches.

6.1.8 Summary of Main Results for Application Domains

Table 6 summarizes the main results for some of the original studies involving application of ART to different domains ("NR" indicates that some details were not reported in the original paper). Some of the studies did not explicitly provide results of the comparison between RT and ART, and so have been omitted from the table. Information about the programs tested in each application domain is available in Table A.1, in the appendix.

In Table 6, the columns "%Effectiveness Improvement" and "%Efficiency Improvement" show the percentage improvements of testing effectiveness and efficiency of ART over RT, respectively. Given an evaluation metric M (for example, to measure testing effectiveness or efficiency), if M_a represents the value for ART, and M_r the value for RT, then the percentage improvement (of ART over RT) can be calculated as:

- $100 * (M_r - M_a)/M_r$, if lower values mean improvements; and
- $100 * (M_a - M_r)/M_r$, if greater values mean improvement.

Because of the additional computations involved in ART, it is intuitive that it should take longer than RT to generate the same number of test cases. The "Efficiency" data here, therefore, refers to the time taken to achieve the stopping criterion — for example, the time taken to detect the first failure (the *F-time* [61]), which includes the combined generation and execution time of all test cases executed before causing the first failure). The "Efficiency" can be considered the *cost-effectiveness* metric.

Based on Table 6, we have the following observations:

- Across all application domains, ART usually achieves better testing effectiveness than RT, especially for numeric and object-oriented programs.
- For a particular application domain, due to the different characteristics of the various programs, ART may perform differently with different programs.
- Many of the original studies surveyed (including the numeric programs and configurable systems) only provided the information for "%Effectiveness Improvement", not for "%Efficiency Improvement".
- Some studies found ART to be more cost-effective than RT (e.g., [114], [150], [166]), with some observing ART to require less time than RT to identify the first failure (e.g., [40], [144]).

6.2 Other Testing Applications

A number of the surveyed studies considered ART as a strategy to support another testing method, with a goal of enhancing the effectiveness and applicability of that target method. Fig. 11 shows that the most popular target testing application is regression testing [167] (14%), followed by combinatorial testing [168] (4%), software reliability testing [169] (3%), fuzzing [170] (1%), integration testing [171] (1%), and active testing [172] (1%).

6.2.1 Regression Testing

Adaptive random sequences [15] have been extensively applied to regression testing, including for both prioritization and selection [167]. Jiang et al. [42] first used adaptive random sequences (obtained with FSCS [9]) to prioritize regression test suites, proposing a family of approaches that use code coverage to measure the distance between test cases. Other studies [173]–[175] have also used code coverage to guide test case prioritization, but with different distance measures or different ART approaches. Jiang & Chan [176], [177] proposed a family of novel input-based randomized local beam search techniques to prioritize test cases. Chen et al. [178] proposed a method using two clustering algorithms to construct adaptive random sequences for object-oriented software. Zhang et al. [179], based on work on string test case prioritization [180], introduced a method to construct adaptive random sequences using string distance metrics. They later used a different distance metric, based on CP-M [127], to propose another method for prioritizing test cases [181]. Huang et al. [182] applied adaptive random prioritization to interaction test suites for combinatorial testing [168]. Zhou [183] used the same code coverage distance information as Zhou et al. [52] to support regression test case selection. Chen et al. [184] extracted tokens reflecting fault-relevant characteristics of the SUT (such as statement characteristics, type and modifier characteristics, and operator characteristics), and used these tokens to represent test cases as vectors for prioritizing programs for C compilers.

Previous investigations have shown that although adaptive random sequences generally incur higher computational costs than random sequences, they are usually significantly more effective in terms of fault detection. Furthermore, adaptive random sequences are also sometimes comparable to test sequences obtained by traditional regression testing techniques [185], in terms of both testing effectiveness and efficiency.

6.2.2 Combinatorial Testing

Huang et al. [186] used two popular ART techniques (FSCS and RRT) to construct an effective test suite (a *covering array* [187]) for combinatorial testing [168]. Nie et al. [188]

TABLE 6
Summary of Main ART Results for each Application Domain

Application Domain	Original study	Method	Program Name	%Effectiveness Improvement	%Efficiency Improvement
Numeric Programs	Chen et al. [32]	FSCS	Airy	42.14%	NR†
			Bessj	41.83%	
			Bessj0	42.24%	
			Cel	47.55%	
			El2	52.02%	
			Erfcc	44.30%	
			Gammq	11.38%	
			Golden	1.67%	
			Plgndr	34.09%	
			Probks	45.25%	
			Sncndn	1.18%	
			Tanh	45.00%	
	Zhou et al. [52]	MCMC-Random Testing	Bessel	93.02%	
			Ellint	28.28%	
			Laguerre	90.67%	
	Chen et al. [109]	FSCS/ECP-FSCS*	Cubic	0.42% – 3.48%	
			Quadratic	0.20% – 4.95%	
			Tcas	-1.99% – 4.72%	
	Walkinshaw and Fraser [138]	FSCS	BesselJ	0.02%	
			Binomial	0.59%	
			DaysBetween	-0.94%	
			DerivativeSin	-2.21%	
			Erf	-0.62%	
			Gamma	-0.63%	
			PeriodToWeeks	8.84%	
			Romberg Integrator	-0.19%	
Object-Oriented Programs	Ciupa et al. [40]	FSCS	Action_sequence	91.49%	-38.29%
			Array	48.95%	-527.46%
			Arrayedlist	93.77%	-7.71%
			Boundedstack	20.91%	-1017.90%
			Fixedtree	28.01%	6.52%
			Hashtable	78.74%	-149.67%
			Linkedlist	61.38%	-442.39%
			String	82.09%	41.07%
	Lin et al. [114]	FSCS	Math.geometry	87.36%	86.31%
			Math.util	85.48%	99.05%
			Lang	20.96%	64.87%
			Lang.text	91.05%	89.78%
			Collections.list	86.85%	87.05%
			Siena	92.13%	82.80%
	Chen et al. [144]	FSCS + Forgetting	CCoinBox	2.46% – 63.83%	-725.00% – -144.44%
			Calendar	0.00% – 76.41%	-210.53% – -13.46%
			Stack	0.00% – 56.02%	-230.86% – -63.41%
			Queue	0.00% – 55.21%	-118.67% – -5.81%
			WindShieldWiper	5.11% – 67.36%	-457.89% – -70.97%
			SATM	4.99% – 62.28%	-325.81% – -94.12%
			BinarySearchTree	10.94% – 77.93%	-428.24% – -93.53%
			RabbitAndFox	0.56% – 76.30%	-354.41% – -83.93%
			WaveletLibrary	2.88% – 72.09%	-153.24% – -3.53%
			BackTrack	3.09% – 76.87%	-128.57% – -1.05%
			NSort	3.61% – 60.20%	-593.75% – -126.53%
			SchoolManagement	3.14% – 79.69%	-510.53% – -109.64%
			EnterpriseManagement	0.00% – 75.50%	-233.64% – -80.30%
			ID3Manage	14.66% – 70.58%	-690.00% – -92.68%
			IceChat	13.00% – 122.96%	-463.93% – -26.47%
			CSPspEmu	8.88% – 92.79%	-169.94% – -25.36%
			Poco-1.4.4: Foundation	6.72% – 86.12%	-144.67% – -20.80%
	Jaygarl et al. [150]	FSCS	Apache Ant	58.16%	65.87%
			ASM	85.99%	92.14%
			ISSTA Containers	41.26%	43.82%
			Java Collections	89.07%	98.76%
Configurable Systems	Huang et al. [151]	FSCS	Count	11.16% – 21.41%	NR
			Series	9.93% – 21.28%	
			Tokens	9.69% – 20.14%	
			Ntree	11.19% – 21.73%	
			Nametbl	9.03 – 20.91%	
			Flex	4.36% – 22.80%	
	Barus et al. [128]	FSCS FSCS + Mirroring	Cal	31.91% – 87.05%	
			Comm	56.28% – 88.25%	
			Grep	-114.76% – 72.60%	
			Look	-133.58% – 58.40%	
			Printtokens	37.61% – 64.59%	
			Printtokens2	-17.69% – 65.36%	
			Replace	-309.64% – 69.68%	
			Schedule	-403.38% – 87.84%	
			Schedule2	-2365.22% – 80.29%	
			Sort	-52.01% – 84.21%	
			Spline	-207.42% – 79.68%	
			TCAS	-129.07% – 65.74%	
			Totinfo	-264.70% – 62.64%	
			Uniq	-30.14% – 86.31%	

Application Domain	Original study	Method	Program Name	%Effectiveness Improvement	%Efficiency Improvement
Web Services and Applications	Selay et al. [115]	FSCS + Forgetting FSCS + Mirroring	RWWA1-7	-2.22% – 16.16%	-1.82% – 3.63%
Embedded Systems	Iqbal et al. [106]	FSCS	IC	3.00%	NR
Simulation and Modelling	Sun et al. [162]	FSCS	UAV	5.00% – 43.00%	
Other Domains (String Programs)	Shahbazi and Miller [165]	FSCS GA Multi-Objective GA	Validation	4.50% – 94.60%	
			PostCode	22.10% – 137.00%	
			Numeric	43.20% – 458.00%	
			DateFormat	43.40% – 462.10%	
			MIMEType	-19.30% – 12.00%	
			ResourceURL	-17.20% – 15.60%	
			URI	-22.50% – -3.30%	
			URN	-47.60% – 15.60%	
			TimeChecker	-27.40% – -0.60%	
			Clocale	38.30% – 321.00%	
			Isbn	-28.20% – 65.60%	
			BIC	4.40% – 103.80%	
			IBAN	23.80% – 90.40%	
Other Domains (SDN OpenFlow Switches)	Koo and Park [166]	FSCS	Open vSwitch	66.67%	62.55%

*ECP-FSCS is *Edge-Center-Partitioning based FSCS*, which combines FSCS with static partitioning [109].
†"NR" indicates that some details were not reported in the original paper.

investigated covering arrays constructed by RT, ART, and combinatorial testing, in terms of their ability to identify interaction-triggered failures.

Overall, ART requires far fewer combinatorial test cases to construct covering arrays than RT, and can also detect more interaction-triggered failures for the same number of test cases [186]. ART also performs comparably to traditional combinatorial testing, especially for identifying interaction failures caused by a large number of influencing parameters [188].

6.2.3 Reliability Testing

Liu and Zhu [189] used mutation analysis [23] to evaluate the reliability of ART's fault-detection ability by analyzing the variation in fault detection, concluding that it is more reliable than RT. Cotroneo et al. [190] evaluated the reliability improvement of two ART techniques, FSCS [9] and evolutionary ART [105]: Based on the same operational profile, for the same test budget, they found that ART had comparable delivered reliability [3] to traditional operational testing. For a given reliability level, however, for the same operational profile, ART typically requires significantly fewer test inputs, compared to traditional operational testing techniques.

6.2.4 Active, Fuzzing, and Integration Testing

Based on FSCS-ART [9], Yue et al. [191] proposed two input-driven active testing approaches for multi-threaded programs, with experimental evaluations indicating that the proposed methods are more cost-effective than traditional active testing. Similarly, Sim et al. [192] applied FSCS-ART [9] to fuzzing the Out-Of-Memory (OOM) Killer on an embedded Linux distribution, with results showing that their ART approach for fuzzing requires significantly fewer test cases than RT to identify an OOM Killer failure. Shin et al. [193] proposed an algorithm based on normalized ART for integration and regression tests of units integrated with a front-end software module. The related simulation studies showed that the proposed ART method could be useful for the integration tests.

Summary of answers to RQ3:

1) *ART has been applied in many different application domains, including: numeric programs, object-oriented programs, embedded systems, configurable systems, and simulations and models. According to the surveyed studies, ART generally has better testing effectiveness than RT for most application domains (with respect to various evaluation metrics, including the number of test case executions necessary to identify the first failure).*
2) *ART has also been used to augment or enhance other testing techniques, such as regression testing, combinatorial testing, and reliability testing. Similarly, the ART enhancement is generally better than the RT version.*
3) *For each application domain, while ART is generally more effective than RT with respect to different evaluation metrics, it may still be less cost-effective overall. Nevertheless, some ART approaches do provide better cost-effectiveness than RT.*

7 Answer to RQ4: How have empirical evaluations in ART studies been performed?

7.1 Distribution of Empirical Evaluations

Of the 140 primary studies examined, 131 (94%) involved empirical evaluations. Fig. 12 shows the distribution of the empirical evaluations in these 131 studies, with 58 papers (44%) having only simulations, 53 (41%) only experiments with real programs, and 20 (15%) containing both simulations and experiments. It can be observed that the number of studies containing only simulations is comparable to the number with only experiments.

7.2 Simulations

Simulations that attempt to construct failures in the numeric input domain to resemble real testing situations have frequently been used to evaluate ART techniques. As discussed by Chen et al. [108], three factors[5] are typically considered in the design of simulations: the dimensionality (d) of the input domain; the failure rate (θ); and the failure pattern.

5. Simulations examining modeling or distance-calculation errors [194] have also appeared, but are very rare.

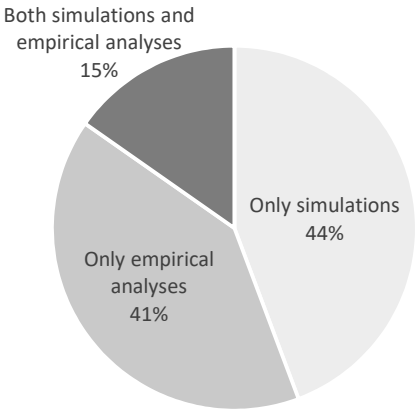


Fig. 12. Distribution of empirical evaluations.

