Centroidal Voronoi Tessellations—A New Approach to Random Testing

Ali Shahbazi, *Student Member*, *IEEE*, Andrew F. Tappenden, *Member*, *IEEE*, and James Miller, *Member*, *IEEE*

Abstract—Although Random Testing (RT) is low cost and straightforward, its effectiveness is not satisfactory. To increase the effectiveness of RT, researchers have developed Adaptive Random Testing (ART) and Quasi-Random Testing (QRT) methods which attempt to maximize the test case coverage of the input domain. This paper proposes the use of Centroidal Voronoi Tessellations (CVT) to address this problem. Accordingly, a test case generation method, namely, Random Border CVT (RBCVT), is proposed which can enhance the previous RT methods to improve their coverage of the input space. The generated test cases by the other methods act as the input to the RBCVT algorithm and the output is an improved set of test cases. Therefore, RBCVT is not an independent method and is considered as an add-on to the previous methods. An extensive simulation study and a mutant-based software testing investigation have been performed to demonstrate the effectiveness of RBCVT against the ART and QRT methods. Results from the experimental frameworks demonstrate that RBCVT outperforms previous methods. In addition, a novel search algorithm has been incorporated into RBCVT reducing the order of computational complexity of the new approach. To further analyze the RBCVT method, randomness analysis was undertaken demonstrating that RBCVT has the same characteristics as ART methods in this regard.

Index Terms—Adaptive random testing, centroidal Voronoi tessellation, P-measure, random testing, software testing, test case generation, test strategies

1 Introduction

Random Testing (RT) [1], [2], [3], [4] is an effective technique at finding software defects. RT has been successfully applied in several applications, including database systems [5], [6], Java Just-In-Time (JIT) compilers [7], .NET error detection [8], security assessment [9], Windows NT [10], and Mac OS robustness assessment [11]. Fuzzing (Fuzz testing), which is a form of blackbox RT, is used by many software companies, most notably Microsoft. The Trustworthy Computing Security Development Lifecycle document (SDL) [12], published by Microsoft in 2005, states that fuzzing is a key tool for security vulnerability identification. In addition, RT can be combined with other testing techniques. To generate well-formatted test cases, grammars are used in conjunction with RT or fuzzing in some situations [9], [13]. That is, RT is applied with a certain set of rules (grammar) to generate test cases. Similarly, RT can be used to support regression testing [8] and coverage testing [14], [15] to generate new, random test cases.

RT is easy to implement and has a low computational cost. However, RT does not provide satisfactory effectiveness with respect to fault detection. This problem can be explained by failure regions, which are areas in the input domain that trigger faults. It has been empirically shown that faults often occur in failure regions or error crystals [16], [17], [18], [19], [20], which means that faults are mapped onto a cluster within the input domain [21]. Since an RT strategy uses random test cases, the input test cases may not cover all the regions of the input domain resulting in poor failure detection. Fig. 1 presents a set of test cases produced by RT. This figure represents RT's inefficacy in "evenly" distributing test cases throughout the input domain. In this figure, regions one and two have equal sizes. However, no test cases are produced in region one by the RT generator, whereas we have 12 test cases in region two.

Since RT is a common testing strategy, any improvement in this domain could have a significant effect on the 500 billion dollars per year lost due to poor software quality [22]. Some approaches have been already introduced to address this problem, such as Adaptive Random Testing (ART) [23], [24], [25], [26], [27], [28], [29], [30], [31], [32], [33], [34] and Quasi-Random Testing (QRT) [35], [36].

ART is designed to evenly distribute random test cases across the input domain in order to effectively detect failure regions with respect to the observation that faults often occur in failure regions or clusters in the input domain [21], [17], [18], [37]. As a consequence, ART methods increase the probability of fault detection within the input domain [33]. Another approach is QRT which applies a quasi-random sequence to generate test cases [35], [36]. Quasi-random sequences take advantage of a group of mathematical algorithms producing low-discrepancy sequences. Superior effectiveness of QRTs against RT has been demonstrated by Chen and Merkel [35] and Chi and Jones [36]. However, the use of quasi-random sequences in software testing suffers from the following limitations: 1) Quasi-random algorithms are only valid for a finite number of dimensions [38], and 2) quasi-random algorithms can generate only a limited

- *A. Shahbazi and J. Miller are with the Department of Electrical and Computer Engineering, University of Alberta, 9107, 116st, Edmonton, AB T6G 2V4, Canada. E-mail: ali.shahbazi@ualberta.ca; jm@ece.ualberta.ca.*
- *A.F. Tappenden is with the Department of Computing Science, The King's University College, 9125-50 Street, Edmonton, AB T6B 2H3, Canada. E-mail: andrew.tappenden@kingsu.ca.*

Manuscript received 7 Apr. 2011; revised 1 Feb. 2012; accepted 8 Feb. 2012; published online 2 Mar. 2012.
Recommended for acceptance by A. Bertolino.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-04-0114.
Digital Object Identifier no. 10.1109/TSE.2012.18.

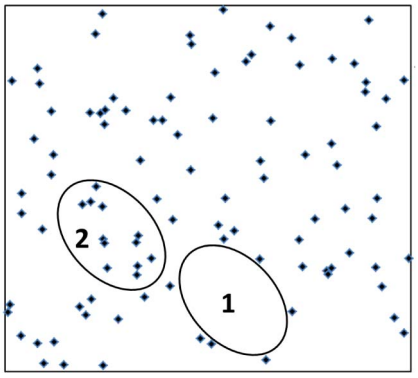

Fig. 1. RT fails to evenly distribute the test cases throughout the input domain. No test cases are produced in region one by the RT generator, whereas we have 12 test cases in region two with the same size.

number of different sequences [35] since they use a deterministic algorithm to produce the test cases.

In this research, we propose a new test case generation approach, namely, Random Border Centroidal Voronoi Tessellations (RBCVT), which utilizes Centroidal Voronoi Tessellations (CVT). The proposed RBCVT approach enhances the existing state-of-the-art test case generation techniques. Specifically, we will demonstrate that RBCVT:

1. is able to produce a superiorly distributed set of test cases when compared to RT, ART, and QRT;
2. still retains the random nature of RT; and,
3. can be optimized to have linear execution characteristics across a wide set of situations.

RBCVT is not an independent method to generate input test cases. It considers other test case generation methods as input and increases software testing effectiveness by spreading the test cases more uniformly throughout the input domain. In addition, a novel search algorithm is proposed to enhance the computational complexity of the RBCVT test case generation from a quadratic to linear runtime order.

In addition to the even distribution of test cases over the input space, the degree of randomness 1) within a set of test cases and 2) between multiple sequences of test sets is an important aspect. The test cases' randomness is critical in avoiding systematic poor performance in certain situations (that is, where a nonrandom sequence could significantly (negatively) correlate with a current set of defects). Similarly, in regression-type testing, we can prevent inefficient testing if test cases are uncorrelated with respect to each other, meaning a high degree of randomness. The proposed RBCVT approach seeks to generate a more effective sequence of test cases with respect to software testing practice while retaining the degree of randomness possessed by RT and ARTs methods. This randomness requirement is investigated using Kolmogorov complexity, which provides a new class of distances appropriate for measuring similarity relations between sequences [39], [40].

The rest of this paper is organized as follows: Section 2 explains notations and assumptions. Section 3 reviews current approaches in software testing. An in-depth description of CVT and its novel application in software testing are presented in Section 4. Accordingly, Section 5 proposes the RBCVT and RBCVT-Fast algorithms and their runtime analysis. Section 6 analyzes the test case generation and introduces two experimental frameworks, namely, a simulation-based and a mutant-based software testing framework. The performance of RBCVT with illustrative results for both testing frameworks as well as empirical runtime analysis is demonstrated in Section 7. Section 8 investigates the test cases' degree of randomness for test case generation methods. Finally, conclusions are drawn in Section 9.

2 Notations and Assumptions

The following notations and assumptions are provided to simplify the discussion in the rest of this paper.

- I denotes the input space which is considered a 2D unit hypercube ($I = [0,1]^2$).
- H denotes the area outside I which is defined as $H = [0-h, 1+h]^2 - I$, where the width of H is indicated by h.
- d denotes the dimension of a test case or input space.
- $|.|$ denotes the size of a set.
- T denotes selected test cases on I generating a test set ($T = \{t_i\}_{i=1}^{|T|}$).
- B denotes a random background point set on $H \cup I$ regarding the RBCVT calculation algorithm ($B = \{b_j\}_{j=1}^{|B|}$).
- R denotes a random border point set on H which simulates random borders in RBCVT approach ($R = \{r_n\}_{n=1}^{|R|}$).
- TR denotes the combination of T and R which is defined as ($TR = T \cup R = \{tr_m\}_{m=1}^{|TR|}$), where $|TR| = |T| + |R|$.
- V_i denotes a Voronoi region (a cell in Voronoi tessellation).
- $dist(p,q)$ denotes the euclidian distance between points p and q.
- $\beta(p,T)$ denotes the nearest point of T to the point p.
- $O(.)$ represents the runtime order of an approach.
- $\arg\max(.)$ returns the index of an element with maximum value.
- θ denotes the failure rate.
- std denotes the standard deviation.
- $\oplus$ is the bit-by-bit exclusive-or operator.
- $\mathrm{XOR}_{j=1,\ldots,k}(.)$ denotes the bit-by-bit exclusive-or for the specified range.
- G_N represents the number of cells in each dimension of the grid with respect to RBCVT-Fast algorithm.
- C_{avg} denotes the average number of points in each cell.
- $Round(.)$ returns the nearest integer value to the input data.
- C_l is a set which contains all the cells in layer l where each cell in C_l is denoted by c_{lm}.
- $dist_c(b_j, c_{lm})$ indicates the minimum euclidian distance between the point b_j and the cell c_{lm}.
- $dist_l(b_j, l)$ represents the minimum euclidian distance between point b_j and cells in layer l.
- $\beta_c(b_j, c_{lm})$ denotes nearest child of c_{lm} to the point b_j.
- tr_{winner} denotes a point of TR with minimum euclidian distance from b_j.
- $RTime(.)$ denotes a runtime of an algorithm or a method.

- $\varphi(T)$ indicates the preprocessing function which preforms the required processing on T regarding randomness analysis.
- $CR(T)$ represents compression ratio of T.
- $\delta(.)$ denotes the Kolmogorov complexity of the input data.
- $NCD(T_i, T_j)$ represents the normalized compression distance (NCD) between T_i and T_j.

3 Current Approaches

3.1 Adaptive Random Testing

Adaptive Random testing methods seek to resolve the deficiencies of RT demonstrated in Fig. 1. These methods seek to retain the random nature of RT while providing a more "even distribution" of the sequence of test cases across the input domain. Since the introduction of ART by Chen et al. [28], a variety of different ART methods have been proposed, including Fixed Size Candidate Set (FSCS) [21], [25], [28], Restricted Random Testing (RRT) [41], Mirror Adaptive Random Testing (M-ART) [24], Adaptive Random Testing by Bisection (ART-B) [32], Adaptive Random Testing by Random Partitioning (ART-RP) [27], ART through Iterative Partitioning (IP-ART) [30], ART based on distribution metrics [42], and Evolutionary Adaptive Random Testing (EAR) [33].

The ART methods are developed based on the observation that failures occur in failure regions which are clustered within the input domain. Each of these methods possesses strengths and weaknesses regarding efficient test case generation and computational complexity. Via empirical investigations, Mayer and Schneckenburger [43] concluded that FSCS [21], [25], [28] and RRT [41] were the best ART methods. Subsequently, Tappenden and Miller [33] introduced EAR and demonstrated that this method has superior performance than FSCS and RRT. Hence, we compare RBCVT's performance against these methods. In each of these ART techniques, the first test case is generated randomly and subsequent test cases are based on each method's specific algorithm.

3.1.1 Fixed Size Candidate Set

FSCS uses a distance-based algorithm to generate test cases. In this method, a fixed size candidate set is used to produce test cases. A set of k randomly generated candidates, cd, are evaluated against all previously selected test cases and a candidate with largest distance from previously executed test cases is selected as

$$J = \arg \max_{j=1,\ldots,k} (dist(cd_j, \beta(cd_j, T))), \quad (1)$$

where cd_j denotes the jth candidate and J represents the index of selected candidate as a next test case. The computational requirement for this method is $FSCS(|T|) \in O(|T|^2)$ due to the computation of the distance between candidates and each previously generated test case [33], [43].

3.1.2 Restricted Random Testing

RRT [41] also uses a distance-based algorithm to generate test cases via a circular exclusion zone [43] centered around each previously generated test case. The radius of each exclusion zone is determined using a constant coverage ratio (γ), which is the sum of the areas of all the existing exclusion zones divided by the total area of the input domain. A candidate test case (cd_j) is generated randomly and disregarded if it is within the exclusion zone of any other test case, i.e., if the following inequality is true:

$$dist(cd_j, \beta(cd_j, T)) < \sqrt{\frac{\gamma}{\pi |T|}}. \quad (2)$$

This process is repeated until an appropriate candidate is found [37], [23], [41]. Calculation of the algorithm's computational efficiency is not straightforward, given the stochastic nature of the technique. However, it has been demonstrated empirically that the average runtime order is within $RRT(|T|) \in O(|T|^2 \log(|T|))$ [43].

3.1.3 Evolutionary Adaptive Random Testing

EAR uses an evolutionary approach to find an approximation for the test case that has the maximum distance from all the previous test cases [33]. For each test case, a pool of k (population size) random candidates is generated. This population is evolved until a stopping criterion is met. This approach is encoded using two genes in each chromosome. The evolution is based upon a euclidean distance-based fitness function [33]:

$$Fitness(ch_j, T) = dist(ch_j, \beta(ch_j, T)), \quad (3)$$

where ch_j represents a chromosome. Single-point crossover was applied to the two chromosomes to generate an offspring evolving the population. When the stopping criterion is met, the best chromosome is selected as the next test point according to the fitness function. The runtime of this algorithm [33] is in the order of quadratic time ($EAR(|T|) \in O(|T|^2)$).

It is worthwhile to note that there are two suboptimum techniques, introduced in previous ART studies, to reduce the ART computational complexity, namely, mirroring [24] and forgetting [44]. Both techniques can be applied to all the studied ART methods. Producing the next test case get more time consuming as the number of test cases grows. Accordingly, the technique of forgetting only considers a constant number of previous test cases when designing a new test case, not all of them. It makes the new test case design independent of $|T|$, leading to a one order reduction in the overall time complexity. In mirroring, ART is only applied to a part of the input domain and then the designed test cases are mirrored to other parts. Obviously, there is a tradeoff between effectiveness and computational complexity if the techniques of mirroring and forgetting are applied.

3.2 Quasi-Random Testing

In addition to ART, the use of quasi-random sequences in software testing has been recently proposed [35], [36]. Quasi-random sequences are mathematically developed sequences which are rigorously designed to produce low-discrepant sample points in a d-dimensional hypercube. They fill the space more uniformly than uncorrelated random points. It has been observed that [35], [36], using these sequences as input test case generators, produce better results than RT in software testing, but it has not been shown that their results are better than ART methods. Until now, various quasi-random sequences has been constructed, including Sobol

[45], Halton [46], Niederreiter [47], Faure [48], and Hammersley [49]. In this paper, we consider the following quasi-random sequences.

3.2.1 The Halton Sequence

The Halton sequence has been derived from the Van der Corput sequence [36], which is defined as

$$\Phi_b(n) = \sum_{j=0}^{k} n_j b^{-j-1}, \tag{4}$$

where n_j is the jth digit of n in the base b, and k denotes the lowest integer that makes $n_j = 0$, for all $j > k$. The Halton sequence can be seen as the natural d-dimensional extension of the Van der Corput sequence. The Halton sequence generates values deterministically using prime numbers as its base. The standard Halton sequence performance is good in low dimensions, whereas in large dimensions a correlation problem between sequences generated in different dimensions appears [50]. As a remedy, several scrambling and randomization methods have been introduced [50].

3.2.2 The Sobol Sequence

The Sobol sequence [45] has been proposed for software testing by Chi and Jones [36]. The Sobol sequence can be considered as a permutation of the binary Van der Corput sequence in each dimension [36] and is defined by the following equations:

$$Sobol(n) = \underset{j=1,\ldots,k}{\mathrm{XOR}}(n_j w_j), \tag{5}$$

$$w_j = \underset{i=1,\ldots,r}{\mathrm{XOR}}\left(\frac{\alpha_i w_{j-i}}{2^j}\right) \oplus \frac{w_{j-r}}{2^{j+r}}, \tag{6}$$

where n_j is the jth digit of n in binary, k represents the number of digits of n in binary, and $Sobol(n)$ denotes the nth element of the Sobol sequence. To construct a Sobol sequence, we need to choose a primitive polynomial of degree r with $\alpha_i \in \{0, 1\}$ coefficients. The required computational overhead for Sobol generator is within the order of $Sobol(|T|) \in O([\log(|T|)]^2)$ [51]. This low computational cost is the primary advantage of QRT compared to ART approaches.

3.2.3 The Niederreiter Sequence

The Niederreiter sequence was introduced in 1988 [47] and provides a general form for quasi-random sequences. This sequence has provided a good reference for other quasi-random sequences, as all of these methods can be described in terms of what Niederreiter called (t, s)-sequence. The discrepancy of this sequence is lower than any other known sequences [35]. Chen and Merkel [35] have proposed this sequence for test case generation where a large number of test cases are required.

4 Centroidal Voronoi Tessellation

In this section, we introduce the concept of CVT and discuss approaches to its calculation as well as its application to software testing. A Voronoi diagram (Voronoi tessellation) is a decomposition of a space, in our case a unit hypercube, into a set of cells (Voronoi regions) such that $V_i \bigcap V_j = \emptyset$ for $i \neq j$, and $\bigcup_{i=1}^{k} V_i = 1$, where V_i is a Voronoi region and k is the number of Voronoi regions. Each Voronoi region is associated with an object and consists of all the areas that are closer to that object than any other object. These objects are disjoint [52] and are referred to as the generators of the Voronoi diagram. In this research, an object is a point (t_i) and euclidian distance is considered as a distance measure. The Voronoi region corresponding to the point t_i is defined as

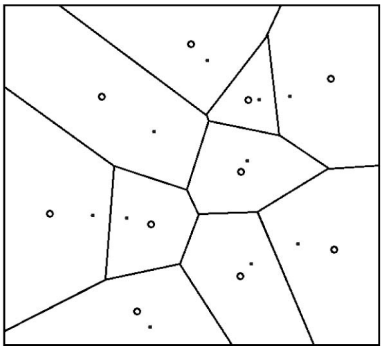

Fig. 2. The lines specify Voronoi regions corresponding to 10 randomly generated points. The points are Voronoi generators and the circles are the centroids of the Voronoi regions.

$$V_i = \{x \in I | \forall j = 1, \ldots, |T|, j \neq i : dist(x, t_i) < dist(x, t_j)\}. \tag{7}$$

Centers of mass, centroids, of a Voronoi region (V_i) are defined as

$$t_i^* = \frac{\int_{V_i} x\rho(x)dx}{\int_{V_i} \rho(x)dx}, \tag{8}$$

where ρ is a density function defined in I. Centroids in the decomposed cells of a Voronoi tessellation possess characteristics that seem to have some advantages with respect to software testing. In Fig. 2, adapted from [53], 10 randomly (RT or alternatively by using ART or QRT techniques) generated points are used as the generators or inputs to the system. Accordingly, the Voronoi regions have been formed corresponding to the generators and the centroid of each Voronoi region is indicated by a circle. As shown in this figure, the resulting circles are "more evenly distributed" compared to the input points making them more appropriate for software testing.