In spite of their popularity, simulations have limitations as evaluation tools, including that: (1) they are mostly only used to simulate numeric input domains; (2) the assumed failure patterns may not be realistic; and (3) simulations that do account for test execution time often have very similar execution times, which means that the simulations are effectively comparing *generation*, rather than execution, time, which may decrease their applicability in practice.

In general, primary studies involving simulations assume the input domain $\mathcal{D}$ to be $[0, 1.0)^d$ — a unit hypercube (each dimension of $\mathcal{D}$ ranging from 0.0 to 1.0). Excluding those few studies using simulations for a special testing environment (such as for combinatorial testing [168]), 73 of the 78 papers involving simulations used numeric input domains. In this section, we review these 73 studies according to the three main simulation design factors (d, θ, and the failure pattern).

7.2.1 Dimensionality Distribution

Fig. 13 shows the input domain dimensionality (d) distribution across the 73 primary studies. It can be observed that d ranges from 1 to 15, with $d = 2$ being the most popular (96%), followed by $d = 3$ (42%), $d = 4$ (32%), and $d = 1$ (27%). Only a maximum of four papers (5%) involved each d greater than 4. In other words, most simulations have been conducted in low dimensional ($d \leq 4$) numeric input domains.

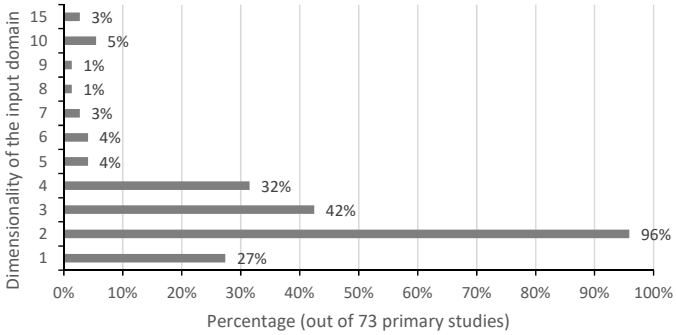


Fig. 13. Input domain dimensionality distribution.

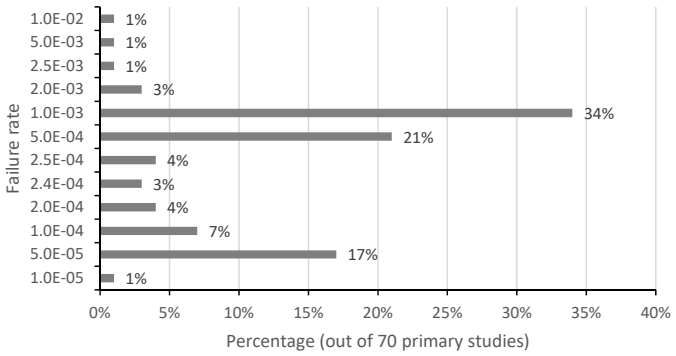


Fig. 14. Distribution of failure rates.

7.2.2 Failure Rate Distribution

Of the 73 primary studies involving simulations, 70 simulated failures in the input domain. Among these 70, the maximum failure rate (θ_H) used was 1.0, and the minimum (θ_L) was $1.0 \cdot 10^{-5}$. Fig. 14 shows the distribution of lowest simulation failure rates (θ_L) across these 70 papers: As shown in Fig. 14, more than half of the simulations involved a minimum failure rate of either $1.0 \cdot 10^{-3}$ (24 papers: 34%) or $5.0 \cdot 10^{-4}$ (15 papers: 21%). The next two most commonly used θ_L values are $5.0 \cdot 10^{-5}$ (17%), and $1.0 \cdot 10^{-4}$ (7%). In total, only one paper had $\theta_L = 1.0 \cdot 10^{-5}$, indicating that the failure rates used in simulations have not been very low. This lack of simulation data for very low failure rates contrasts with Anand et al.'s report that "lower failure rates are actually favorable scenarios for ART with respect to F-measures" [18]. Thus, it would be interesting and worthwhile to conduct more simulations involving lower failure rates to better evaluate ART performance.

7.2.3 Failure Pattern Distribution

Of the 73 primary studies involving simulations, 69 involved specific failure pattern designs, which, according to a simple classification, can be categorized as either *regular* or *irregular* shapes [195], [196]. As shown in Fig. 15, all 69 papers (100%) included regular shapes that, in a two-dimensional input domain ($d = 2$)[6], could be described as:

6. For ease of description, the failure patterns in Fig. 15 focus on two-dimensions, but the categories are similar for higher d: For example, when $d = 3$, a square becomes a cube, and a circle becomes a sphere.

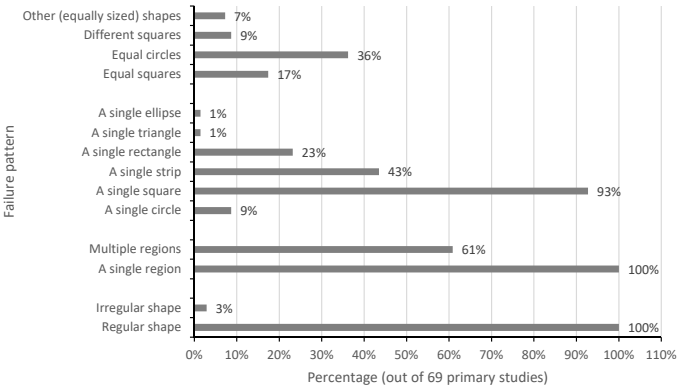


Fig. 15. Distribution of failure patterns.

square, *rectangle*, *strip*[7], *circle*, *ellipse*, or *triangle*. Only two papers (3%) used irregular shapes [195], [196], constructed by combining between two and five regular shapes.

Fig. 15 shows that all 69 papers (100%) had simulations involving a single region, and 42 papers (61%) also investigated multiple regions. The most popular shape used in single-region simulations was the square (93%), followed by strip (43%), rectangle (23%), and circle (9%); both triangle and ellipse were used in only one primary study (1%) each. In simulations with multiple failure regions, most studies used *equal circles* (36% of the 69 papers), *equal squares* (17%), *different squares*[8] (9%), or other (equally-sized) *shapes*[9] (7%).

As shown in Fig. 15, there are many types of failure patterns. Fig. 16 presents how many failure pattern types were used in the 69 studies involving failure patterns. Half of the papers examined three types, followed by one (34%), four (10%), and two types (4%). Only one paper (2%) looked at five types of failure pattern in the simulations [108]. A conclusion from this analysis is that many studies have not designed the simulations comprehensively enough to accurately evaluate ART.

7.3 Experiments with Real Programs

In this section, we summarize some details about the ART experiments involving real programs.

7.3.1 Subject Programs

We collected the details of each subject program used in the ART experiments, including its name, implementation language, size, description, and references to the primary studies that reported results for that program. This information is summarized in the appendix, in Table A.1 — "NR" again indicates that some details were not reported in the original paper. For ease of illustration, we ordered the programs according to the number of references (the last column), listing the most studied ART subject programs at the top of the table.

In total, as can be seen from Table A.1, 211 subject programs were found, ranging in size from 8 to 4,727,209 lines of code. Fig. 17 shows the distribution of programming languages used to implement the programs. It can be observed that most programs were written in C/C++ (36%) and Java (33%), followed by C# (9%).

7.3.2 Types of Faults

The testing effectiveness of ART is generally evaluated according to its ability to identify failures caused by faults in the subject programs. Because actual defects are not always available in real programs, artificially faulty programs, in the form of *mutants*, are often used. The mutants can be created manually, or with an automatic mutation tool [23]. Similar to previous surveys [26], we investigated the relationship between artificial and real faults in the empirical evaluations of ART by calculating the cumulative number of primary studies using each, as shown in Fig. 18. It can be seen that the first study involving artificial faults was reported in 2002, while the first with real faults was in 2008. Furthermore, although both studies with artificial and real faults are increasing, the rate of increase for artificial faults is much higher than that for real faults. By the end of

7. The rectangle type is a special case of the strip type, with the main difference being that each side of the rectangle type is parallel to the corresponding dimension of $\mathcal{D}$, but strips are not necessarily so, and may not be parallelograms [22].

8. Previous studies [108] designed a predominant region by assigning $q\%$ of the failure region to one square (e.g., q may be equal to 30, 50, or 80), with the other squares sharing the remaining percentage of the failure region in a random manner.

9. The unknown shapes refer to situations where the shape details were not provided.

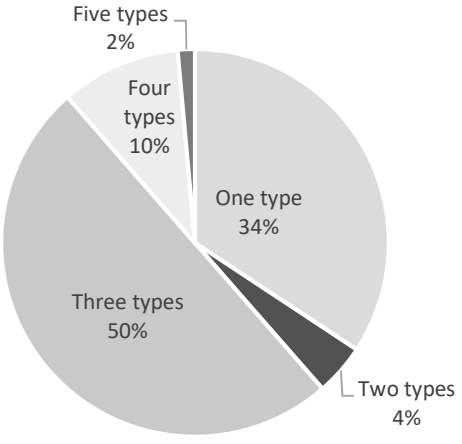


Fig. 16. Distribution of failure pattern types.

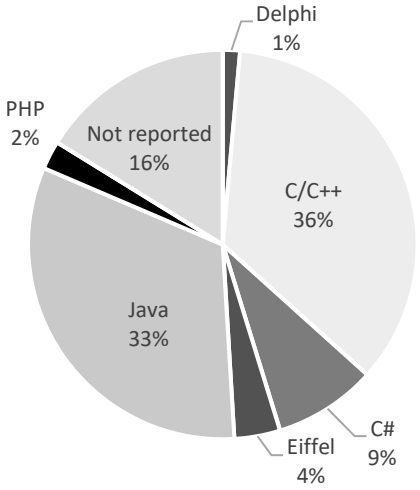


Fig. 17. Programming language distribution of subject programs.

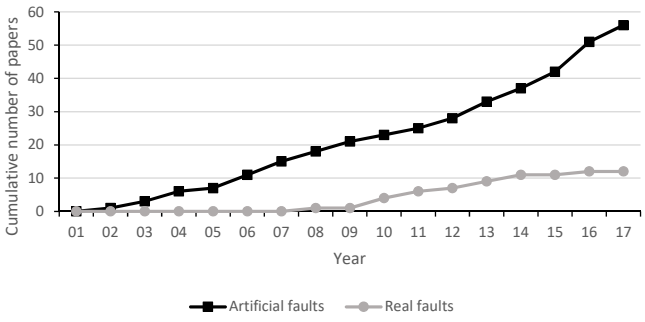


Fig. 18. Real versus artificial faults.

2017, more than 55 primary studies had used artificial faults, compared with only 12 identifying real faults, indicating that relatively few studies have used ART to detect real bugs. Nevertheless, the number of ART studies detecting real faults has been increasing.

7.3.3 Additional Information used to Support ART

ART uses information from previously executed tests to guide generation of subsequent test cases to achieve an even-spreading over the input domain. Each test case contains some intrinsic information: A test case $(0.5, 0.5)$, for example, is a point in a 2-dimensional input domain, with intrinsic information represented by its location; another test case, "xyz", is a list of characters whose intrinsic information is represented by a string. In addition to a test case's intrinsic information, some further information may also be available that can be extracted and applied to support ART execution. In this section, we review some additional information obtained from the ART studies.

Most studies made some use of white-box information (including branch coverage, statement coverage, and mutation score) to guide test case generation. Several studies [42], [174], [183] have used branch coverage information, but have adopted different representations. For a given SUT with a list of β branches, denoted $\mathcal{BR} = \{br_1, br_2, \cdots, br_\beta\}$, a test case tc could cover a set of these branches, denoted $BR(tc)$, where $BR(tc) \subseteq \mathcal{BR}$. Some previous studies [174], [183] have used a binary vector $(x_1, x_2, \cdots, x_\beta)$ where each element x_i $(1 \leq i \leq \beta)$ represents whether or not the branch br_i is covered by tc: if br_i is covered, then $x_i = 1$, otherwise $x_i = 0$. This information can also be represented by a set of branches [42], i.e., $BR(tc)$. Zhou et al. [173] also used a test vector based on branch information $(y_1, y_2, \cdots, y_\beta)$, however, each element y_i $(1 \leq i \leq \beta)$ in their vector represents the number of times that br_i is covered by tc. Jiang et al. [42] used sets of statements or methods for each test case. Both Hou et al. [41] and Sinaga et al. [175] used the program path to represent each test case by constructing a *Control Flow Graph* (CFG) [197] for the program.

Tappenden and Miller [153] also used a binary vector for individual test cases to represent the existence (or lack) of certain cookies within a global cookie collection. Patrick and Jia [198], [199] used mutation scores to construct a probability distribution for test case selections. Some previous studies [155]–[158] have described test cases using UML state machine test paths, considering each test path as either a set or sequence. Iqbal et al. [106], using the same UML state machine as in other studies [155]–[158], used a test data matrix to represent test cases. Matinnejad et al. [159] represented test cases using a sequence of signals that could be described as a function over time; and Liu et al. [116] represented test cases with an event sequence. Indhumathi and Sarala [200] used .NET Solution Manifest files to generate test case scenarios, each one producing at least one test case. Nikravan et al. [201] applied the path constraints of input parameters to support ART. Nie et al. [91] enhanced ART testing effectiveness through the use of I/O relations. When testing C compilers, where each test case was a C program, Chen et al. [184] counted the occurrence of certain tokens in each program, constructing a numeric vector to represent each test case. Hui and Huang [202] applied metamorphic relations to support ART test case generation, and Yuan et al. [203] have incorporated program invariant information into ART.

7.4 Evaluation Metrics

Various metrics have been used to evaluate the testing effectiveness and efficiency of ART approaches. In this section, we review those metrics used in the primary studies.