A CVT is a collection of Voronoi regions where their generator points are the centroids of the corresponding Voronoi regions [53]. This case is a special case, and the probability of a set of random generators having the same positions as the centroids is quite low. In general, the generators of Voronoi tessellations will not be at the same places as the centroids. An important property of CVT is that these special generators producing a CVT are not unique and we can have distinct CVTs within a d-dimensional unit hypercube [53], [54].

A CVT can be produced either deterministically or probabilistically [53], [54], [55]. A deterministic approach, such as Lloyd's method [53], produces a consistent output for every input, whereas a probabilistic approach such as MacQueen's method [54] uses a random mechanism to generate a CVT leading to distinct outputs, for the same input

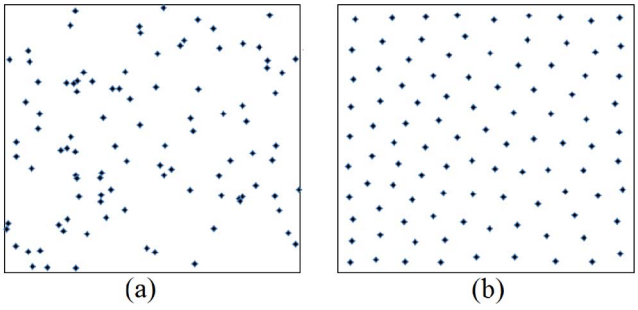

Fig. 3. The (a) RT and (b) corresponding CVT points generated using a probabilistic approach.

set, in different runs, allowing additional exploration of the input space. Since this is beneficial during testing scenarios (e.g., regression situations), we develop a probabilistic calculation approach in this study for the RBCVT test case generation method which is introduced in Section 5.

4.1 CVT and Software Testing

In this section, we introduce the application of CVT in software testing as well as its desirable and undesirable features in this regard. CVT has been applied within the wide array of applications [53]. However, the use of this technique for improving RT, ART, and QRT techniques is novel. The CVT methodology requires a set of initial points named generators. The use of the output from other test case generation methods (RT, ART, and QRT) is proposed as inputs (generators) to the CVT algorithm leading to an improved set of test cases. Chen and Merkel [26] presented a new calculation method for FSCS using Voronoi diagrams; they utilized Voronoi diagrams to develop a search algorithm with the ability to calculate $\beta(c_j, T)$ with a reduced computational complexity. This work is significantly different from our proposed use of Voronoi diagrams in test case generation since they use Voronoi diagrams to speed up finding the nearest point in FSCS test case generation approach, whereas we use the centroids of Voronoi regions to improve the effectiveness of the test case production.

To indicate CVT's effect on test cases, Fig. 3 is presented. This figure indicates the generator (input) points for CVT (Fig. 3a), points generated by RT, as well as the resultant points generated by CVT (Fig. 3b). According to this figure, one can observe that CVT points possess the following desirable properties:

- The CVT points are more "evenly distributed" than their generators in the space. Since faults often occur in failure regions or error crystals, the CVT points are likely to detect a failure region more efficiently.
- As discussed in the previous section, as CVT generates its (output) points by a probabilistic approach, the displayed points are not unique as the CVT process is stochastic. Furthermore, the input generators are generated using a random procedure, except for quasi-random points. Therefore, the output CVT points seem to possess "randomness" (the randomness will be investigated in Section 8).

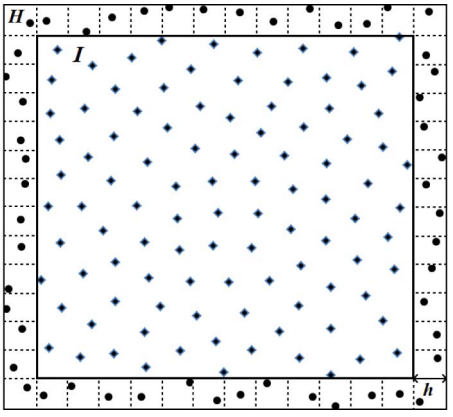

Fig. 4. RBCVT test cases in I and the random border points (R) in H.

Further, the application of CVT to software testing requires a unique solution to the "boundary conditions" introduced by this domain. It is a well-established principle that the probability of a software defect is higher near the boundaries. In this regard, CVT needs to be extended to explicitly consider defect behavior near these boundaries. As indicated in Fig. 3b, all the test cases near the borders have a relatively constant distance with the border. Accordingly, CVT is unable to generate test cases near or on the border. This undesirable feature is due to the traditional CVT definition. To solve this problem, we propose the novel RBCVT approach, which is presented in the next section.

5 Proposed Test Case Generation Approach: Random Border CVT

In this section, we propose the novel RBCVT test case generation approach, which removes the undesirable feature of the CVT discussed in the previous section. In this regard, we propose a RBCVT calculation approach and investigate its associated runtime order. In addition, we propose a novel search algorithm to reduce the computational complexity of RBCVT. Finally, we investigate the generalization of the RBCVT beyond two dimensions.

RBCVT is based on defining an imaginary random border outside the real borders of I. In this regard, we introduce a set of random points (R) in H which simulate an imaginary random border, as discussed in the next section. In Fig. 4, a set of RBCVT test cases is demonstrated as well as the random border points in H. As indicated in this figure, RBCVT effectively removes the aforementioned undesirable feature of the CVT. Accordingly, Fig. 5 indicates the generator points of RBCVT (one for each of the seven test generation methods studied) in the left-hand side and the resultant RBCVT points on the right-hand side.

5.1 RBCVT Calculation Method

To calculate the RBCVT test cases using a set of generator points, we propose a probabilistic method as follows:

- Step 1: Determine the initial set of $T = \{t_i\}_{i=1}^{|T|}$ as generators, $t_i \in I$, where $i = 1, \ldots, |T|$.
- Step 2: Initialize a random border point set of $R = \{r_n\}_{n=1}^{|R|}$ in which $r_n \in H$, where $n = 1, \ldots, |R|$. In addition, the combination of T and R is defined as

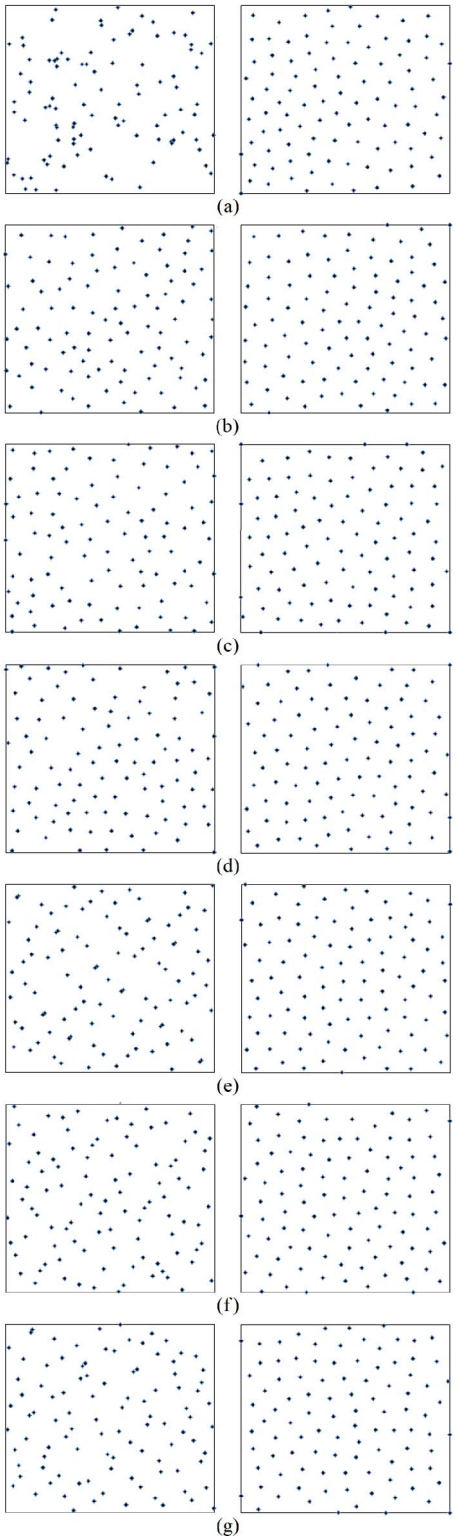

Fig. 5. The (a) RT, (b) FSCS, (c) RRT, (d) EAR, (e) Sobol, (f) Halton, and (g) Niederreiter on the left and corresponding RBCVT points on the right.

$TR = T \cup R = \{tr_m\}_{m=1}^{|TR|}$, where $|TR| = |T| + |R|$. Each tr_m has an associated Voronoi cell named V_m.

- Step 3: Initialize a random background point set of $B = \{b_j\}_{j=1}^{|B|}$ in which $b_j \in (I \cup H)$, where $j = 1, \ldots, |B|$.
- Step 4: Cluster the B into $|TR|$ cells such that $b_j \in V_m, tr_m = \beta(b_j, TR)$.
- Step 5: Calculate the centroids of Voronoi regions only for those V_m where the generator belongs to T, denoted by V_i (we do not need to update border points). For the probabilistic approach, (8) is simplified to $t_i^* = \frac{\sum_{b_j \in V_i} b_j}{\sum_{b_j \in V_i} 1}$, where ρ is set to a unit value in this application.
- Step 6: Update the generators, t_i, where $i = 1, \ldots, |T|$ are replaced with the corresponding t_i^*.
- Step 7: Go to step 3 until the stoppage criterion is met.

A stopping criterion can be 1) the distortion value between t_i and t_i^*, $i = 1, 2, \ldots, |T|$, in each iteration, is reduced to less than a threshold, or 2) a constant number of iterations. Within this study, a constant number of 10 iterations has been selected. This stopping criterion was selected due to its perceived convergence among all trial runs of the algorithm. The parameter $|B|$ was set relative to the value of $|T|$, $100 \times |T|$. It has been observed that with 10 iterations and considering $|B| = 100 \times |T|$, the produced RBCVT test cases are in a stable situation and no further iterations were required to more uniformly distribute the generators. Finally, we need to specify how to generate random border points of R. As indicated in Fig. 4, we considered a set of square cells around I as H and a random point is inserted in each cell. The number of cells in each side of I is selected in accordance with the $|T|$ as $\alpha \times \sqrt{|T|}$, where α is a coefficient which is selected as $\alpha = 2$ based upon an initial empirical exploration. Accordingly, $|R| = 4 \times \alpha \times \sqrt{|T|}$. Finally, the h which is defined as the width of H, indicated in Fig. 4, is equal to a side of a square cell.

5.1.1 RBCVT Runtime Analysis

In this section, we discuss the order of computational complexity of the RBCVT algorithm. In each RBCVT iteration, the main computational load is associated with clustering the set B (Step 4). Since each b_j is clustered by comparing it to the all members of TR, each b_j clustering complexity grows linearly with $|TR|$ given by $RBCVT(|TR|)_{b_j} \in O(|TR|)$. Obviously, the runtime order of RBCVT is also dependent on $|B|$ and the number of iterations (held constant in this study); hence $RBCVT(|TR|, |B|) \in O(10 \times |TR| \times |B|)$. Since the number of $|B| = 100 \times |T|$ grows linearly with $|T|$ and $|TR| = |T| + |R|$, the previous equation can be simplified as $RBCVT(|T|, |R|) \in O(1{,}000 \times |T| \times (|T| + |R|))$. However, the constant number of 1,000 becomes insignificant as $|T|$ grows. As a result, $RBCVT(|T|) \in O(|T|^2 + 4\alpha|T|^{1.5})$. Finally, we need to keep the term with highest order. Therefore, the runtime complexity of RBCVT grows within the order of quadratic time as $RBCVT(|T|) \in O(|T|^2)$.

5.2 RBCVT's Runtime Order Reduction (RBCVT-Fast)

The runtime of $O(|T|^2)$ which was calculated for the RBCVT method in the previous section is the basic calculation method without any algorithmic optimizations. Hence, in this section, we propose an optimized RBCVT calculation method (RBCVT-Fast) using a novel search algorithm to generate test cases with a linear runtime given by $RBCVTFast(|T|) \in O(|T|)$. Although there are some

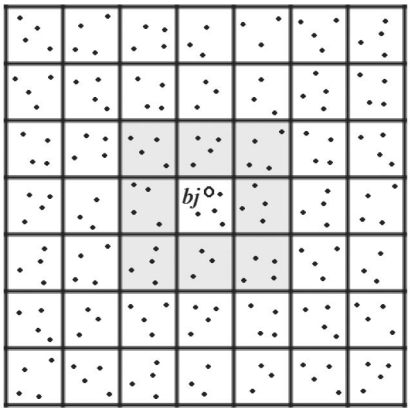

Fig. 6. A grid divides $H \cup I$ into a set of cells. The points are tr_m, $m = 1, \ldots, |TR|$, and the circle is b_j. Cells in layer one regarding b_j are highlighted as an example.

special search algorithms like R^*-tree [56], none of them are appropriate for our application. The steps of the new algorithm are similar to the previous section, with an additional preprocessing step after step 3 that we call step 3B to prevent the renumbering of steps. Furthermore, step 4's calculation procedure is updated with a new algorithm.

Each b_j in step 4 is clustered by comparing it to the all members of TR given by $\beta(b_j, TR)$. This process produces a linearly growing runtime with $|TR|$ for clustering b_j, given by $RBCVT(|TR|)_{b_j} \in O(|TR|)$. In contrast, we propose a novel search algorithm, specifically designed for RBCVT, which results in a constant runtime for clustering each b_j. In other words, b_j clustering runtime is independent from the size of TR or T, and we will find the nearest tr_m to the b_j by comparing b_j to a constant number of points in TR.

5.2.1 Preprocessing Step

This section explains step 3B of the RBCVT-Fast algorithm that is intended to prepare tr_m, $m = 1, \ldots, |TR|$, for the search algorithm (proposed in the next section). As indicated in Fig. 6, the preprocessing step involves defining a grid on $H \cup I$ which divides $H \cup I$ into a set of cells, called grid cells. Consequently, each tr_m is placed in one of the cells, which is referred to as the parent cell for that tr_m. All the tr_m points that are in a cell are called child points of that cell.

In the preprocessing step, we determine each cell's child points and store them in an array. The parent cell of each point is simply determined from the point's coordinates. The critical parameter in the preprocessing step that affects the runtime of RBCVT-Fast is C_{avg}, which must be a constant for any size of TR. We have informally (empirically) observed that $C_{avg} = 20$ produces the most efficient algorithm with respect to runtime. Having the C_{avg} value, we can calculate the number of cells in each dimension, G_N, given by

$$G_N = Round\left(\sqrt{\frac{|T|}{C_{avg}}}\right). \tag{9}$$

Consequently, the total number of cells in a 2D space is $G_N \times G_N$.

```
Begin
    l ← 0 // l denotes the layer number
    MD ← 1 // MD indicates minimum distance
    while dist_l(b_j, l) < MD do
      for each cell in C_l do
        if dist_c(b_j, c_lm) < MD then
          if dist(b_j, β_c(b_j, c_lm)) < MD then
            tr_winner ← β_c(b_j, c_lm)
            MD ← dist(b_j, tr_winner)
          end if
        end if
      end for
      l ← l + 1
    end while
end
```

Fig. 7. Pseudocode for the proposed search algorithm utilized in the RBCVT-Fast algorithm.

5.2.2 A Novel Search Algorithm

In this section, a novel search algorithm is discussed which reduces the linear runtime order of clustering b_j to a constant runtime. The main idea behind this search algorithm is that we do not need to compare the b_j with all of the tr_m. As indicated in Fig. 6, to find the nearest point to b_j, we need to calculate the distance between b_j and the children of the adjacent cells, not all the cells. That is, we need to compare b_j with the children of C_l (a set which contains all the cells in layer l), where l starts from zero. Layer l includes all the cells that have a similar Chebychev distance from the cell with b_j as a child. The highlighted cells in Fig. 6 are in layer one. This algorithm starts by calculating $tr_{winner} \leftarrow \beta_c(b_j, c_{lm})$ for layer zero, where each cell of C_l is denoted by c_{lm} (c_{lm} for layer zero is only one cell which is the cell parent of b_j). Then, we check that tr_{winner} is the nearest point to b_j by comparing $dist(b_j, tr_{winner})$ with $dist_l(b_j, 1)$. If $dist(b_j, tr_{winner}) < dist_l(b_j, 1)$, then the process is finished and tr_{winner} is the nearest point of TR to b_j. Otherwise, we have to compare b_j with the children of layer one's cells and update tr_{winner} in case we found a closer point to b_j. To reduce the runtime complexity, b_j is only compared with the children of those cells in layer one that $dist_c(b_j, c_{lm}) < dist(b_j, tr_{winner})$. This process will continue until we find the nearest point to b_j. Pseudocode for the proposed search algorithm is indicated in Fig. 7.

5.2.3 RBCVT-Fast Runtime Analysis

Although the proposed search algorithm does not guarantee that finding the nearest point to b_j is accomplished by comparing b_j with a constant number of points, empirical investigations have indicated that the average number of comparisons stays constant independent from the size of TR. Similarly, since $|TR|$ is only dependent to $|T|$, the average number of comparisons is independent from $|T|$. Fig. 8 represents the average number of points and cells compared to b_j in order to find tr_{winner} in an RBCVT-Fast calculation, where an RT test set is utilized as generator points. This graph is presented for different sizes of T with respect to the optimized $C_{avg} = 20$. Since considering other ART and QRT approaches as initial generator points

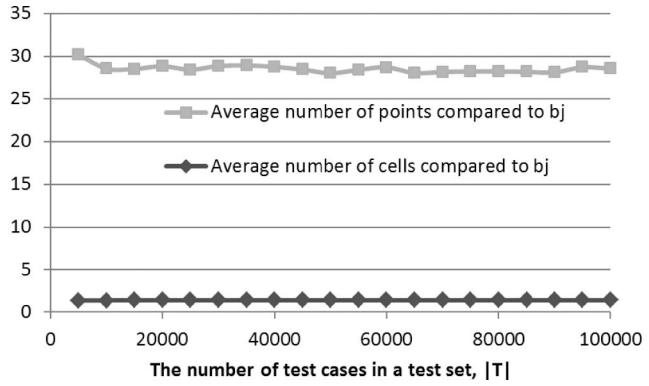

Fig. 8. Average number of points/cells that are compared to b_j calculating the nearest point of TR to b_j in an RBCVT-Fast calculation, where an RT test set is utilized as generators.

revealed similar results with RT as initial generator points, we only included RBCVT with RT as generator points to avoid duplication.

As indicated in Fig. 8, we have produced a search algorithm that, on average, requires a constant number of comparisons to calculate $\beta(b_j, TR)$, leading to $RBCVTFast(|TR|)_{b_j} \in O(1)$. Another distinction between RBCVT and RBCVT-Fast regarding runtime is the preprocessing step that is included in the RBCVT-Fast. Obviously, $Preprocessing(|TR|) \in O(10 \times |TR|)$, where 10 indicates the number of iterations. Accordingly, the total RBCVT-Fast runtime order is $O(10 \times 1 \times |B| + 10 \times |TR|)$. Similar to the discussion in Section 5.1.1, this runtime order can be simplified as $O(1{,}000|T| + 10|T| + 10|R|) = O(1{,}010|T| + 40\alpha\sqrt{|T|})$. Since we need to keep the term with highest order, the final runtime of the RBCVT-Fast algorithm is linear, given by $RBCVTFast(|T|) \in O(|T|)$. The linear runtime is also investigated in empirical runtime analysis section.