7.4.1 Effectiveness Metrics

The effectiveness metrics, which are used to evaluate the effectiveness of ART techniques, can be classified into three categories: fault-detection; test-case-distribution; and structure-coverage.

1) *Fault-detection metrics*: These metrics assess the fault-detection ability of ART, and include the *F-measure* [204], *E-measure* [205], and *P-measure* [205]. The F-measure is the expected *F-count* [206] (the number of test cases required to detect a failure in a specific test run); the E-measure refers to the expected number of failures to be identified by a set of test cases; and the P-measure is the probability of a test set identifying at least one program failure. Liu et al. [207] proposed a variant of the F-measure, the *F^m-measure*, which they defined as the expected number of test cases required to identify the first m failures. These metrics may have different application environments: the E-measure and P-measure, for example, are appropriate for the evaluation of automated testing systems [73]; while the F-measure is more realistic for situations where testing stops once a failure is detected.

In addition to these three metrics, another widely-used one is the *fault detection ratio*, which is defined as the ratio of faults detected by a test set to the total number of faults present [185]. It should be noted that in the context of artificial faults (mutants), the fault detection ratio can be interpreted as the *mutation score* [23].

2) *Test-case-distribution metrics*: These metrics are used to evaluate the distribution of a test set, i.e., how evenly spread the test cases are. For ease of description in the following, assume a test set $T = \{tc_1, tc_2, \cdots, tc_n\}$, of size n, from input domain $\mathcal{D}$.

- *Discrepancy* [31]: The definition of discrepancy was given in Eq. (5.1).
- *Dispersion* [31]: The dispersion of T is calculated as the maximum distance among all pairs of nearest neighbor distances. Its definition is:

$$Dispersion(T) = \max_{1 \leq i \leq n} \min_{1 \leq j \neq i \leq n} dist(tc_i, tc_j). \quad (7.1)$$

- *Diversity* [83], [84]: The diversity is similar to the dispersion, but uses the sum (not maximum) of all nearest neighbor distances. Its definition is:

$$Diversity(T) = \sum_{1 \leq i \leq n} \min_{1 \leq j \neq i \leq n} dist(tc_i, tc_j). \quad (7.2)$$

- *Divergence* [114]: Similar to diversity, *divergence* [114] is defined as:

$$Divergence(T) = \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq n} dist(tc_i, tc_j). \quad (7.3)$$

- *Spatial distribution* [31], [82], [87], [104], [117]: This refers to the position of each test case over the entire input domain $\mathcal{D}$, and can only be used for numeric input domains. The most intuitive version of spatial distribution depicts the locations of test cases: Mayer and Schneckenburger [82], for example, recorded the locations of the i-th test case in a 2-dimensional input domain using 10 million test sets, generating a picture of pixels. However, each picture only shows up to the i-th test case, and their method cannot depict spatial distributions for input domains with more than three dimensions. Some methods have tried to project the test case positions onto a single dimension: Chen et al. [117], for instance, projected test cases from T onto one dimension (the x-axis), dividing it into 100 equally-sized bins. The number of test cases within each bin was then counted, and the spatial distribution of T was thus described with a histogram.
  Other methods have described the spatial distribution of T by dividing $\mathcal{D}$ into a number of equally-sized, disjoint subdomains, from $\mathcal{D}$'s edge to its centre: Chen et al. [31], for example, partitioned $\mathcal{D}$ into two subdomains, the *edge* and *center* regions, defining a new measure of spatial distribution as:

$$Edge : Center = \frac{|T_{Edge}|}{|T_{Center}|}, \tag{7.4}$$

  where T_{Edge} and T_{Center} are the sets of test cases from T located in the edge and center regions, respectively. Chen et al. [87] also partitioned $\mathcal{D}$ into 128 subdomains, and analyzed the frequency distribution of test cases in each subdomain. Similarly, Mayer and Schneckenburger [104] divided $\mathcal{D}$ into 100 subdomains, and formalized the relative distance of a test case $tc \in T$ to the center of $\mathcal{D}$:

$$dist_{max}(c, tc) = \max_{i=1,2,\cdots,d} |c_i - tc_i|, \tag{7.5}$$

  where c is the center of $\mathcal{D}$, and d is its dimensionality.

3) *Structure-coverage metrics* [208]: These metrics, which make use of structural elements in the SUT, have been widely used in the evaluation of many testing strategies. Among them, two popular categories are *control-flow coverage* [112] and *data-flow coverage* [111]. Control-flow coverage focuses on some control constructs of the SUT, such as *block*, *branch*, or *decision* [112]. Data-flow coverage, in contrast, checks patterns of data manipulation[10] within the SUT, such as *p-uses*, *c-uses*, and *all-du-paths* [111]. These metrics have also been used to evaluate (fixed-size) test sets generated by ART.

7.4.2 Efficiency Metrics

There have generally been two metrics used to evaluate the testing efficiency of ART: the *generation time*, and the *execution time*. The generation time reflects the computational cost of generating n test cases; while the execution time refers to the time taken to execute the SUT with n test cases. On the one hand, because RT has fewer computations involved in the test case generation, it is intuitive that it should have a much lower generation time than ART: Given the same amount of time, RT typically generates more test cases than ART. On the other hand, the variation in execution time depends mainly on the SUT.

7.4.3 Cost-Effectiveness Metric

The *F-time* [61] is defined as the running time taken to find the first failure. Suppose the testing process requires n test cases to identify the first failure (i.e., the F-count is equal to n), then the F-time comprises the generation time for these n test cases, and the execution time for running them on the SUT. The F-time, therefore, not only shows the testing efficiency of ART, but also reflects its effectiveness: it is a cost-effectiveness metric [18].

7.4.4 Application of Evaluation Metrics

Fig. 19 presents the frequency of each applied metric from the 131 primary studies involving empirical evaluations.

Among the effectiveness metrics, the fault-detection metrics were the most used, followed by test-case-distribution metrics. Very few studies used the structure-coverage metrics. The majority of papers (73%) used the F-measure to evaluate ART fault-detection effectiveness, followed by the fault detection ratio (15%), and the P-measure (11%). Only one paper (1%) used the E-measure, which reflects one of its main criticisms: that higher E-measures do not necessarily imply more distinct failures or faults [32].

Regarding the efficiency metrics, 21% of the 131 papers used the generation time, whereas only 1% used the execution time. Finally, about 8% of the studies adopted the F-time as the cost-effectiveness metric.

7.5 Number of Algorithm Runs

To accommodate the randomness in test cases generated by both RT and ART, empirical evaluations require that the techniques be run in an independent manner, a certain number of times (called the *number of algorithm runs*, $\mathcal{S}$) [209]. In this section, we analyze the number of algorithm runs

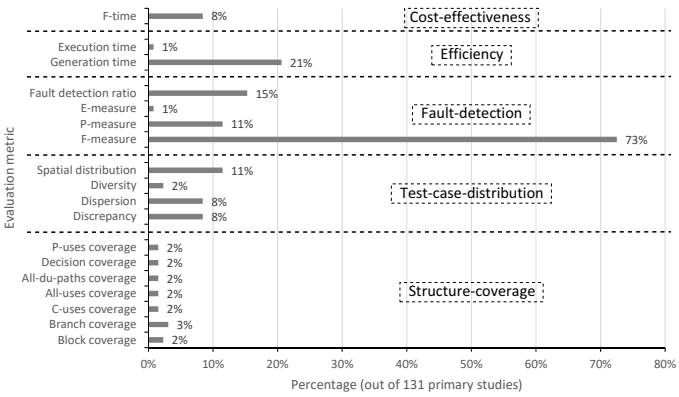
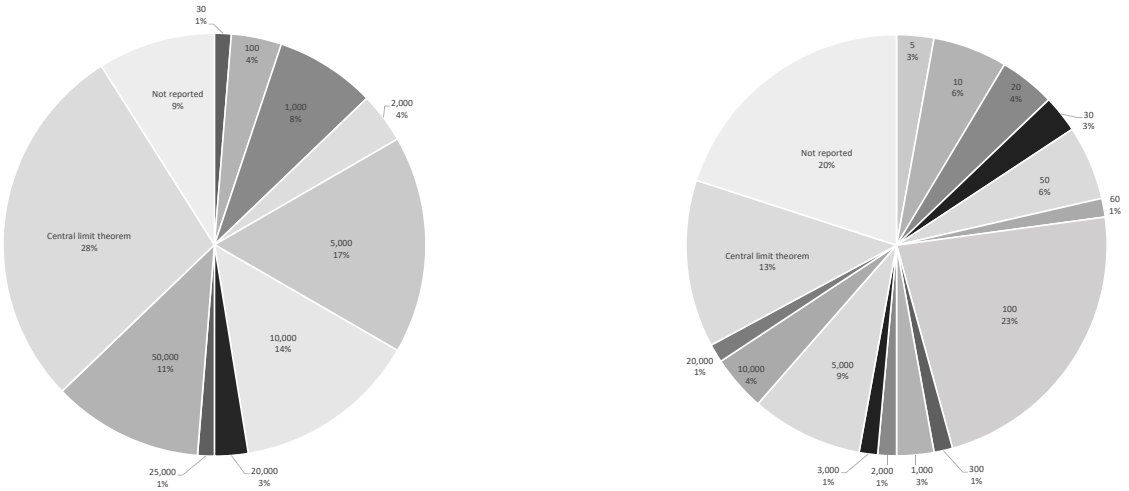


Fig. 19. Application of evaluation metrics.

10. Patterns of data manipulation refer to the definition of some data (*def*, where values are assigned to the data), and its usage (*use*, where the values are used by an operation). Additionally, *use* can be categorized into *c-use* (where data are used as an output or in a computational expression), and *p-use* (where data appears in a predicate within the program) [111].



(a) Simulations.

(b) Experiments with real programs.

Fig. 20. Distribution of the number of algorithm runs reported in the empirical studies.

used in each study[11]. Because some studies involving experiments may have had practical constraints (such as limited testing resources), to present the results in an unbiased way, we investigated the numbers of algorithm runs for both simulations and for experiments with real programs.

Fig. 20 presents the paper classification based on the number of algorithm runs reported, with Fig. 20(a) showing the distribution of the 78 studies with simulations, and Fig. 20(b) showing the 74 studies involving experiments. According to the *Central limit theorem* [210], to estimate the mean of a set of evaluation values (such as F-measures), with an accuracy range of $\pm r$ and a confidence level of $(1-\alpha)\times 100\%$, the size of $\mathcal{S}$ should be at least:

$$\mathcal{S} = \left(\frac{100 \cdot z \cdot \sigma}{r \cdot \mu}\right)^2, \tag{7.6}$$

where z is the normal variate of the desired confidence level, μ is the population mean, and σ is the population standard deviation.

As shown in Fig. 20(a), other than 5% of papers with $\mathcal{S} \leq 100$, and 9% "Not reported", all other simulation papers used either a value of $\mathcal{S}$ determined by the central limit theorem (28%), or had at least $1,000$ algorithm runs (58%). Most studies determined $\mathcal{S}$ based on the central limit theorem, followed by $\mathcal{S} = 5,000$ (17%), $10,000$ (14%), and $50,000$ (11%). On the other hand, as shown in Fig. 20(b), about 46% of papers involving experiments had 100 or less algorithm runs ($\mathcal{S} \leq 100$), followed by 19% with $1,000$ or more ($\mathcal{S} \geq 1,000$). Only 13% of experiment papers used a value of $\mathcal{S}$ calculated using the central limit theorem. According to Arcuri and Briand's practical guidelines [209], algorithms involving randomness should be run at least one thousand times ($\mathcal{S} = 1,000$) for each artifact (exceptions being for heavy time-consuming SUTs, such as embedded systems [106], [155]–[158]). Therefore, while overall the number of algorithm runs for ART simulations was sufficient, it appears that the number of runs in some studies involving experiments was not.

11. If a paper used different numbers of algorithm runs in different empirical studies, the minimum number was selected for $\mathcal{S}$.

7.6 Statistical Significance

One of the initial motivations behind developing ART was to enhance the testing effectiveness of RT. It is natural, therefore, to compare each new ART technique with RT, in terms of testing effectiveness. As already discussed, because test cases generated by both RT and ART contain randomness, it is necessary to determine the *statistical significance* of any comparison [209]. Statistical tests can, amongst other things, determine whether or not there is sufficent empirical evidence to support, with a high level of confidence, that there is a difference between the performance of two algorithms $\mathcal{A}$ and $\mathcal{B}$. Furthermore, when $\mathcal{A}$ does outperform $\mathcal{B}$, it is also important to quantify the magnitude of the improvement. In this section, we report on the application of statistical tests in the empirical evaluations of the ART studies.

Of the 78 primary studies involving simulations, only four papers (5%) used statistical tests. However, of the 73 papers with experiments, 26 (36%) examined the statistical significance when comparing two techniques, with the most used statistical tests being: the *t-test*; the *Mann-Whitney U-test*; *ANalysis Of VAriance (ANOVA) test*; and *Z-test* [209]. The *effect size* was the statistic most often used to measure the magnitude of improvements [209], with two papers using it for simulations [73], [105], and six including it for experimental data [73], [158], [159], [165], [198], [199]. The two main approaches used to calculate the effect sizes are from the work of Cohen [211], and Vargha and Delaney [212].

In summary, it appears that relatively few ART empirical studies have used sufficient and appropriate statistical testing.