5.3 Generalization of the RBCVT beyond Two Dimensions

The concept of the RBCVT is not limited to a 2D hypercube. As defined in Section 4 in (7), the Voronoi region related to t_i is all the areas that are closer to t_i than any other point. Obviously, we can observe from the definition that the Voronoi region can be of any dimension having an appropriate d-dimensional distance function. The distance function used in this study is euclidian (l^2-norm), which can be used in any dimension. To analyze the calculation of RBCVT for higher dimensions, we go through the steps presented in Section 5.1, as well as the RBCVT-Fast calculation method as follows:

- The initial generator set (T) in step 1, which is the result of other test case generation approaches (RT, ARTs, and QRTs), can be of any dimension since RT, ARTs, and QRTs can produce test cases beyond two dimensions.
- To generate the random border points (R) in step 2, we define a set of cells around the d-dimensional input space hypercube and then we insert a random point in each cell which is straightforward. The number of cells in each dimension of the input space is selected as $\alpha \times \sqrt[d]{|T|}$. Accordingly, each side of the input space hypercube has $(\alpha \times \sqrt[d]{|T|})^{d-1}$ cells since the dimension of each side of a d-dimensional unit hypercube is $d-1$. Finally, a d-dimensional unit hypercube has $2 \times d$ sides, leading to the following equation for the number of cells, which covers all borders of the input space:

$$|R| = 2 \times d \times \left(\alpha \times \sqrt[d]{|T|}\right)^{d-1}. \tag{10}$$

- The background points (B) in step 3 are easy to generalize to higher dimensions since we only need d-dimensional random numbers.
- In step 3B regarding the preprocessing step of the RBCVT-Fast, we can define the grid on d dimensions rather than a 2D hypercube. Then, each d-dimensional tr_m can be assigned to a cell of the grid. In addition, G_N for the d-dimensional hypercube can be calculated by

$$G_N = Round\left(\sqrt[d]{\frac{|T|}{C_{avg}}}\right). \tag{11}$$

- In the nonoptimized RBCVT approach, Step 4 is easy to calculate in any dimension as we compute the distance of each b_j with all tr_m, $m = 1, 2, \ldots, |TR|$, with the d-dimensional Euclidian distance function. The algorithm of this step in the RBCVT-Fast is exactly equal to the pseudocode presented in Fig. 7. The only changes are the generalization of $dist_l(b_j, l)$, $dist_c(b_j, c_{lm})$, and $\beta_c(b_j, c_{lm})$ into d dimensions. All of these functions require a d-dimensional euclidian distance function which is available.
- Finally, steps 5-7, including the calculation and updating of the centroids (t_i^*), can be calculated for any dimension.

In many real-world programs, a simple d-dimensional input may not be a suitable model of a program under the test. For example, when using a structured input such as trees, graphs, and lists, this structure needs to be embedded into each test case. In such cases, other extensions are required to generalize the input test cases which are outside of the scope of this research. As an example, a grammar can be utilized to describe the structures embedded in the input, and RBCVT can be utilized to assist with enumerating this grammar.

5.3.1 Runtime Analysis of d-Dimensional RBCVT

Looking precisely to the nonoptimized RBCVT algorithm, one can observe that the only process dependent on the dimension is the distance function and its runtime changes linearly with d. The number of comparisons is independent of d, leading to $RBCVT(d, |T|) \in O(d \times |T|^2)$. This indicates a linear increase in $RTime(RBCVT)$ as d grows.

On the contrary, the order of $RTime(RBCVTFast)$ is not linear with d since the number of required comparisons grows as d increases. The increasing number of cells in layer l as d increases is the cause of this issue. The number of cells in layer l increases exponentially as d grows, leading to exponential increase in the number of distance comparisons.

In addition, each distance comparison runtime grows linearly with d. As a result, the order of $RTime(RBCVTFast)$ is given by $RBCVTFast(d, |T|) \in O(d \times E^d \times |T|)$, where E is a constant. Note that in a given d, $RTime(RBCVTFast)$ is still linear regarding $|T|$.

Although the runtime complexity of RBCVT-Fast with respect to d is higher than the nonoptimized RBCVT, the runtime complexity for RBCVT-Fast is lower with respect to $|T|$ than the nonoptimized RBCVT. Combining these two observations results in $RTime(RBCVTFast) \leq RTime(RBCVT)$ for any $|T|$ and d. That is, the number of comparisons in the RBCVT-Fast is less than or equal to the nonoptimized RBCVT algorithm. According to (11), G_N reduces when d increases with constant C_{avg} and $|T|$, leading to an increasing $\frac{RTime(RBCVTFast)}{RTime(RBCVT)}$. With $G_N = 1$, the RBCVT and RBCVT-Fast are exactly equal since there is only one cell in the hypercube. In $G_N = 3$, the runtime of both approaches are similar since the RBCVT-Fast uses layers 0 and 1 on average to find the nearest point. As G_N increases, the runtime effectiveness of the RBCVT-Fast grows compared to the nonoptimized RBCVT algorithm. To summarize, $RTime(RBCVTFast) \ll RTime(RBCVT)$ when $G_N \gg 3$, leading to $\sqrt[d]{\frac{|T|}{C_{avg}}} \gg 3$, which is concluded from (11). Therefore, when the number of test cases is large enough, RBCVT-Fast algorithm is more efficient than nonoptimized RBCVT algorithm regarding time complexity.

6 Experimental Frameworks

The study conducted to investigate effectiveness of RBCVT against ART and QRT methods is described in this section. We have designed two experimental frameworks: a simulation-based and a mutant-based software testing framework. The simulation framework utilizes three failure patterns derived from empirical studies [16], [17], [18], [19], [20] investigating defect types. The mutant-based software testing framework simulates defects in software by producing mutants within the code in a systematic fashion [57]. For the mutant-based software testing framework, we utilize the Briand and Arcuri [57] framework; this framework has been accepted via publication as a valuable mechanism for empirically exploring such mechanisms. This framework is based on 11 short mathematical programs that appear in the ART literature [20]. Both frameworks require an effectiveness measure to evaluate the results which is discussed in the following section.

6.1 Testing Effectiveness Measure

There are three well-known testing effectiveness measures, E-measure, P-measure, and F-measure. The E-measure is defined as the expected number of detected failures in a series of tests. Assuming the probability of a test case to detect a failure is θ, similar to a random test case, then the E-measure and its standard deviation are [58]

$$Emeasure = |T| \times \theta, \tag{12}$$

$$std = \sqrt{\theta |T| (1 - \theta)}. \tag{13}$$

The P-measure is defined as the probability of at least one failure being detected within a test set. Considering the number of test sets as M_t and the number of test sets that detect at least one failure as M_{fault}, the P-measure can be estimated as M_{fault}/M_t. In addition, in RT, the P-measure is equal to [58]

$$Pmeasure = 1 - (1 - \theta)^{|T|}. \tag{14}$$

The standard deviation associated with the calculation of a P-measure for RT can be approximated by Chen et al. [58]:

$$std \approx \sqrt{(1 - \theta)^{|T|} - (1 - \theta)^{2|T|}}. \tag{15}$$

The last testing effectiveness measure is the F-measure, which is defined as the number of test cases required to detect the first failure within the input domain. Chan et al. [41] have indicated that for RT the expected value of the F-measure is equal to θ^{-1}. The sampling distribution of the P-measure and the E-measure can be approximated by the normal distribution [58], whereas the probability distribution of the F-measure is geometric [58].

The main question that should be answered is: Which of these measures best characterizes software testing? Since the software testing trend is toward automating the process, selecting a measure that best represents the operation of an Automated Testing System (ATS) is essential. When we consider the "desirable" aspects of automated software testing with respect to RT, ART, QRT, or RBCVT, it does impose certain constraints on the measurement process that must be adhered to:

- ATS is intrinsically an automated technique, at least on the test case generation side. This implies that the traditional incremental cost of manual production of a new, additional test case is minimized. ATS is characterized by
  - a tester selecting an arbitrary large number of test cases to be produced, and
  - the ATS system producing the required volume of test cases.
- Test case generation often seeks to generate values with a specific purpose, while we can generate truly random values and exercise them against the entire system. The huge dimension of the input space for modern software systems tends to imply that this "scatter gun" approach is ineffective. Instead, the tester will often have a specific testing objective and will attempt to generate a specific set of test cases under specific circumstances that answer this question. That is, the tester tends to test aspects of the system or subcomponents of the system rather than blindly "attacking" the entire system. For example, automated security testing investigates an aspect of the system and automated unit testing explores a subcomponent. Accordingly, the tester will require a large volume of test cases, possessing limited dimensions, which are cost-effective for an automated testing process.
- These large volumes of test sets are automatically applied to the system under test and the "outputs" from the system are automatically captured. The system under test is normally placed into a known state before each execution commences. The large

volume of test cases implies that manual application of the test data is not a realistic option.

- This input process results in large volumes of test results, again implying that the manual examination of every test result is prohibited by cost. Instead, two options are commonly deployed:
  - A Test Oracle is constructed. The test oracle typically has a simplified description of a defect. Does the system crash or not is an example of such a description. Here, each crash is considered a "defect." The oracle either stops after finding the first crash or collects all of the crashes. Data about the crashes are presented to the tester for analysis. If the oracle collects multiple crashes, the system has no mechanism to understand if these crashes have the same root cause or are in fact independent. The tester may select to only investigate a subset of these multiple crashes to avoid excessive, potentially redundant (when crashes are in fact dependent) costs.
  - The output is investigated manually as a single integrated entity. Here, the test results, or shorter proxies of the results, are sent to a log file or other recording mechanism. The tester inspects this mechanism after all the test runs are finished. Here, the tester is looking for output values that look anomalous. Again, the tester may select one or more test results to explore more closely, However, the number of test results explored is always small to ensure a cost-effective process.