Summary of answers to RQ4:

1) *Studies involving simulations generally focused on the dimensionality of the input domain, the failure rate, and the failure pattern. However, many of these simulations appear not to have been comprehensively designed, and thus may not be accurate evaluations of ART.*
2) *The results of the experiments with real programs were influenced by both the subject programs themselves, and the types of faults. It was also noted that, so far, papers reporting the detection of real faults still represent only a small proportion of all empirical studies.*
3) *The F-measure (number of test case executions required to detect the first failure) was the most popular metric for evaluating ART testing effectiveness; test case generation time was the most widely-used for testing efficiency; and F-time (execution time required to detect the first failure) was the most commonly used for cost-effectiveness.*
4) *Overall, the number of algorithm runs was sufficient in simulations, but inadequate for the experiments with real programs. Furthermore, only a small proportion of empirical studies used statistical tests.*

8 Answer to RQ5: What misconceptions surrounding ART exist?

During the development of ART, a number of misconceptions and misunderstandings have arisen, leading to confusion or incorrect conclusions. Some misconceptions have been discussed previously [18], indicating that they have existed for multiple potential ART users, especially those just beginning to apply it. Two main misconceptions are discussed in this section.

8.1 Misconception 1: ART is Equivalent to FSCS

Anand et al. [18] noted that, because FSCS was the first published ART algorithm [9], many studies have presented FSCS as *being* ART, or being equivalent or exchangeable. As discussed in Section 5, FSCS is an ART implementation belonging to the STFCS category, and there are many other STFCS implementations. There are also other ART implementation categories. ART refers to a family of testing approaches in which randomly-generated test cases are evenly spread over the input domain [15]. FSCS is only one of many ART algorithms, and hence ART and FSCS are not equivalent.

8.2 Misconception 2: ART Should Always Replace RT

Although RT requires very little information when testing, ART does make use of additional information (such as locations of previously executed test cases) to guide the test case generation. It may, therefore, seem reasonable that ART should always be better than RT, and thus always replace it. From the perspective of testing effectiveness, however, Chen et al. [108] found that ART's effectiveness is influenced by many factors, including the failure rate, failure pattern, and dimensionality of the input domain. They identified several favorable conditions for ART, including a small failure rate, a low dimensionality, a compact failure region, and a small number of failure regions. Furthermore, because different approaches to achieve an even spread of test cases have resulted in different ART implementations, each implementation also has its own relative advantages and disadvantages (resulting in favorable and unfavorable conditions for its application). In other words, there are situations where ART can have similar, or even worse, testing effectiveness compared to RT. In terms of testing efficiency, compared with RT, in spite of several overhead-reduction algorithms [56], [73], [123], [126], ART still incurs more computational overheads. Consequently, even though ART may have better testing effectiveness than RT, there needs to be a balance between effectiveness and efficiency when choosing either RT or ART: If the ART test case generation time is considerably less than the test setup and execution time, then it would be appropriate to replace RT with ART; otherwise, RT may be more appropriate [18]. Nevertheless, it should be feasible to use ART rather than RT as a baseline when evaluating the state-of-the-art techniques for test case generation, especially from the perspective of testing effectiveness.

Summary of answers to RQ5:

- *Two main misconceptions exist in much of the literature: that ART is equivalent to FSCS; and that RT should always be replaced by ART.*

9 Answer to RQ6: What are the remaining challenges and other future ART work?

A number of open ART research challenges remain, requiring further investigation and additional (future) work.

9.1 Challenge 1: Guidelines for Simulation Design

Although simulations may have limitations compared with real-life programs (because they may not easily and accurately simulate real-world environments, especially complex ones), they are indispensable in the field of ART research. For any given SUT, the fault details — including the size, number, location, and shape of failure regions — are fixed (but unknown) before testing begins. Intuitively, therefore, it is reasonable that studies attempt to simulate faults by controlling and adjusting the factors that create different failure patterns (resulting in different faults). Although some such simulated faults may seldom occur in real-world programs, they may nonetheless be representative of potential real-world situations, especially for numeric programs. Furthermore, for a number of reasons, it can be challenging to obtain real-world faulty programs: their existence or availability, for example, may be limited Simulations, therefore, can be used to compensate for this lack of appropriate real-world programs.

As discussed in Section 7, many studies (58 papers) have used simulations to evaluate ART, and different papers may have different simulation designs. However, these studies only simulated numeric and configurable programs. Furthermore, there is a lack of reliable guidelines regarding simulation design, especially from the perspective of those factors that influence ART effectiveness (such as the failure rate and failure pattern details). The existence of such

guidelines could help testers when choosing simulations for experimental evaluations.

9.2 Challenge 2: Extensive Investigations Comparing Different ART Approaches

As discussed in Section 7, although many ART studies are based on simulations and experiments with real programs, all studies have used simulations with failure rates greater than 10^{-6}, and very few [122], [134] have used experiments with failure rates less than 10^{-6} [18]. Similarly, few studies have investigated the favorable and unfavorable conditions for each ART approach [108]. Furthermore, only a very limited number of studies have used statistical testing with a sufficient number of algorithm runs to evaluate ART. It is therefore necessary to more fully and extensively investigate and compare the different ART approaches. This investigation and comparison needs to address not only the strengths and weaknesses of each ART approach, but should also seek to confirm those theoretical results not yet empirically supported (including, as discussed in Section 5.7.1, the potential for ART to support software reliability estimation).

9.3 Challenge 3: ART Applications

As discussed in Section 6, ART has been used to test many different applications. Although ART could theoretically be applied to test any software applications where RT can be used, there remain some applications that have only been tested by RT, such as SQL database systems [6]. It will therefore be interesting and significant to apply ART to these domains. Furthermore, to date, only those ART approaches using the concept of *similarity*, such as STFCS and SBS, have been used in different applications — other approaches, such as PBS and QRS, have mainly been confined to numeric input domains. It will therefore also be important to apply more different ART approaches to different applications.

A goal of ART is to achieve an even-spreading of test cases over different input domains (including nonnumeric input domains). Unlike numeric input domains, nonnumeric domains cannot be considered Cartesian spaces, making visualization of test input locations and failure pattern shapes infeasible. A key requirement for the application of ART in nonnumeric input domains, therefore, is the availability of a suitable *dissimilarity* or *distance* metric to enable comparison of the nonnumeric inputs. Consider, for example, a program that checks whether or not an input string of the form "*YYYY-MM-DD*" is a valid date: Given three potential string input tests — tc_1 = "*2019-01-31*", tc_2 = "*2019-01-3X*", and tc_3 = "*1998-12-24*" — some string dissimilarity metrics (e.g., *Hamming distance*, *Levenshtein distance*, and *Manhattan distance*) may indicate that tc_3 is farther away from tc_1 than from tc_2 [165]. However, while both tc_1 and tc_3 are valid inputs, tc_2 is invalid, and is thus likely to trigger different behavior and output. If different test inputs trigger different functionalities and computations, they are also likely to have different failure behavior (including detecting or not detecting failures), which means that they are dissimilar to each other. This suggests that it would be desirable to incorporate the *semantics* of nonnumeric inputs into their dissimilarity metrics. If a dissimilarity metric exists that accurately captures the semantic differences between test cases (based on functionality and computation), then ART should be considered.

9.4 Challenge 4: Cost-effective ART Approaches

ART cost-effectiveness is critical for real-life applications, and a number of approaches to reduce the computational overheads while attempting to maintain testing effectiveness have been proposed [55]–[57], [73], [92], [123], [126], [128]. However, some approaches are only applicable to numeric inputs [55]–[57], [73], [92], using the location information of disjoint subdomains to enable their division. The main obstacles to applying these cost-reduction techniques to nonnumeric domains include: (1) how to partition a nonnumeric input domain into disjoint subdomains; and (2) how to represent the "*locations*" of these subdomains.

ART based on the concept of mirroring (MART) [55]–[57], [92] first partitions the numeric input domain into equally-sized, disjoint subdomains, designating one as the *source subdomain* and others as *mirror subdomains*. A mapping relation is used to translate test cases between the source domain and each mirror domain. For example, consider a two-dimensional input domain $\mathcal{D}$, divided into four equally-sized subdomains, $\mathcal{D}_1$, $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$. Without loss of generality, assuming that $\mathcal{D}_1$ is the source subdomain, and the others are mirror subdomains, then once a new test case is generated in $\mathcal{D}_1$ using ART (e.g., FSCS or RRT), a mapping relation between $\mathcal{D}_1$ and $\mathcal{D}_i$ $(i = 2, 3, 4)$ maps the test case to three other test cases in $\mathcal{D}_2$, $\mathcal{D}_3$, and $\mathcal{D}_4$. Although, intuitively speaking, subdomain locations can only be visualized/identified in a numeric input domain, not in a nonnumeric one, if partitioning and subdomain location assignment can be applied to nonnumeric input domains, then MART can be used.

While other overhead-reduction approaches [126], [128] may be applied to both numeric and nonnumeric input domains, they may also involve discarding some information, which may decrease their testing effectiveness. It is therefore necessary to investigate more cost-effective ART approaches for different applications.

9.5 Challenge 5: Framework for Selection of ART Approaches

A framework for the selection of an ART approach could help guide testers to apply ART in practice, especially when facing a choice among multiple approaches. Anand et al. [18] discussed two simple application frameworks, but only at a very high level, and a lot of technical details remain to be determined. The framework design will also need to address the favorable and unfavorable conditions for each ART approach, as identified in the various studies.

9.6 Challenge 6: ART Tools

Although many approaches have been proposed for ART, there are very few tools [40], [73], [114], [144], [150], some of which are: AutoTest, which supports ART for object-oriented (ARTOO) programs written in Eiffel [40]; ARTGen, which supports divergence-oriented ART for Java programs [114]; *Practical Extensions of Random Testing* (PERT), which

supports testing for various input types [150]; and OMISS-ART, which supports FSCS for C++ and C# programs. However, these tools are not publicly available. The only publicly available tool [213] was developed to support FSCS, RRT, evolutionary ART, RBCVT, and RBCVT-Fast [73], but this can only be used for purely numeric input domains. Currently, testers wanting to use ART have to implement the corresponding algorithm themselves. There is, therefore, a desire and need to develop and make available more ART tools to support both research and actual testing.

Summary of answers to RQ6:

- *Six current challenges have been identified for ART that will require further investigation. These are the current lack of: (i) guidelines for the design of ART simulations; (ii) extensive investigations comparing different ART approaches; (iii) ART applications; (iv) cost-effective ART approaches; (v) a framework for the selection of ART approaches; and (vi) ART tools.*

10 Conclusion

In this article, we have presented a survey covering 140 ART papers published between 2001 and 2017. In addition to tracing the evolution and distribution of ART topics, we have classified the various ART approaches into different categories, analyzing their strengths and weaknesses. We also investigated the ART application domains, noting that it has been applied in multiple domains, and has been integrated with various other testing techniques. Furthermore, we have identified that different types of failure patterns have been used in the various reported simulations, and that there has been an increasing number of real faults detected and reported. Finally, we discussed some misconceptions about ART, and listed some current and future ART challenges requiring further investigation. We believe that this article represents a comprehensive reference for ART, and may also guide its future development.

Acknowledgments

We would like to thank the anonymous reviewers for their many constructive comments. We would also like to thank T. Y. Chen for his many helpful suggestions for this article. This work is supported by the National Natural Science Foundation of China, under grant nos. 61502205 and 61872167; the China Postdoctoral Science Foundation, under grant no. 2019T120396; and the Senior Personnel Scientific Research Foundation of Jiangsu University, under grant no. 14JDG039. This work is also supported by the Young Backbone Teacher Cultivation Project of Jiangsu University, and the Postgraduate Research & Practice Innovation Program of Jiangsu Province, under grant no. KYCX19_1614. Rubing Huang is the corresponding author.

References

[1] A. Orso and G. Rothermel, "Software testing: A research travelogue (2000-2014)," in *Proceedings of Future of Software Engineering (FOSE'14)*, 2014, pp. 117–132.

[2] *ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary*, Std., Dec 2010.

[3] P. G. Frankl, R. G. Hamlet, B. Littlewood, and L. Strigini, "Evaluating testing methods by delivered reliability," *IEEE Transactions on Software Engineering*, vol. 24, no. 8, pp. 586–601, 1998.

[4] J. Forrester and B. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *Proceedings of the 4th USENIX Windows Systems Symposium (WSS'00)*, 2000, pp. 59–68.

[5] J. Regehr, "Random testing of interrupt-driven software," in *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT'05)*, 2005, pp. 290–298.

[6] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A genetic approach for random testing of database systems," in *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*, 2007, pp. 1243–1251.

[7] W. Muangsiri and S. Takada, "Random GUI testing of Android application using behavioral model," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 9-10, pp. 1603–1612, 2017.

[8] G. J. Myers, *The Art of Software Testing*. John Wiley & Sons: New York, 2004.

[9] T. Y. Chen, T. H. Tse, and Y. T. Yu, "Proportional sampling strategy: A compendium and some insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65–81, 2001.

[10] L. White and E. Cohen, "A domain strategy for computer program testing," *IEEE Transactions on Software Engineering*, vol. 6, no. 3, pp. 247–257, 1980.

[11] P. E. Ammann and J. C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, 1988.

[12] G. B. Finelli, "NASA software failure characterization experiments," *Reliability Engineering and System Safety*, vol. 32, no. 1–2, pp. 155–169, 1991.

[13] P. G. Bishop, "The variation of software survival times for different operational input profiles," in *Prceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS'93)*, 1993, pp. 98–107.

[14] C. Schneckenburger and J. Mayer, "Towards the determination of typical failure patterns," in *Proceedings of the 4th International Workshop on Software Quality Assurance (SOQUA'07)*, 2007, pp. 90–93.

[15] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse, "Adaptive random testing: The ART of test case diversity," *Journal of Systems and Software*, vol. 83, no. 1, pp. 60–66, 2010.

[16] K. P. Chan, D. Towey, T. Y. Chen, F.-C. Kuo, and R. G. Merkel, "Using the information: Incorporating positive feedback information into the testing process," in *Proceedings of the 7th Annual International Workshop on Software Technology and Engineering Practice (STEP'03)*, 2003, pp. 71–76.

[17] D. Towey, "Adaptive random testing; ubiquitous testing to support ubiquitous computing," in *Proceedings of the International Conference of Korea Society of Information Technology Applications (KITA'07)*. The Korea Society of Information Technology Applications, 2007, pp. 1–9.

[18] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[19] M. S. Roslina, A. A. A. Ghani, S. Baharom, and H. Zulzazil, "A preliminary study of adaptive random testing techniques," *International Journal of Information Technology & Computer Science*, vol. 19, no. 1, pp. 116–127, 2015.

[20] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou, "A revisit of three studies related to random testing," *Science China Information Sciences*, vol. 58, no. 5, pp. 1–9, 2015.

[21] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 4, no. 25, pp. 465–470, 1982.

[22] F. T. Chan, T. Y. Chen, I. K. Mak, and Y. T. Yu, "Proportional sampling strategy: guidelines for software testing practitioners," *Information and Software Technology*, vol. 38, no. 12, pp. 775–782, 1996.

[23] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[24] B. Kitchenham and S. Charters, "Guidelines for performing systematic literature reviews in software engineering," Keele University, Keele, U.K., Tech. Rep. EBSE-2007-01, 2007.

[25] K. Petersen, S. Vakkalanka, and L. Kuzniarz, "Guidelines for conducting systematic mapping studies in software engineering: An update," *Information and Software Technology*, vol. 64, pp. 1–18, 2015.

[26] S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Corts, "A survey on metamorphic testing," *IEEE Transactions on Software Engineering*, vol. 42, no. 9, pp. 805–824, 2016.

[27] T. Y. Chen, F. Kuo, H. Liu, P. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou, "Metamorphic testing: A review of challenges and opportunities," *ACM Computing Surverys*, vol. 51, no. 1, pp. 4:1–4:27, 2018.

[28] B. S. Ahmed, K. Z. Zamli, W. Afzal, and M. Bures, "Constrained interaction testing: A systematic literature study," *IEEE Access*, vol. 5, pp. 25 706–25 730, 2017.

[29] M. Khatibsyarbini, M. A. Isa, D. N. Jawawi, and R. Tumeng, "Test case prioritization approaches in regression testing: A systematic literature review," *Information and Software Technology*, vol. 93, pp. 74–93, 2018.

[30] T. Y. Chen, "Adaptive random testing," in *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, 2008, p. 443.

[31] T. Y. Chen, F.-C. Kuo, and H. Liu, "On test case distributions of adaptive random testing," in *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*, 2007, pp. 141–144.

[32] T. Y. Chen, H. Leung, and I. K. Mak, "Adaptive random testing," in *Proceedings of the 9th Asian Computing Science Conference (ASIAN'04)*, 2004, pp. 320–329.

[33] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing," in *Proceedings of the 7th European Conference on Software Quality (ECSQ'02)*, 2002, pp. 321–330.

[34] ——, "Normalized restricted random testing," in *Proceedings of the 8th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'03)*, 2003, pp. 368–381.

[35] K. P. Chan, T. Y. Chen, F.-C. Kuo, and D. Towey, "A revisit of adaptive random testing by restriction," in *Proceedings of the IEEE 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, 2004, pp. 78–85.

[36] K. P. Chan, T. Y. Chen, and D. Towey, "Restricted random testing: Adaptive random testing by exclusion," *International Journal of Software Engineering and Knowledge Engineering*, vol. 16, no. 4, pp. 553–584, 2006.

[37] T. Y. Chen, F.-C. Kuo, and H. Liu, "Application of a failure driven test profile in random testing," *IEEE Transactions on Reliability*, vol. 58, no. 1, pp. 179–192, 2009.

[38] F. C. Kuo, T. Y. Chen, H. Liu, and W. K. Chan, "Enhancing adaptive random testing in high dimensional input domains," in *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC'07)*, 2007, pp. 1467–1472.

[39] ——, "Enhancing adaptive random testing for programs with high dimensional input domains or failure-unrelated parameters," *Software Quality Journal*, vol. 16, no. 3, pp. 303–327, 2008.

[40] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "ARTOO: Adaptive random testing for object-oriented software," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, 2008, pp. 71–80.

[41] S. Hou, C. Zhang, D. Hao, and L. Zhang, "PathART: Path-sensitive adaptive random testing," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware (Internetware'13)*, 2013, pp. 23:1–23:4.

[42] B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, "Adaptive random test case prioritization," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, 2009, pp. 233–244.

[43] F. T. Chan, K. P. Chan, T. Y. Chen, and S. M. Yiu, "Adaptive random testing with CG constraint," in *Proceedings of the 1st International Workshop on Software Cybernetics (IWSC'04)*, 2004, pp. 96–99.

[44] I. P. E. S. Putra and P. Mursanto, "Centroid based adaptive random testing for object oriented program," in *Proceedings of the 5th International Conference on Advanced Computer Science and Information Systems (ICACSIS'13)*, 2013, pp. 39–45.

[45] T. Y. Chen, F.-C. Kuo, and H. Liu, "Distribution metric driven adaptive random testing," in *Proceedings of the 7th International Conference on Quality Software (QSIC'07)*, 2007, pp. 274–279.

[46] ——, "Adaptive random testing based on distribution metrics," *Journal of Systems and Software*, vol. 82, no. 9, pp. 1419–1433, 2009.

[47] K. P. Chan, T. Y. Chen, and D. Towey, "Good random testing," in *Proceedings of the 9th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'04)*, 2004, pp. 200–212.

[48] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, no. 3, pp. 338–353, 1965.

[49] B. Zhou, H. Okamura, and T. Dohi, "Markov Chain Monte Carlo random testing," in *Proceedings of the 2nd International Conference on Advanced Science and Technology (AST'10)*, 2010, pp. 447–456.

[50] K. P. Chan, T. Y. Chen, and D. Towey, "Probabilistic adaptive random testing," in *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, 2006, pp. 274–280.

[51] T. Y. Chen, D. Towey, and K. P. Chan, "Controlling restricted random testing: An examination of the exclusion ratio parameter," in *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*, 2007, pp. 163–166.

[52] B. Zhou, H. Okamura, and T. Dohi, "Enhancing performance of random testing through Markov Chain Monte Carlo methods," in *Proceedings of the IEEE 12th International Symposium on High Assurance Systems Engineering (HASE'10)*, 2010, pp. 162–163.

[53] ——, "Enhancing performance of random testing through Markov Chain Monte Carlo methods," *IEEE Transactions on Computers*, vol. 62, no. 1, pp. 186–192, 2013.

[54] B. Stephen, "Markov chain monte carlo method and its application," *Journal of the Royal Statistical Society: Series D (The Statistician)*, vol. 47, no. 1, pp. 69–100, 1998.

[55] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and S. P. Ng, "Mirror adaptive random testing," in *Proceedings of the 3rd International Conference on Quality Software (QSIC'03)*, 2003, pp. 4–11.

[56] ——, "Mirror adaptive random testing," *Information and Software Technology*, vol. 46, no. 15, pp. 1001–1010, 2004.

[57] F.-C. Kuo, "An indepth study of mirror adaptive random testing," in *Proceedings of the 9th International Conference on Quality Software (QSIC'09)*, 2009, pp. 51–58.

[58] K. K. Sabor and S. Thiel, "Adaptive random testing by static partitioning," in *Proceedings of the 10th International Workshop on Automation of Software Test (AST'15)*, 2015, pp. 28–32.

[59] T. Y. Chen, R. G. Merkel, P. K. Wong, and G. Eddy, "Adaptive random testing through dynamic partitioning," in *Proceedings of the 4th International Conference on Quality Software (QSIC'04)*, 2004, pp. 79–86.

[60] C. Chow, T. Y. Chen, and T. H. Tse, "The ART of divide and conquer: An innovative approach to improving the efficiency of adaptive random testing," in *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*, 2013, pp. 268–275.

[61] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, "Adaptive random testing through iterative partitioning," in *Proceedings of the 11th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'06)*, 2006, pp. 155–166.

[62] J. Mayer, T. Y. Chen, and D. H. Huang, "Adaptive random testing through iterative partitioning revisited," in *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA'06)*, 2006, pp. 22–29.

[63] F.-C. Kuo, K. Y. Sim, C.-A. Sun, S. Tang, and Z. Q. Zhou, "Enhanced random testing for programs with high dimensional input domains," in *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE'07)*, 2007, pp. 135–140.

[64] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing by exclusion through test profile," in *Proceedings of the 10th International Conference on Quality Software (QSIC'10)*, 2010, pp. 92–101.

[65] H. Liu, T. Y. Chen, and F.-C. Kuo, "Dynamic test profiles in adaptive random testing: A case study," in *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE'09)*, 2009, pp. 418–421.

[66] R. G. Merkel, F.-C. Kuo, and T. Y. Chen, "An analysis of failure-based test profiles for random testing," in *proceedings of the IEEE 35th Annual Computer Software and Applications Conference (COMPSAC'11)*, 2011, pp. 68–75.

[67] T. Y. Chen and R. G. Merkel, "Quasi-random testing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, 2005, pp. 309–312.

[68] ——, "Quasi-random testing," *IEEE Transactions on Reliability*, vol. 56, no. 3, pp. 562–568, 2007.

[69] J. H. Halton, "Algorithm 247: Radical-inverse quasi-random point sequence," *Communications of the ACM*, vol. 7, no. 12, pp. 701–702, 1964.

[70] I. M. Sobol, "Uniformly distributed sequences with an additional uniform property," *USSR Computational Mathematics and Mathematical Physics*, vol. 16, no. 5, pp. 236–242, 1976.

[71] H. Niederreiter, "Low-discrepancy and low-dispersion sequences," *Journal of Number Theory*, vol. 30, no. 1, pp. 51–70, 1988.

[72] H. Chi and E. L. Jones, "Computational investigations of quasirandom sequences in generating test cases for specification-based tests," in *Proceedings of the 38th Conference on Winter Simulation (WCS'06)*, 2006, pp. 975–980.

[73] A. Shahbazi, A. F. Tappenden, and J. Miller, "Centroidal Voronoi Tessellations - A new approach to random testing," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 163–183, 2013.

[74] H. Liu and T. Y. Chen, "Randomized quasi-random testing," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1896–1909, 2016.

[75] R. Cranley and T. N. L. Patterson, "Randomization of number theoretic methods for multiple integration," *SIAM Journal on Numerical Analysis*, vol. 13, no. 6, pp. 904–914, 1976.

[76] T. Kollig and A. Keller, "Efficient multidimensional sampling," *Computer Graphics Forum*, vol. 21, no. 3, pp. 557–563, 2002.

[77] A. B. Owen, "Randomly permuted (t,m,s)-nets and (t, s)-sequences," in *Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing*, H. Niederreiter and P. J.-S. Shiue, Eds. New York, NY: Springer New York, 1995, pp. 299–317.

[78] H. Liu and T. Y. Chen, "An innovative approach to randomising quasi-random sequences and its application into software testing," in *Proceedings of the 9th International Conference on Quality Software (QSIC'09)*, 2009, pp. 59–64.

[79] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004.

[80] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Transactions on Software Engineering*, vol. 36, no. 2, pp. 226–247, 2010.

[81] C. Schneckenburger and F. Schweiggert, "Investigating the dimensionality problem of adaptive random testing incorporating a local search technique," in *Proceedings of the 1st IEEE International Conference on Software Testing Verification and Validation Workshop (ICSTW'08)*, 2008, pp. 241–250.

[82] J. Mayer and C. Schneckenburger, "Statistical analysis and enhancement of random testing methods also under constrained resources," in *Proceedings of the 4th International Conference on Software Engineering Research and Practice (SERP'06)*, 2006, pp. 16–23.

[83] P. M. S. Bueno, M. Jino, and W. E. Wong, "Diversity oriented test data generation using metaheuristic search techniques," *Information Sciences*, vol. 259, pp. 490–509, 2014.

[84] P. M. S. Bueno, W. E. Wong, and M. Jino, "Improving random test sets using the diversity oriented test data generation," in *Proceedings of the 2nd International Workshop on Random Testing (RT'07)*, 2007, pp. 10–17.

[85] X. Huang, L. Huang, S. Zhang, L. Zhou, M. Wu, and M. Chen, "Improving random test sets using a locally spreading approach," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'17)*, 2017, pp. 32–41.

[86] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi Tessellations: Applications and algorithms," *SIAM Review*, vol. 41, no. 4, pp. 637–676, 1999.

[87] T. Y. Chen, F.-C. Kuo, and H. Liu, "Enhancing adaptive random testing through partitioning by edge and centre," in *Proceedings of the 18th Australian Software Engineering Conference (ASWEC'07)*, 2007, pp. 265–273.

[88] ——, "Distributing test cases more evenly in adaptive random testing," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2146–2162, 2008.

[89] J. Mayer, "Efficient and effective random testing based on partitioning and neighborhood," in *Proceedings of the 18th International Conference on Software Engineering and Knowledge Engineering (SEKE'06)*, 2006, pp. 499–504.

[90] C. Mao, T. Y. Chen, and F.-C. Kuo, "Out of sight, out of mind: A distance-aware forgetting strategy for adaptive random testing," *Science China Information Sciences*, vol. 60, no. 9, pp. 092 106:1–092 106:21, 2017.

[91] J. Nie, Y. Qian, and N. Cui, "Enhanced mirror adaptive random testing based on I/O relation analysis," in *Proceedings of the Pacific-Asia Conference on Knowledge Engineering and Software Engineering (KESE'09)*, 2009, pp. 33–47.