The above description of ATS is in correspondence with many ATS systems reported in the literature, including [8], [59], [60]. Accordingly, it is believed that this process is well characterized by the E- or the P-measure rather than the F-measure. That is, the incremental viewpoint of the F-measure is not supported by the operation of these automated testing systems [8], [59], [60] in the operational profile discussed. Since in software testing failure areas tend to be clustered [21], [17], [18], [37], detecting multiple failures are often redundant as it is indicative of multiple test cases discovering the same defect. This argument strongly suggests the use of the P-measure over the E-measure. Therefore, the P-measure is utilized in this study as an appropriate effectiveness measure for automated software testing.

Chen et al. [58] demonstrate that the F-measure has better statistical power than the P-measure. However, this "performance difference" tends to zero as the number of measurements tends to infinity. It is believed that the above analysis effectively implies that this difference is essentially zero at the number of measurements utilized within this study.

6.2 Parameters of Test Case Generation Methods

A number of parameters are associated with each ART algorithm, which are considered constant through all the experiments. We selected the value of these parameters as recommended in their respective works. The k in FSCS method, representing the number of randomly selected candidates, is held constant at $k = 10$ based on the recommendation of Chen et al. [25]. Similarly, the coverage ratio in the RRT method is considered constant at 1.5 due to the recommendation of Chan et al. [37]. The EAR method [33] has several parameters regarding the evolutionary approach which are set to identical values to those reported in the original work [33]. The k (population size) has been set to 20 and the probability of crossover is set at 0.6. Furthermore, the probability of mutation is considered as 0.1, the size of the mutation was set at 0.01, and the stopping criterion is set to the constant number of 100 iterations. The parameters associated with RBCVT are in accordance with the values discussed in Section 5.1, the number of background points is set to $100 \times |T|$, and the number of RBCVT iterations is equal to 10 for all the tests.

6.3 Simulation Framework

For the simulation framework, we will introduce the utilized failure patterns, the failure rate associated with each failure pattern, the number of test cases in each test set, and the number of test sets. These features are discussed in the next two sections.

6.3.1 Failure Patterns and Failure Rates

To be able to evaluate test case generation methods, we need to consider some parts of the input domain as a failure area, where a failure is produced when a test case is placed in this area. Several works have performed an empirical investigation through failure patterns within the input domain [16], [17], [18], [19], [20]. White and Cohen [16] indicated that failures usually occur on or near the boundary of (sub)domains. As a result, failure areas form types of strip patterns since domain boundaries form lines or hyper planes. Ammann and Knight [17] explain that failure regions seem to be locally continuous. They present 2D empirical failure patterns that possess similarities to rectangular geometry. Similarly, Finelli [18] describes that there are continuous regions, called error crystals, that produce failures. Bishop [19] also explains continuous failure regions that are much more angular and elongated than a pure "blob" [20]. Schneckenburger and Mayer [20] have analyzed the failure area geometry in a systematic way using three numerical programs, each possessing a 2D input space. They presented strip faulty patterns for all three programs under test. Therefore, significant empirical evidence exists that failure areas are clustered into a contiguous region within the input domain and that they produce error crystals or failure regions.

While we cannot generalize one software failure pattern to others, researchers have empirically indicated common characteristics between failure patterns. Accordingly, Chan et al. [21] have introduced three common types of failure patterns, shown in Fig. 9 (the block, strip, and point failure patterns). We have selected these patterns as a testing framework since the empirical studies support the use of these patterns as an approximation to real software failures. Although these failure patterns are not real, these patterns are believed to best represent multiple clustered values in the input domain, which, in general, imply a single root cause failure.

The main parameter associated with each pattern is a failure rate (θ) which is the total failure area divided by the total area of the input domain. In this research, failure rates

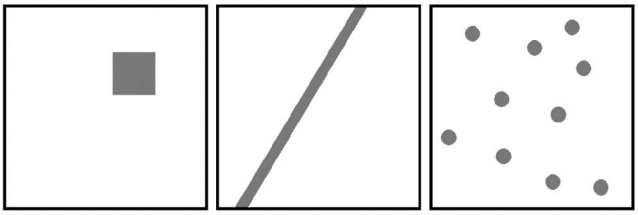

Fig. 9. Typical 2D failure patterns: (a) block, (b) strip, and (c) point failure patterns [21].

of $\theta = 10^{-2}$, 10^{-3}, 10^{-4}, and 10^{-5} have been considered as a basis to analyze testing strategies effectiveness. In the software testing literature [33], [43], failure rates between 10^{-2} and 10^{-3} are usually investigated, whereas in real-life applications the failure rates may be lower. Considering the fact that the average programmer introduces 5 to 10 defects per Kilo Line Of Code (KLOC) [22], θ is certainly nonzero. However, no reliable industrial information exists on θ. Hence, we include the failure rates of 10^{-4} and 10^{-5} to explore a wider range of values.

Although the implementation of these three failure patterns is straightforward, implementation details are included for the sake of completeness. The block pattern is generated by randomly choosing a point in I and then a square is constructed around this point with respect to the failure rate. Due to the section of the random point near the boundaries of I, the constructed block pattern may not fit within I. In this situation, this pattern is disregarded and another random point is selected until a valid block pattern is generated. The strip pattern is generated using a random point in I and a random angle associated with a line passing over the selected random point. The width of the strip pattern is calculated according to the failure rate. This strip pattern generation method is different from the method introduced by Chen et al. [58], whereby one point is selected on the vertical boundary and another point on the horizontal boundary of I. Then, the strip pattern is generated by connecting the two points and calculating the width of the line using θ. Unfortunately, we observed that this implementation does not produce a uniform distribution of strip patterns—with an excessive concentration of points near the boundaries compared to the middle of I. To generate the point pattern, 10 random points were selected within I. A circular area is constructed around each point so that the sum of these circular areas is equal to failure rate. Similar to the block pattern, if a circular area is not within I, the associated random point is disregarded and another random point is selected.

In short, the block and point patterns are in-line with those used in the literature [35], [33], [43], [58], and the strip pattern is redefined to overcome traditional limitations and produce a uniform distribution of the strip pattern.

6.3.2 Number of Tests

Due to the random nature of test case generation methods, we generated $M_t = 100$ distinct test sets for RT, FSCS, RRT, EAR, and, accordingly, RBCVT to evaluate the effectiveness of each approach using the P-measure. Therefore, a P-measure is evaluated using 100 tests for a specific failure pattern. In addition to test set generation, the failure patterns are also generated randomly. Hence, we generated $M_f = 10{,}000$ random failure patterns, leading to 10,000 P-measure results which are normally distributed [58] between zero and one. Therefore, $M_f = 10{,}000$ statistics are used to evaluate the mean and standard deviation of the normally distributed P-measure for each approach, at each failure rate, and with each of the three failure patterns.

QRT methods are deterministic and hence each method produces a unique test set. Therefore, to draw statistical analysis with the same population size, for each QRT method, we generated a sequence of test cases where the length of this QRT sequence is M_t times larger than the test set size. Then, we split this sequence into M_t test sets which result in distinct test sets. So all the approaches have been tested using $M_f = 10{,}000$ P-measure results, each calculated by $M_t = 100$ measurements.

In addition to failure pattern type and θ (10^{-2}, 10^{-3}, 10^{-4}, and 10^{-5}), to evaluate a P-measure we need to set the number of test cases in each test set ($|T|$). The best $|T|$ to analyze the test case generation approaches using the P-measure is the worst case in terms of the standard error, which can be estimated as

$$SE = \frac{std}{\sqrt{M_t}}. \tag{16}$$

Since M_t is a constant number, worst case SE leads to maximizing the standard deviation. According to Chen et al. [58], the maximum standard deviation of P-measure calculation is 0.5. Solving (15) as $std = 0.5$ results in $|T|$ based on θ as follows:

$$|T| = \frac{\log(0.5)}{\log(1-\theta)}. \tag{17}$$

Since $\theta = 10^{-2}$, 10^{-3}, 10^{-4}, and 10^{-5} have been chosen for the experimental test, the respective values for $|T|$ are 68.97 (69), 692.80 (693), 6,931.12 (6,931), 69,314.37 (69,314). Since $|T|$ is an integer value, the rounded values are given in the brackets. Finally, all the generated test cases are within I and every test case consists of a floating-point number, with double precision, for each dimension.

6.4 A Mutant-Based Software Testing Framework

To evaluate the proposed RBCVT approach on a testing framework which utilizes independently produced programs, we selected the mutant-based software testing framework introduced by Briand and Arcuri [57]. This framework is outlined in detail in [57, Section 4]. For the sake of completeness, we present a summary of the main features of this framework. This work utilizes 11 programs, written in Java, which implements basic mathematical functions that appear in the ART literature [20]. We directly utilized their source code without any modification. Their framework utilizes mutation analysis to produce a large number of faults in a systematic fashion [57]. They produced 3,727 mutants for the 11 programs using muJava [61]. Further, in [57], the P-measure is utilized to evaluate these mutants against RT and ART test sets, where the size of test sets varies between 1 and 50.

This framework assumes an input space of each program, an integer value in the range of $[0, 2^{24/d} - 1]$ for each dimension (d). This leads to 2^{24} input possibilities for

each program. The framework first measures each mutants failure rate by testing all possible 2^{24} states, so they could measure failure rates as low as 2^{-24}. Then, those mutants that revealed no failure or had the failure rate over 0.01 were removed. Therefore, they kept 780 appropriate mutants with $2^{-24} \leq \theta \leq 0.01$.

In this study, we use these 780 mutants to test the effectiveness of the proposed test case generation approach. Since we assume that we do not know the failure rate of the programs under the test, we apply four test set sizes, including $|T| =$ 10, 20, 50, and 100 to each mutant to evaluate the effectiveness of each test case generation approach. Accordingly, the P-measure is evaluated for each test case generation approach for discussed test set sizes. To evaluate a P-measure, we tested each mutant using 100 distinct test sets and then the average over all the mutants is calculated as a P-measure. To draw a statistical analysis, we repeated this P-measure evaluation 100 times, leading to 100 statistics that are used to evaluate the mean and standard deviation of the normally distributed P-measure [58] for each approach, at each test set size. To draw a statistical analysis with the same population size for QRT methods, we utilized a similar procedure as described in the simulation pattern where a longer sequence of QRT test cases is split to generate distinct test sets.