[92] R. Huang, H. Liu, X. Xie, and J. Chen, "Enhancing mirror adaptive random testing through dynamic partitioning," *Information and Software Technology*, vol. 67, pp. 13–29, 2015.

[93] T. Y. Chen and D. H. Huang, "Adaptive random testing by localization," in *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, 2004, pp. 292–298.

[94] J. Mayer, "Adaptive random testing by bisection and localization," in *Proceedings of the 5th International Workshop on Formal Approaches to Software Testing (FATES'05)*, 2005, pp. 72–86.

[95] C. Mao and X. Zhan, "Towards an improvement of bisection-based adaptive random testing," in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*, 2017, pp. 689–694.

[96] J. Mayer, "Adaptive random testing by bisection with restriction," in *Proceedings of the 7th International Conference on Formal Engineering Methods (ICFEM'05)*, 2005, pp. 251–263.

[97] ——, "Restricted adaptive random testing by random partitioning," in *Proceedings of the 4th International Conference on Software Engineering Research and Practice (SERP'06)*, 2006, pp. 59–65.

[98] C. Mao, "Adaptive random testing based on two-point partitioning," *Informatica*, vol. 36, no. 3, pp. 297–303, 2012.

[99] T. Y. Chen, D. H. Huang, and F.-C. Kuo, "Adaptive random testing by balancing," in *Proceedings of the 2nd International Workshop on Random Testing (RT'07)*, 2007, pp. 2–9.

[100] J. Mayer, "Lattice-based adaptive random testing," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, 2005, pp. 333–336.

[101] T. Y. Chen, D. H. Huang, F.-C. Kuo., R. G. Merkel, and J. Mayer, "Enhanced lattice-based adaptive random testing," in *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, 2009, pp. 422–429.

[102] K. K. Sabor and M. Mohsenzadeh, "Adaptive random testing through dynamic partitioning by localization with distance and enlarged input domain," *International Journal of Innovative Technology and Exploring Engineering*, vol. 1, no. 6, pp. 1–5, 2012.

[103] ——, "Adaptive random testing through dynamic partitioning by localization with restriction and enlarged input domain," in *Proceedings of the 1st International Conference on Information Technology and Software Engineering (ITSE'12)*, 2012, pp. 147–155.

[104] J. Mayer and C. Schneckenburger, "Adaptive random testing with enlarged input domain," in *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, 2006, pp. 251–258.

[105] A. F. Tappenden and J. Miller, "A novel evolutionary approach for adaptive random testing," *IEEE Transactions on Reliability*, vol. 58, no. 4, pp. 619–633, 2009.

[106] M. Z. Iqbal, A. Arcuri, and L. C. Briand, "Combining search-based and adaptive random testing strategies for environment model-based testing of real-time embedded systems," in *Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE'12)*, 2012, pp. 136–151.

[107] H. Liu, X. Xie, J. Yang, Y. Lu, and T. Y. Chen, "Adaptive random testing through test profiles," *Software: Practice and Experience*, vol. 41, no. 10, pp. 1131–1154, 2011.

[108] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "On favourable conditions for adaptive random testing," *International Journal of Software Engineering and Knowledge Engineering*, vol. 17, no. 06, pp. 805–825, 2007.

[109] T. Y. Chen, F.-C. Kuo, H. Liu, and W. E. Wong, "Does adaptive random testing deliver a higher confidence than random testing?" in *Proceedings of the 8th International Conference on Quality Software (QSIC'08)*, 2008, pp. 145–154.

[110] ——, "Code coverage of adaptive random testing," *IEEE Transactions on Reliability*, vol. 62, no. 1, pp. 226–237, 2013.

[111] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Transactions on Software Engineering*, vol. 9, no. 3, pp. 347–354, 1983.

[112] J. C. Huang, "An approach to program testing," *ACM Computing Surveys*, vol. 7, no. 3, pp. 113–128, 1975.

[113] M.-H. Chen, M. R. Lyu, and W. E. Wong, "Effect of code coverage on software reliability measurement," *IEEE Transactions on Reliability*, vol. 50, no. 2, pp. 165–170, 2001.

[114] Y. Lin, X. Tang, Y. Chen, and J. Zhao, "A divergence-oriented approach to adaptive random testing of Java programs," in

Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), 2009, pp. 221–232.

[115] E. Selay, Z. Q. Zhou, and J. Zou, "Adaptive random testing for image comparison in regression web testing," in *Proceedings of the 16th International Conference on Digital Image Computing: Techniques and Applications (DICTA'14)*, 2014, pp. 1–7.

[116] Z. Liu, X. Gao, and X. Long, "Adaptive random testing of mobile application," in *Proceedings of the 2nd International Conference on Computer Engineering and Technology (ICCET'10)*, 2010, pp. V2–297–V2–301.

[117] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Yang, "An innovative approach to tackling the boundary effect in adaptive random testing," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS 2007)*, 2007, pp. 262a–262a.

[118] J. Geng and J. Zhang, "A new method to solve the boundary effect of adaptive random testing," in *Proceedings of the 7th International Conference on Educational and Information Technology (ICEIT'10)*, 2010, pp. V1–298–V1–302.

[119] T. Y. Chen, D. H. Huang, and Z. Q. Zhou, "On adaptive random testing through iterative partitioning," *Journal of Information Science and Engineering*, vol. 27, no. 4, pp. 1449–1472, 2011.

[120] J. Mayer, "Adaptive random testing with randomly translated failure region," in *Proceedings of the 1st International Workshop on Random Testing (RT'06)*, 2006, pp. 70–77.

[121] ——, "Towards effective adaptive random testing for higher-dimensional input domains," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO'06)*, 2006, pp. 1955–1956.

[122] J. Mayer and C. Schneckenburger, "An empirical analysis and comparison of random testing techniques," in *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering (ISESE'06)*, 2006, pp. 105–114.

[123] K. P. Chan, T. Y. Chen, and D. Towey, "Adaptive random testing with filtering: An overhead reduction technique," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, 2005, pp. 292–299.

[124] T. Y. Chen and R. G. Merkel, "Efficient and effective random testing using the Voronoi diagram," in *Proceedings of the 17th Australian Software Engineering Conference (ASWEC'06)*, 2006, pp. 300–305.

[125] F. Aurenhammer, "Voronoi diagrams – a survey of a fundamental geometric data structure," *ACM Computing Surveys*, vol. 23, no. 3, pp. 345–405, 1991.

[126] K. P. Chan, T. Y. Chen, and D. Towey, "Forgetting test cases," in *Proceedings of the IEEE 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, 2006, pp. 485–494.

[127] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[128] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. G. Merkel, and G. Rothermel, "A cost-effective random testing method for programs with non-numeric inputs," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3509–3523, 2016.

[129] T. Y. Chen, F.-C. Kuo, and Z. Q. Zhou, "On the relationships between the distribution of failure-causing inputs and effectiveness of adaptive random testing," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, 2005, pp. 306–311.

[130] R. Bellman, *Dynamic programming*. Princeton University Press, 1957.

[131] R. Huang, X. Xie, J. Chen, and Y. Lu, "Failure-detection capability analysis of implementing parallelism in adaptive random testing algorithms," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC'13)*, 2013, pp. 1049–1054.

[132] W. H. Press, B. P. Flannery, S. A. Teulolsky, and W. T. Vetterling, *Numerical Recipes*. Cambridge University Press: Cambridge, 1986.

[133] ACM, *Collected algorithms from ACM*. Association for Computer Machinery: New York, 1980.

[134] A. Arcuri and L. C. Briand, "Adaptive random testing: An illusion of effectiveness?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis (ISSTA'11)*, 2011, pp. 265–275.

[135] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[136] "GNU Scientific Library," http://www.gnu.org/software/gsl/.

[137] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[138] N. Walkinshaw and G. Fraser, "Uncertainty-driven black-box test data generation," in *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST'17)*, 2017, pp. 253–263.

[139] "Apache Commons Math framework," https://commons.apache.org/proper/commons-math/.

[140] "Jodatime," http://www.joda.org/joda-time/.

[141] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Object distance and its application to adaptive random testing of object-oriented programs," in *Proceedings of the 1st International Workshop on Random Testing (RT'06)*, 2006, pp. 55–63.

[142] "The EiffelBase Library," http://www.eiffel.com/.

[143] "Apache Commons," http://commons.apache.org/.

[144] J. Chen, F.-C. Kuo, T. Y. Chen, D. Towey, C. Su, and R. Huang, "A similarity metric for the inputs of oo programs and its application in adaptive random testing," *IEEE Transactions on Reliability*, vol. 66, no. 2, pp. 373–402, 2017.

[145] "Codeforge," http://www.codeforge.com/.

[146] "Sourceforge," http://sourceforge.net/.

[147] "Codeplex," http://www.codeplex.com/.

[148] "Codeproject," http://www.codeproject.com/.

[149] "Github," https://github.com/.

[150] H. Jaygarl, C. K. Chang, and S. Kim, "Practical extensions of a randomized testing tool," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, 2009, pp. 148–153.

[151] R. Huang, J. Chen, and Y. Lu, "Adaptive random testing with combinatorial input domain," *The Scientific World Journal*, vol. 2014, no. 15, pp. 1–16, 2014.

[152] "Chris Lott's website," http://www.maultech.com/chrislott/work/exp/.

[153] A. F. Tappenden and J. Miller, "Automated cookie collection testing," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 1, pp. 3:1–3:40, 2014.

[154] J. Chen, H. Wang, D. Towey, C. Mao, R. Huang, and Y. Zhan, "Worst-input mutation approach to web services vulnerability testing based on SOAP messages," *Tsinghua Science and Technology*, vol. 19, no. 5, pp. 429–441, 2014.

[155] H. Hemmati, A. Arcuri, and L. C. Briand, "Reducing the cost of model-based testing through test case diversity," in *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems (ICTSS'10)*, 2010, pp. 63–78.

[156] ——, "Empirical investigation of the effects of test suite properties on similarity-based test case selection," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST'11)*, 2011, pp. 327–336.

[157] ——, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 6:1–6:42, 2013.

[158] A. Arcuri, M. Z. Iqbal, and L. C. Briand, "Black-box system testing of real-time embedded systems using random and search-based testing," in *Proceedings of the 22nd IFIP International Conference on Testing Software and Systems (ICTSS'10)*, 2010, pp. 95–110.

[159] R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann, "Effective test suites for mixed discrete-continuous stateflow controllers," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, 2015, pp. 84–95.

[160] "Delphi," https://www.delphi.com/.

[161] "Common modeling errors stateflow can detect," http://nl.mathworks.com/help/stateflow/ug/common-modeling-errors-the-debugger-can-detect.html/.

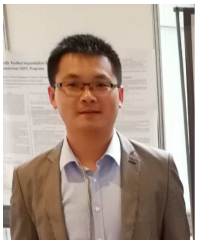
[162] B. Sun, Y. Dong, and H. Ye, "On enhancing adaptive random testing for AADL model," in *Proceedings of the 9th International Conference Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic and Trusted Computing (UTI-ATC'12)*, 2012, pp. 455–461.

[163] "Delphi," https://www.delphi.com/.

[164] R. M. Parizi and A. A. A. Ghani, "On the preliminary adaptive random testing of aspect-oriented programs," in *Proceedings of the 6th International Conference on Software Engineering Advances (ICSEA'11)*, 2011, pp. 49–57.

[165] A. Shahbazi and J. Miller, "Black-box string test case generation through a multi-objective optimization," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 361–378, 2016.
[166] D.-S. Koo and Y. B. Park, "OFART: OpenFlow-switch adaptive random testing," in *Proceedings of the 10th International Conference on Ubiquitous Information Technologies and Applications (CUTE'16)*, 2016, pp. 631–636.
[167] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.
[168] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computer Survey*, vol. 43, no. 2, pp. 11:1–11:29, 2011.
[169] J. D. Musa, "Software reliability-engineered testing," *Computer*, vol. 29, no. 11, pp. 61–68, 1996.
[170] P. Purdom, "A sentence generator for testing parsers," *BIT Numerical Mathematics*, vol. 12, no. 3, pp. 366–375, 1972.
[171] A. Haley and S. Zweben, "Development and application of a white box approach to integration testing," *Journal of Systems and Software*, vol. 4, no. 4, pp. 309–315, 1984.
[172] K. Sen, "Race directed random testing of concurrent programs," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, 2008, pp. 11–21.
[173] Z. Q. Zhou, A. M. Sinaga, and W. Susilo, "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites," in *Proceedings of the 45th Hawaii International Conference on System Sciences (HICSS'12)*, 2012, pp. 5584–5593.
[174] B. Zhou, H. Okamura, and T. Dohi, "Application of Markov Chain Monte Carlo random testing to test case prioritization in regression testing," *IEICE Transactions on Information and Systems*, vol. E95.D, no. 9, pp. 2219–2226, 2012.
[175] A. M. Sinaga, O. D. Hutajulu, R. T. Hutahaean, and I. C. Hutagaol, "Path coverage information for adaptive random testing," in *Proceedings of the 1st International Conference on Information Technology (ICIT'17)*, 2017, pp. 248–252.
[176] B. Jiang and W. K. Chan, "Bypassing code coverage approximation limitations via effective input-based randomized test case prioritization," in *Proceedings of the IEEE 37th Annual Computer Software and Applications Conference (COMPSAC'13)*, 2013, pp. 190–199.
[177] ——, "Input-based adaptive randomized test case prioritization: A local beam search approach," *Journal of Systems and Software*, vol. 105, pp. 91–106, 2015.
[178] J. Chen, L. Zhu, T. Y. Chen, R. Huang, D. Towey, F.-C. Kuo, and Y. Guo, "An adaptive sequence approach for OOS test case prioritization," in *Proceedings of the 27th IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'16)*, 2016, pp. 205–212.
[179] X. Zhang, T. Y. Chen, and H. Liu, "An application of adaptive random sequence in test case prioritization," in *Proceedings of the 26th International Conference on Software Engineering and Knowledge Engineering (SEKE'14)*, 2014, pp. 126–131.
[180] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Enggineering*, vol. 19, no. 1, pp. 65–95, 2012.
[181] X. Zhang, X. Xie, and T. Y. Chen, "Test case prioritization using adaptive random sequence with category-partition-based distance," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS'16)*, Aug 2016, pp. 374–385.
[182] R. Huang, J. Chen, Z. Li, R. Wang, and Y. Lu, "Adaptive random prioritization for interaction test suites," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC'14)*, 2014, pp. 1058–1063.
[183] Z. Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," in *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW'10)*, 2010, pp. 208–213.
[184] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *Proceedings of the 9th IEEE International Conference on Software Testing, Verification and Validation (ICST'16)*, 2016, pp. 266–277.
[185] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.
[186] R. Huang, X. Xie, T. Y. Chen, and Y. Lu, "Adaptive random test case generation for combinatorial testing," in *Proceedings of the IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12)*, 2012, pp. 52–61.
[187] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn, "Constructing test suites for interaction testing," in *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, 2003, pp. 38–48.
[188] C. Nie, H. Wu, X. Niu, F.-C. Kuo, H. Leung, and C. J. Colbourn, "Combinatorial testing, random testing, and adaptive random testing for detecting interaction triggered failures," *Information and Software Technology*, vol. 62, pp. 198–213, 2015.
[189] Y. Liu and H. Zhu, "An experimental evaluation of the reliability of adaptive random testing methods," in *Proceedings of the 2nd International Conference on Secure System Integration and Reliability Improvement (SSIRI'08)*, 2008, pp. 24–31.
[190] D. Cotroneo, R. Pietrantuono, and S. Russo, "RELAI testing: A technique to assess and improve software reliability," *IEEE Transactions on Software Engineering*, vol. 42, no. 5, pp. 452–475, 2016.
[191] H. Yue, P. Wu, T. Y. Chen, and Y. Lv, "Input-driven active testing of multi-threaded programs," in *Proceedings of the 22nd Asia-Pacific Software Engineering Conference (APSEC'15)*, 2015, pp. 246–253.
[192] K. Y. Sim, F.-C. Kuo., and R. G. Merkel, "Fuzzing the out-of-memory killer on embedded Linux: An adaptive random approach," in *Proceedings of the 26th Annual ACM Symposium on Applied Computing (SAC'11)*, 2011, pp. 378–392.
[193] S. H. Shin, S. K. Park, K. H. Choi, and K. H. Jung, "Normalized adaptive random test for integration tests," in *Proceedings of the IEEE 34th Annual Computer Software and Applications Conference Workshops (COMPSACW'10)*, 2010, pp. 335–340.
[194] Y. Qi, Z. Wang, and Y. Yao, "Influence of the distance calculation error on the performance of adaptive random testing," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C'17)*, 2017, pp. 316–319.
[195] T. Y. Chen and F.-C. Kuo, "Is adaptive random testing really better than random testing," in *Proceedings of the 1st International Workshop on Random Testing (RT'06)*, 2006, pp. 64–69.
[196] T. Y. Chen, F.-C. Kuo, and C.-A. Sun, "Impact of the compactness of failure regions on the performance of adaptive random testing," *Ruan Jian Xue Bao(Journal of Software)*, vol. 17, no. 12, pp. 2438–2449, 2006.
[197] F. E. Allen, "Control flow analysis," *ACM SIGPLAN Notices*, vol. 5, no. 7, pp. 1–19, 1970.
[198] M. Patrick and Y. Jia, "Kernel density adaptive random testing," in *Proceedings of the 8th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'15)*, 2015, pp. 1–10.
[199] ——, "KD-ART: Should we intensify or diversify tests to kill mutants?" *Information and Software Technology*, vol. 81, pp. 36–51, 2017.
[200] D. Indhumathi and S. Sarala, "Fragment analysis and test case generation using F-measure for adaptive random testing and partitioned block based adaptive random testing," *International Journal of Computer Applications*, vol. 93, pp. 11–15, 2014.
[201] E. Nikravan, F. Feyzi, and S. Parsa, "Enhancing path-oriented test data generation using adaptive random testing techniques," in *Proceedings of the 2nd International Conference on Knowledge-Based Engineering and Innovation (KBEI'15)*, 2015, pp. 510–513.
[202] Z. Hui and S. Huang, "MD-ART: A test case generation method without test oracle problem," in *Proceedings of the 1st International Workshop on Specification, Comprehension, Testing, and Debugging of Concurrent Programs (SCTDCP'16)*, 2016, pp. 27–34.
[203] Y. Yuan, F. Zeng, G. Zhu, C. Deng, and N. Xiong, "Test case generation based on program invariant and adaptive random algorithm," in *Proceedings of the 6th International Conference on Computer Science and Education (ICCSE'11)*, 2011, pp. 274–282.
[204] T. Y. Chen, F.-C. Kuo, and R. G. Merkel, "On the statistical properties of the F-measure," in *Proceedings of the 4th International Conference on Quality Software (QSIC'04)*, 2004, pp. 146–153.
[205] ——, "On the statistical properties of testing effectiveness measures," *Journal of Systems and Software*, vol. 79, no. 5, pp. 591–601, 2006.
[206] T. Y. Chen and R. G. Merkel, "An upper bound on software testing effectiveness," *ACM Transactions on Software Engineering and Methodology*, vol. 17, no. 3, pp. 16:1–16:27, 2008.

[207] H. Liu, F.-C. Kuo, and T. Y. Chen, "Comparison of adaptive random testing and random testing under various testing and debugging scenarios," *Software: Practice and Experience*, vol. 42, no. 8, pp. 1055–1074, 2012.
[208] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, no. 4, pp. 366–427, 1997.
[209] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Software Testing, Verification and Reliability*, vol. 24, no. 3, pp. 219–250, 2014.
[210] H. Tijms, *Understanding Probability: Chance Rules in Everyday Life*. Cambridge University Press, 2004.
[211] J. Cohen, "A power primer," *Psychological Bulletin*, vol. 112, no. 1, pp. 155–159, 1992.
[212] A. Vargha and H. D. Delaney, "A critique and improvment of the cl common language effect size statistics of mcgraw and wong," *Journal of Education and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.
[213] "RBCVT," http://www.steam.ualberta.ca/main/Papers/RBCVT/.



Rubing Huang received the Ph.D. degree in computer science and technology from the Huazhong University of Science and Technology, Wuhan, China, in 2013. From 2016 to 2018, he was a visiting scholar at Swinburne University of Technology and at Monash University, Australia, respectively. He is an associate professor in the Department of Software Engineering, School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His current research interests include software testing (including adaptive random testing, random testing, combinatorial testing, and regression testing), debugging, and maintenance. He has more than 40 publications in journals and proceedings, including in IEEE Transactions on Software Engineering, IEEE Transactions on Reliability, Journal of Systems and Software, Information and Software Technology, IET Software, International Journal of Software Engineering and Knowledge Engineering, ICSE, and COMPSAC. He is a senior member of the China Computer Federation, and a member of the IEEE and the ACM. More about him and his work is available online at https://huangrubing.github.io/.

Weifeng Sun received the B.Eng. degree in computer science and technology in 2018 from Jiangsu University, Zhenjiang, China, where he is currently working toward the M.Eng. degree with the School of Computer Science and Communication Engineering. His current research interests include software testing and software debugging.

Yinyin Xu received the B.Eng. degree in Computer Science and Technology from Jinling Institute of Technology, Nanjing, China, in 2018. She is currently working toward the M.Eng. degree with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. Her current research interests include software maintenance.

Haibo Chen received the B.Eng. degree in Computer Science and Technology from Changzhou Institute of Technology, Changzhou, China, in 2018. He is currently working toward the M.Eng. degree with the School of Computer Science and Communication Engineering, Jiangsu University, Zhenjiang, China. His current research interests include software testing.

Dave Towey received the B.A. and M.A. degrees in computer science, linguistics, and languages from the University of Dublin, Trinity College, Ireland; the M.Ed. degree in education leadership from the University of Bristol, U.K.; and the Ph.D. degree in computer science from The University of Hong Kong, China. He is an associate professor at University of Nottingham Ningbo China (UNNC), in Zhejiang, China, where he serves as the director of teaching and learning, and deputy head of school, for the School of Computer Science. He is also the deputy director of the International Doctoral Innovation Centre at UNNC. He is a member of the UNNC Artificial Intelligence and Optimization research group. His current research interests include software testing (especially adaptive random testing, for which he was amongst the earliest researchers who established the field, and metamorphic testing), computer security, and technology-enhanced education. He co-founded the ICSE International Workshop on Metamorphic Testing in 2016. He is a member of both the IEEE and the ACM.

Xin Xia is a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received the Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: https://xin-xia.github.io/.

TABLE A.1
Subject Programs in ART

No.	Name	Language	Size	Description	Reference
1.	Bessj	C/C++	NR	Bessel function of general integer order	[16], [32], [33], [34], [36], [46], [47], [67], [68], [74], [88], [90], [92], [107], [108] [109], [123], [207]
2.	Plgndr	C/C++	36	Compute the associated Legendre polynomial $P_l^m(x)$	[32], [33], [34], [36], [46], [47], [67], [68], [74], [88], [90], [92], [107], [123], [108], [198], [199], [207]
3.	Gammq	C/C++	89–106	Normalized incomplete gamma function	[16], [32], [33], [34], [36], [47], [52], [53], [67], [68], [90], [92], [107], [108], [123], [198], [199]
4.	Bessj0	C/C++	28	Bessel function of the first kind	[16], [32], [33], [34], [36], [47], [67], [68], [74], [90], [92], [107], [108], [123], [198], [199], [207]
5.	Erfcc	C/C++	14	Complementary error function erfc(x)	[16], [32], [33], [34], [36], [47], [67], [68], [90], [92], [107], [108], [123], [198], [199]
6.	Cel	C/C++	NR	Cel function	[16], [32], [33], [34], [36], [43], [47], [67], [68], [90], [92], [107], [108], [123]
7.	Sncndn	C/C++	NR	Returns the Jacobian elliptic functions	[16], [32], [33], [34], [36], [47], [67], [68], [90], [92], [107], [108], [123], [207]
8.	Airy	C/C++	43	Evaluate the Airy functions $A_i(z)$ and $B_i(z)$ and their derivatives	[32], [36], [46], [67], [68], [88], [90], [92], [107], [108], [109], [198], [199]
9.	El2	C/C++	NR	El2 function	[32], [36], [46], [67], [68], [88], [90], [92], [107], [108]
10.	Probks	C/C++	22	Probks function	[32], [36], [67], [68], [90], [92], [107], [108], [198], [199]
11.	Golden	C/C++	NR	Golden section search function	[16], [32], [36], [67], [68], [90], [92], [107], [108]
12.	Tanh	C/C++	NR	Tanh function	[32], [36], [67], [68], [90], [92], [107], [108]
13.	TCAS	C/C++	133–206	A traffic collision avoidance system (Siemens suite)	[42], [83], [84], [109], [128], [198], [199], [202]
14.	Gammq	Java	89	Normalized incomplete gamma function	[14], [62], [73], [89], [122], [134]
15.	Replace	C/C++	508–516	Regular expression matching and substitutions (Siemens suite)	[42], [128], [173], [174], [181]
16.	Tot_info	C/C++	272–346	Information measure (Siemens suite)	[42], [128], [174], [181], [183]
17.	Grep	C/C++	3,161–10,068	Regular expression processor	[42], [128], [176], [177], [190]
18.	Printtokens2	C/C++	350–402	Lexical analyzer (Siemens suite)	[42], [128], [174], [181]
19.	Schedule2	C/C++	261–297	Priority scheduler (Siemens suite)	[42], [128], [174], [181]
20.	Flex	C/C++	8,426–10,124	Fast lexical analyzer	[42], [151], [176], [177]
21.	Space	C/C++	5,905–9,564	An interpreter for an array definition language	[173], [174], [175], [183]
22.	Bessj	Java	131	Bessel function of general integer order	[14], [62], [134]
23.	Exprint	Java	86	NR	[14], [73], [134]
24.	Schedule	C/C++	291–299	Priority scheduler (Siemens suite)	[42], [128], [181]
25.	Printtokens	C/C++	341–483	Lexical analyzer (Siemens suite)	[42], [128], [181]
26.	Sed	C/C++	4,756–9,289	Stream editor that perform text transformations on an input stream	[42], [176], [177]
27.	Gzip	C/C++	4,081–5,159	File compression and decompression	[42], [176], [177]
28.	Look	C/C++	135	Find words in the system dictionary or lines in a sorted files	[110], [128], [181]
29.	Spline	C/C++	289	Interpolate smooth curve based on given data	[110], [128], [181]
30.	Cal	C/C++	163	Print a calendar for a specified year or month	[110], [128], [181]
31.	Bessel	C/C++	NR	Calculate the regular cylindrical Bessel function of order n	[52], [53]
32.	Laguerre	C/C++	NR	Calculate generalized Laguerre polynomials	[52], [53]
33.	Ellint	C/C++	NR	Calculate incomplete elliptic integral to the accuracy	[52], [53]
34.	Triangle	Java	26	Classification of isosceles and equilateral triangles	[73], [134]
35.	Triangle2	Java	41	Classification of isosceles and equilateral triangles	[73], [134]
36.	Median	Java	20	Calculate the median value	[73], [134]
37.	Remainder	Java	48	NR	[73], [134]
38.	Variance	Java	22	Calculate the variance value	[73], [134]
39.	BubbleSort	Java	14	Bubble sorting algorithm	[73], [134]
40.	Encoder	Java	65	NR	[73], [134]
41.	Fisher	Java	71	NR	[73], [134]
42.	Select	C/C++	NR	Find the q-th smallest element from an array of p real numbers	[74], [207]
43.	Expint	C/C++	NR	Compute the real exponential integral of a floating point variable	[83], [84]
44.	Tritype	C/C++	NR	Classify the type of triangle and calculates its area	[83], [84]
45.	Alt-sep-test	C/C++	NR	A function from program TCAS	[83], [84]
46.	IC	Java	NR	A seismic acquisition system that interacts with sensors and actuators	[106], [158]
47.	Sort	C/C++	842	Sort and merge files	[110], [128]
48.	CCoinBox	C/C++	120	Simulate a vending machine	[110], [128]
49.	WindShieldWiper	C/C++	233	Simulate a windshield wiper	[144], [178]
50.	SATM	C/C++	197	Simulate an ATM	[144], [178]
51.	RabbitAndFox	C#	770	Simulate a predator-prey model	[144], [178]
52.	WaveletLibrary	C#	2406	Do wavelet analysis	[144], [178]
53.	Multimedia system	C/C++	NR	Manage sending and receiving of multimedia streams	[156], [157]