This process leads to the execution of over 78 billion test cases which took more than a month on a Intel dual-core Processor E6300 (2.8 GHz) with 8 GB of RAM.

7 Experimental Results and Discussion

7.1 Formal Analysis

Since P-measure values are normally distributed [58], Tables 2, 3, 4, and 5 present statistical parameters reflecting the effectiveness of RT, ARTs, QRTs, and the corresponding results after the RBCVT process. In addition, the following parameters were calculated:

1. a test of statistical significance (z-test, one-tailed; our working hypothesis is that RBCVT will produce superior results) with a conservative type I error of 0.01, and
2. an effect size (Cohen's method [62], [63]) which indicates "size" discrepancy between two statistical populations given by

$$effectsize = \frac{\mu_2 - \mu_1}{\sqrt{\frac{(n_2-1)std_2^2+(n_1-1)std_1^2}{n_2+n_1}}}, \qquad (18)$$

where μ, std, and n represent the mean, the standard deviation, and the number of elements within the populations, respectively.

In this study, a positive value of effect size represents the size of the improvement that has been achieved by applying the RBCVT process. Cohen [62], [63], [64] defines the standard value of an effect size as small (0.2), medium (0.5), and large (0.8). Effect size can also be interpreted as the average percentile standing, which indicates the relative position of the two populations. Similarly, effect sizes can be interpreted in terms of the percent of the nonoverlapped portion of the populations. Corresponding values are presented in Table 1.

TABLE 1
Cohen's Effect Size Description (Large, Medium, and Small) as Well as Corresponding Values for Percentile Standing and Percent of Nonoverlapped Portion of Two Populations [64]

Cohen's Description	Effect Size	Percentile Standing	Percent of Non-overlap
	2.0	97.7	81.1
	1.5	93.3	70.7
	1.0	84	55.4
Large	0.8	79	47.4
Medium	0.5	69	33.0
Small	0.2	58	14.7
	0.0	50	0.0

7.2 Block Pattern Simulation Results

Table 2 indicates the testing effectiveness of all the studied approaches and the corresponding results after the RBCVT process was applied with respect to the block failure pattern. This table demonstrates that performing the RBCVT process on the outputs of other methods has a positive effect on the P-measure since RBCVT consistently provides statistically significant improvement. The amount of improvement in terms of effect size is larger than the highest Cohen's description (Large) in most of the cases, only RRT at $\theta = 10^{-4}, 10^{-5}$ and EAR at $\theta = 10^{-5}$ have effect size between large and medium.

Comparing the amount of improvement (effect size) among all approaches in Table 2, one can observe that the largest RBCVT improvement belongs to the RT for all failure rates. In contrast, no individual method has the smallest increase in effectiveness regarding the effect size; EAR has the smallest improvement for $\theta = 10^{-2}$ and 10^{-3} and RRT for $\theta = 10^{-4}$ and 10^{-5}. Fig. 10 indicates the improvement of each approach after the RBCVT process comparing to the effectiveness of test cases used as inputs to the RBCVT process (effect size) with respect to block pattern at each failure rate. In this figure, in all methods, the level of changes before and after the RBCVT process is decreasing as the failure rate decreases.

In Table 2, the mean values of the P-measures appear dissimilar for the different approaches, whereas the corresponding results after the application of the RBCVT process represents a sizable reduction of the variation between these values. Therefore, for comparison of RBCVT, as a single method, against all other approaches, we assume the average RBCVT results as the performance of the RBCVT approach. Fig. 11 represents the effect size of the testing effectiveness at each strategy against RT in the block pattern simulations. Contrasting RBCVT against FSCS, RRT, EAR, Sobol, Halton, and Niederreiter, this figure highlights the increased efficiency of RBCVT regarding the block pattern. Another conclusion from this figure is that all of the testing methods outperformed RT at every failure rate with respect to the block pattern.

7.3 Strip Pattern Simulation Results

Testing effectiveness results regarding the strip failure pattern are shown in Table 3. The results demonstrate that

TABLE 2
The P-Measure Testing Effectiveness Mean and Standard Deviation for All Approaches Including the Corresponding Results After the RBCVT Process as Well as Effect Size, Z-Score, and Significance Value with Respect to Block Pattern

$\theta, \lvert T\rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4972	0.0565	0.7484	0.0468	4.8423	444.91	0.0000
	Sobol	0.5694	0.0627	0.7668	0.0432	3.6657	314.83	0.0000
	Niederreiter	0.5833	0.0568	0.7705	0.0432	3.7113	329.66	0.0000
	Halton	0.6118	0.0531	0.7742	0.0424	3.3799	305.86	0.0000
	FSCS	0.5953	0.0512	0.7629	0.0434	3.5296	327.00	0.0000
	RRT	0.6021	0.0527	0.7623	0.0435	3.3164	304.20	0.0000
	EAR	0.5951	0.0534	0.7538	0.0442	3.2367	297.28	0.0000
10^{-3}, 693	RT	0.5006	0.0502	0.7016	0.0460	4.1721	400.12	0.0000
	Sobol	0.5769	0.0554	0.7121	0.0448	2.6832	244.12	0.0000
	Niederreiter	0.5612	0.0584	0.7122	0.0452	2.8918	258.69	0.0000
	Halton	0.5863	0.0504	0.7127	0.0452	2.6399	250.74	0.0000
	FSCS	0.6242	0.0484	0.7096	0.0454	1.8200	176.60	0.0000
	RRT	0.6407	0.0478	0.7100	0.0452	1.4891	145.01	0.0000
	EAR	0.6420	0.0484	0.7092	0.0453	1.4335	138.86	0.0000
10^{-4}, 6931	RT	0.5000	0.0494	0.6830	0.0465	3.8154	370.49	0.0000
	Sobol	0.5442	0.0595	0.6850	0.0460	2.6474	236.61	0.0000
	Niederreiter	0.5687	0.0525	0.6850	0.0463	2.3497	221.41	0.0000
	Halton	0.5908	0.0497	0.6847	0.0461	1.9564	188.72	0.0000
	FSCS	0.6328	0.0477	0.6915	0.0467	1.2429	122.91	0.0000
	RRT	0.6555	0.0473	0.6918	0.0461	0.7774	76.72	0.0000
	EAR	0.6456	0.0481	0.6911	0.0461	0.9684	94.87	0.0000
10^{-5}, 69314	RT	0.5004	0.0497	0.6764	0.0460	3.6801	354.54	0.0000
	Sobol	0.5971	0.0556	0.6847	0.0465	1.7100	157.51	0.0000
	Niederreiter	0.5937	0.0511	0.6835	0.0466	1.8349	175.59	0.0000
	Halton	0.5924	0.0492	0.6830	0.0465	1.8929	184.29	0.0000
	FSCS	0.6356	0.0483	0.6854	0.0482	1.0312	102.96	0.0000
	RRT	0.6604	0.0474	0.6873	0.0455	0.5779	56.61	0.0000
	EAR	0.6496	0.0476	0.6853	0.0457	0.7668	75.15	0.0000

for $\theta = 10^{-2}$, RBCVT is statistically significantly superior to all approaches. In contrast, the results for other failure rates suggest similar performance between each approach and the corresponding results after the RBCVT. Although there are differences between the P-measure results of the RBCVT and other approaches at $\theta = 10^{-3}$, 10^{-4}, and 10^{-5}, the results cannot be compared since the level of significance values does not indicate a significant difference between the results in most of the cases.

The magnitude of improvement for the strip pattern at $\theta = 10^{-2}$ is lower than for the block pattern testing effectiveness results since the effect size has been reduced by around an order of magnitude on average. Comparing the amount of improvement among all the approaches in Table 3, again the largest improvement belongs to RT for $\theta = 10^{-2}$ and 10^{-3}. To highlight some strip pattern features regarding the RBCVT approach, Fig. 12 is presented, which indicates the effect sizes between each approach's effectiveness result and corresponding result after the RBCVT

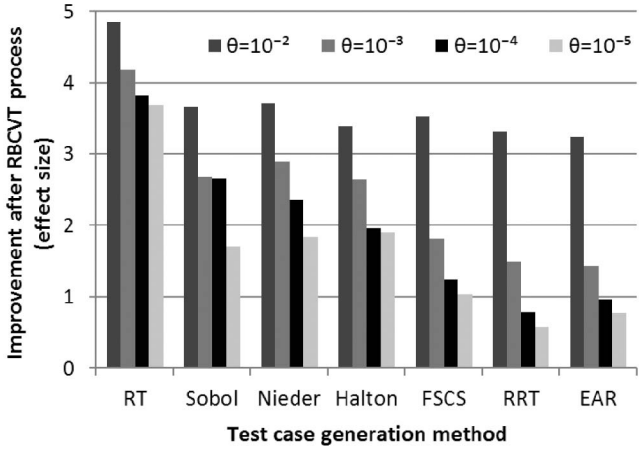

Fig. 10. Improvement of test case generation methods with respect to RBCVT process at different failure rates regarding the block failure pattern.