continued on next page

No.	Name	Language	Size	Description	Reference
54.	Safety control system	C/C++	NR	A safety monitoring component	[156], [157]
55.	Comm	C/C++	144	Select or reject lines common to two sorted files	[110], [181]
56.	Uniq	C/C++	125	Report or remove adjacent duplicate lines	[110], [181]
57.	Triangle	C/C++	75	Classification of isosceles and equilateral triangles	[198], [199]
58.	Action_sequence	Eiffel	2,477	System library of the class for sequence	[40]
59.	Array	Eiffel	1,208	System library of the class for array	[40]
60.	Arrayed_list	Eiffel	2,164	System library of the class for arrayed list	[40]
61.	Bounded_stack	Eiffel	779	System library of the class for bounded stack	[40]
62.	Fixed_tree	Eiffel	1,623	System library of the class for fixed tree	[40]
63.	Hash_table	Eiffel	1,791	System library of the class for hash table	[40]
64.	Linked_list	Eiffel	1,893	System library of the class for linked list	[40]
65.	String	Eiffel	2,980	System library of the class for string	[40]
66.	Biorhythms Problem	Java	NR	A program from the ACM International Collegiate Programming Contest	[41]
67.	Packing Problem	Java	NR	A program from the ACM International Collegiate Programming Contest	[41]
68.	Chocolate Problem	Java	NR	A program from the ACM International Collegiate Programming Contest	[41]
69.	Multiply Problem	Java	NR	A program from the ACM International Collegiate Programming Contest	[41]
70.	Josephus Problem	Java	NR	A program from the ACM International Collegiate Programming Contest	[41]
71.	ConjugateGradient	Java	107	Conjugate Gradient function	[44]
72.	DefaultFieldMatrixChangingVisitor	Java	18	Create custom visitors without defining all methods	[44]
73.	EigenDecomposition	Java	344	EigenDecomposition function	[44]
74.	Abs	Java	8	Abs function	[44]
75.	Gaussian	Java	81	Gaussian function	[44]
76.	HarmonicOscillator	Java	58	Harmonic Oscillator function	[44]
77.	Sigmoid	Java	58	Sigmoid function	[44]
78.	Minus	Java	10	Minus function	[44]
79.	TD1	NR	NR	Calculate employee pay with a simple version	[72]
80.	TD2	NR	NR	Calculate employee pay with a complex version	[72]
81.	Sncndn	Java	NR	Returns the Jacobian elliptic functions	[122]
82.	Coom	C/C++	144	File comparator	[128]
83.	Quadratic	C/C++	NR	Calculate complex roots of the quadratic equation	[109]
84.	Cubic	C/C++	NR	Calculate complex roots of the cubic equation	[109]
85.	Gamma	Java	783	Regularized version of Gammaq	[138]
86.	BesselJ	Java	1,211	Calculate Bessel function	[138]
87.	Binomial	Java	501	Calculate binomial coefficient function	[138]
88.	DerivativeStructure	Java	306	Calculate asinh function	[138]
89.	Erf	Java	763	Calculate erf function	[138]
90.	RombergIntegrator	Java	735	Calculate Romberg integration function	[138]
91.	PeriodToWeeks	Java	1,128	Convert a period to to standard weeks	[138]
92.	DaysBetween	Java	1,251	Date Duration Calculator	[138]
93.	Math.geometry	Java	340	3D calculation	[114]
94.	Math.util	Java	1,161	Mathematic functions	[114]
95.	Lang	Java	4,276	Basic utility	[114]
96.	Lang.text	Java	1,475	Text processing	[114]
97.	Collections.list	Java	823	A container structure	[114]
98.	Siena	Java	1,438	A wide-area event notification system	[114]
99.	Calendar	C/C++	287	Calendar operation	[144]
100.	Stack	C#	420	Stack operation	[144]
101.	Queue	C#	201	Queue operation	[144]
102.	BinarySearchTree	C#	588	Binary search tree algorithms	[144]
103.	BackTrack	C#	1,051	Backtracking algorithms	[144]
104.	NSort	C#	1,118	Sorting algorithms	[144]
105.	SchoolManagement	C#	1,726	Manage school activities	[144]
106.	EnterpriseManagement	C#	1,357	Manage enterprise business	[144]
107.	ID3Manage	C#	4,538	Read and writ of ID3 tags in MP3 files	[144]
108.	IceChat	C#	71,000	Implement an IRC (Internet Relay Chat) Client	[144]
109.	CSPspEmu	C#	406,808	A PSP (PlayStation Portable) emulator	[144]
110.	Poco-1.4.4: Foundation	C/C++	149,547	A platform abstraction layer	[144]
111.	ISSTA Containers	Java	2,000	Container classes	[150]
112.	Java Collections	Java	22,000	Java collection library	[150]
113.	ASM	Java	40,000	A Java bytecode manipulation and analysis framework	[150]
114.	Apache Ant	Java	209,000	A Java-based build tool	[150]
115.	BugTracker.net	C#, ASP.NET	52,147	A time-tested website content management system	[153]
116.	E107	PHP	175,265	A web-based bug tracking and issue tracking application	[153]
117.	GeekLog	PHP	118,706	Manage dynamic web content	[153]
118.	PhpBB2	PHP	43,831	A flat-forum bulletin board software solution	[153]
119.	PhpBB3	PHP	203,465	A flat-forum bulletin board software solution	[153]
120.	PhpMyAdmin	PHP	194,870	Handle the administration of MySQL over the Web	[153]
121.	RWWA1	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
122.	RWWA2	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
123.	RWWA3	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
124.	RWWA4	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
125.	RWWA5	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
126.	RWWA6	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
127.	RWWA7	NR	NR	A real world web application from theTickets.com Pty Ltd	[115]
128.	Stock	NR	NR	Searching stock information	[154]
129.	Weatherforecast	NR	NR	Weather forecast service	[154]
130.	E-Banking	NR	NR	Online banking service	[154]
131.	Bookfinding	NR	NR	Searching book information	[154]

continued on next page

No.	Name	Language	Size	Description	Reference
132.	Domainfinding	NR	NR	Searching domain and	[154]
133.	Petinformation	NR	NR	Searching pet information	[154]
134.	Traintime	NR	NR	Searching train timetable	[154]
135.	Planetime	NR	NR	Searching aircraft flight	[154]
136.	QQcheckonline	NR	NR	Searching QQ online	[154]
137.	Queryresults	NR	NR	Searching student achievement	[154]
138.	Producedorder	NR	NR	Searching production order	[154]
139.	Calculator	NR	NR	Arithmetic calculating service	[154]
140.	Maxdivisor	NR	NR	Finding the greatest	[154]
141.	Mod	NR	NR	Finding the remainder	[154]
142.	Reversestring	NR	NR	Reversing the string	[154]
143.	Stringcopy	NR	NR	Copying the string	[154]
144.	Stringlength	NR	NR	Obtaining the length	[154]
145.	Login	NR	NR	User login	[154]
146.	Vote	NR	NR	Getting the vote	[154]
147.	Echoinformation	NR	NR	Echoing personal information	[154]
148.	Checkeq	C/C++	90	Report missing or unbalanced delimiters and .EQ/.EN pairs	[110]
149.	Col	C/C++	274	Filter reverse paper motions for nroff output for display on a terminal	[110]
150.	Crypt	C/C++	121	Encrypt and decrypt a file using a user supplied password	[110]
151.	Tr	C/C++	127	Translate characters	[110]
152.	Count	C/C++	42	Count lines, words, and characters	[151]
153.	Series	C/C++	288	Generate an additive series of numbers	[151]
154.	Tokens	C/C++	192	Sort/count alphanumeric tokens	[151]
155.	Ntree	C/C++	307	Functions for managing a tree	[151]
156.	Nametbl	C/C++	329	Functions for a symbol table	[151]
157.	SCC	Delphi	NR	A supercharger clutch controller that controls a supercharger clutch	[159]
158.	ASS	Delphi	NR	An auto start-Stop controller that controls the engine torque	[159]
159.	GCS	Delphi	NR	A guidance control system that controls the position of a missile	[159]
160.	UAV	NR	NR	Unmanned aerial vehicle (UAV) cruise control system	[162]
161.	Validation	Java	80	Check whether the string is a valid email address	[165]
162.	PostCode	Java	293	Check whether the string is a valid UK postcodes	[165]
163.	Numeric	Java	217	Validate strings that represent integers	[165]
164.	DateFormat	Java	236	Validate a date in the format 'dd/mm/yyyy'	[165]
165.	MIMEType	Java	145	Validate MIME types	[165]
166.	ResourceURL	Java	339	Validate Resource URLs	[165]
167.	URI	Java	267	Validate different types of URIs	[165]
168.	URN	Java	327	Check whether the string is a valid URNs	[165]
169.	TimeChecker	Java	267	Validate 24 hour time format supplied as strings	[165]
170.	Clocale	Java	751	Validate POSIX locale identifiers	[165]
171.	Isbn	Java	420	Check whether the string is a valid international standard book number	[165]
172.	BIC	Java	200	Check whether the string is a valid bank identifiers code	[165]
173.	IBAN	Java	288	Check whether the string is a valid international bank account numbers	[165]
174.	Bluetooth	Java	NR	A Bluetooth application for file sharing	[116]
175.	Contact	Java	NR	Management of user contact	[116]
176.	SMS	Java	NR	SMS client for sending and receiving SMS	[116]
177.	Bluetalk	Java	NR	An VOIP Bluetooth application	[116]
178.	Dialer	Java	NR	Make and answering calls	[116]
179.	Browser	Java	NR	Mobile Web Browser for surfing Internet	[116]
180.	Open_vSwitch	NR	NR	A OpenFlow-Switch	[166]
181.	Crossword Sage	Java	NR	Unambiguous GUIs for testers to design the test cases	[179]
182.	LLVM	C/C++	4,727,209	Open-source C compiler	[184]
183.	GCC	C/C++	3,343,377	Open-source C compiler	[184]
184.	GCD	Java	NR	Calculate the greatest common divisor	[189]
185.	LCM	Java	6,199	Calculate the least common multiplier	[189]
186.	Siena	Java	6,035	A framework for constructing event notification services	[190]
187.	NanoXML	Java	7,646	A simple XML parser for Java	[190]
188.	Make	C/C++	35,545	Unix build utility	[190]
189.	SP	C/C++	NR	A simple multi-threaded program	[191]
190.	SV	C/C++	NR	A simple multi-threaded program	[191]
191.	SPF	C/C++	NR	A modified version of SP	[191]
192.	SVF	C/C++	NR	A modified version of SV	[191]
193.	Fft	C/C++	NR	A simple multi-threaded program	[191]
194.	Lu_cb	C/C++	NR	A simple multi-threaded program	[191]
195.	Radix	C/C++	NR	A simple multi-threaded program	[191]
196.	TS-7260_ARM	NR	NR	An embedded Linux system with Out-Of-Memory (OOM) Killer	[192]
197.	TextureAtlas	C/C++	405	Store and manipulate multiple textures efficiently for graphics libraries	[199]
198.	Chunkybar	C/C++	451	Implement multi-piece progress bars used in bittorrent clients	[199]
199.	PseudoLRU	C/C++	472	An implementation of the Pseudo-LRU (Least Recently Used) cache algorithm	[199]
200.	QPHashMap	C/C++	568	A hashmap data structure that makes use of quadratic probing in order to manage collisions	[199]
201.	P1	C#	NR	A small C# program	[200]
202.	P2	C#	NR	A small C# program	[200]
203.	P3	C#	NR	A small C# program	[200]
204.	P4	C#	NR	A small C# program	[200]
205.	P5	C#	NR	A small C# program	[200]
206.	Foo	C/C++	8	A simple program with two integer inputs	[201]
207.	TriSquare	NR	168	Check whether 3 positive real numbers could construct a triangle	[202]
208.	Sin	C/C++	99	A Sin mathematic function	[202]
209.	Bessjy	C/C++	332	The Bessel function	[202]
210.	Mulriple	NR	48	Calculate the absolute value, reciprocal or multiple of the input data	[203]
211.	MyMath	NR	90	Calculate the most value of the input data or the simple arithmetic	[203]