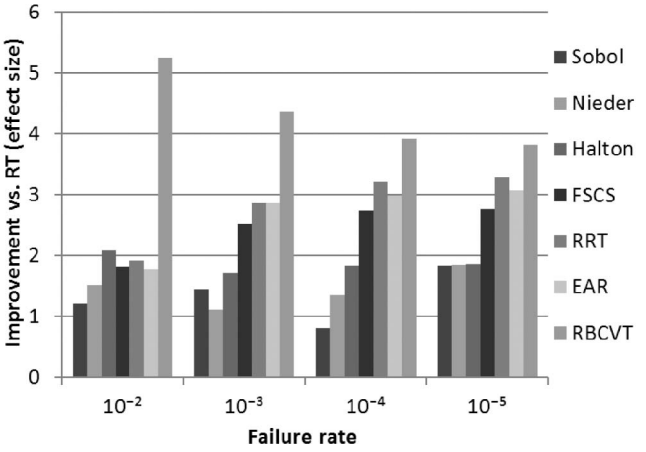

Fig. 11. P-measure testing effectiveness for block pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

TABLE 3
The P-Measure Testing Effectiveness Mean and Standard Deviation for All Approaches Including the Corresponding Results After the RBCVT Process as Well as Effect Size, Z-Score, and Significance Value with Respect to Strip Pattern

$\theta, \lvert T\rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4992	0.0505	0.5192	0.0502	0.3954	39.41	0.0000
	Sobol	0.5214	0.0492	0.5233	0.0502	0.0382	3.86	0.0001
	Niederreiter	0.5165	0.0510	0.5211	0.0503	0.0910	9.05	0.0000
	Halton	0.5172	0.0498	0.5222	0.0494	0.1017	10.13	0.0000
	FSCS	0.5162	0.0505	0.5220	0.0502	0.1168	11.64	0.0000
	RRT	0.5161	0.0503	0.5228	0.0506	0.1323	13.26	0.0000
	EAR	0.5067	0.0503	0.5204	0.0497	0.2729	27.13	0.0000
10^{-3}, 693	RT	0.4997	0.0502	0.5061	0.0501	0.1281	12.80	0.0000
	Sobol	0.5062	0.0499	0.5065	0.0502	0.0054	0.54	0.2929
	Niederreiter	0.5047	0.0501	0.5069	0.0502	0.0450	4.51	0.0000
	Halton	0.5052	0.0495	0.5065	0.0503	0.0248	2.50	0.0062
	FSCS	0.5063	0.0506	0.5069	0.0499	0.0110	1.09	0.1372
	RRT	0.5063	0.0502	0.5068	0.0505	0.0102	1.02	0.1542
	EAR	0.5056	0.0500	0.5068	0.0500	0.0235	2.35	0.0095
10^{-4}, 6931	RT	0.4993	0.0499	0.5024	0.0498	0.0632	6.31	0.0000
	Sobol	0.5027	0.0502	0.5025	0.0501	-0.0043	-0.43	0.3327
	Niederreiter	0.5012	0.0499	0.5030	0.0499	0.0355	3.55	0.0002
	Halton	0.5022	0.0498	0.5024	0.0501	0.0052	0.52	0.2999
	FSCS	0.5020	0.0495	0.5016	0.0503	-0.0094	-0.94	0.1727
	RRT	0.5023	0.0506	0.5018	0.0501	-0.0101	-1.01	0.1573
	EAR	0.5022	0.0501	0.5026	0.0497	0.0081	0.81	0.2101
10^{-5}, 69314	RT	0.5001	0.0501	0.5007	0.0499	0.0120	1.20	0.1156
	Sobol	0.5015	0.0500	0.5010	0.0503	-0.0085	-0.85	0.1967
	Niederreiter	0.5007	0.0494	0.5017	0.0498	0.0213	2.14	0.0161
	Halton	0.5007	0.0499	0.5009	0.0509	0.0040	0.40	0.3436
	FSCS	0.5014	0.0503	0.5008	0.0508	-0.0106	-1.06	0.1439
	RRT	0.5019	0.0505	0.5028	0.0505	0.0192	1.92	0.0277
	EAR	0.5017	0.0502	0.5010	0.0484	-0.0148	-1.46	0.0726

process. Fig. 12 indicates that the impact of the RBCVT process is reducing as the failure rate decreases in most of the cases. This fact as well as the results for $\theta = 10^{-3}$, 10^{-4}, and 10^{-5} suggest that the impact of the RBCVT approach for strip patterns tends to zero as the failure rate tends to zero.

Similar to the block pattern, the strip pattern testing effectiveness results after the application of the RBCVT represents a sizable reduction of the variation among these values compared to the effectiveness of the input test cases to the RBCVT process. Therefore, we again consider the average RBCVT results as the performance of the RBCVT approach creating the possibility of comparing it against all of the test case generation methods. Accordingly, all of the approaches have been compared against RT; these results are provided in Fig. 13. In this figure, one can observe the decreasing trend of testing effectiveness against RT as the failure rate reduces. This leads to similar effectiveness for RT with other approaches with respect to the strip pattern at very low failure rates like 10^{-5}; this is not true for the block pattern (Fig. 11). This can be explained by the intrinsic difference

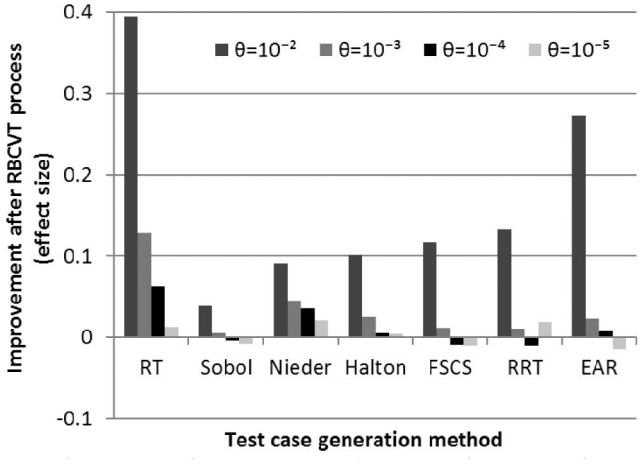

Fig. 12. Improvement of test case generation methods with respect to the RBCVT process at different failure rates regarding the strip failure pattern.

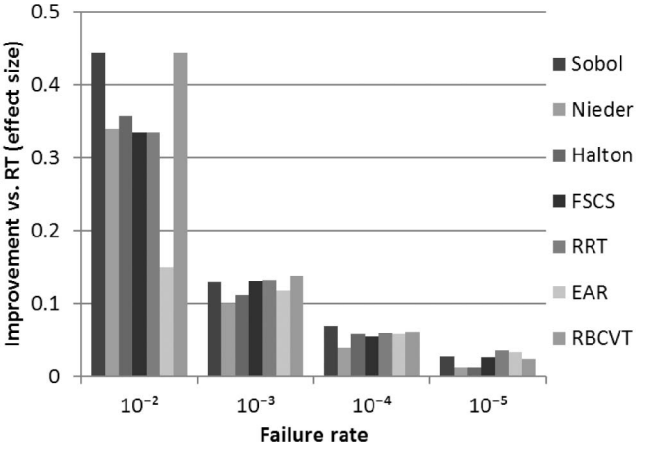

Fig. 13. P-measure testing effectiveness for strip pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

TABLE 4
The P-Measure Testing Effectiveness Mean and Standard Deviation for All Approaches Including the Corresponding Results After the RBCVT Process as Well as Effect Size, Z-Score, and Significance Value with Respect to Point Pattern

$\theta, \lvert T\rvert$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10^{-2}, 69	RT	0.4970	0.0506	0.5247	0.0500	0.5509	54.74	0.0000
	Sobol	0.5083	0.0507	0.5242	0.0501	0.3172	31.54	0.0000
	Niederreiter	0.5064	0.0506	0.5243	0.0499	0.3552	35.28	0.0000
	Halton	0.5080	0.0502	0.5246	0.0493	0.3348	33.17	0.0000
	FSCS	0.4956	0.0506	0.5230	0.0496	0.5470	54.17	0.0000
	RRT	0.4927	0.0507	0.5234	0.0505	0.6081	60.69	0.0000
	EAR	0.4788	0.0507	0.5219	0.0499	0.8581	85.11	0.0000
10^{-3}, 693	RT	0.4994	0.0502	0.5163	0.0500	0.3378	33.73	0.0000
	Sobol	0.5054	0.0516	0.5164	0.0496	0.2171	21.30	0.0000
	Niederreiter	0.5095	0.0509	0.5166	0.0496	0.1420	14.02	0.0000
	Halton	0.5113	0.0498	0.5162	0.0503	0.0984	9.89	0.0000
	FSCS	0.5072	0.0496	0.5158	0.0494	0.1740	17.37	0.0000
	RRT	0.5051	0.0497	0.5166	0.0497	0.2319	23.19	0.0000
	EAR	0.5061	0.0502	0.5164	0.0498	0.2065	20.56	0.0000
10^{-4}, 6931	RT	0.4993	0.0495	0.5130	0.0500	0.2759	27.74	0.0000
	Sobol	0.5080	0.0510	0.5131	0.0502	0.1009	10.01	0.0000
	Niederreiter	0.5107	0.0504	0.5131	0.0500	0.0482	4.80	0.0000
	Halton	0.5112	0.0503	0.5128	0.0499	0.0305	3.04	0.0012
	FSCS	0.5109	0.0500	0.5139	0.0502	0.0604	6.05	0.0000
	RRT	0.5096	0.0501	0.5140	0.0503	0.0870	8.71	0.0000
	EAR	0.5100	0.0496	0.5136	0.0496	0.0728	7.29	0.0000
10^{-5}, 69314	RT	0.5006	0.0500	0.5136	0.0482	0.2659	26.12	0.0000
	Sobol	0.5111	0.0499	0.5128	0.0494	0.0340	3.38	0.0004
	Niederreiter	0.5114	0.0496	0.5126	0.0499	0.0233	2.34	0.0097
	Halton	0.5112	0.0496	0.5125	0.0494	0.0264	2.63	0.0043
	FSCS	0.5111	0.0500	0.5122	0.0485	0.0225	2.21	0.0134
	RRT	0.5114	0.0499	0.5126	0.0502	0.0253	2.54	0.0056
	EAR	0.5115	0.0508	0.5126	0.0522	0.0205	2.08	0.0189

between the strip and the block pattern: As the failure rate decreases, the width of a strip pattern reduces as its length is constant, whereas in the block pattern both dimensions reduce together. Therefore, the similarity between block and strip pattern decreases as the failure rate reduces leading to less testing effectiveness for strip patterns.

7.4 Point Pattern Simulation Results

Point pattern simulations yield results as indicated in Table 4. The presented results suggest an improvement comparing the P-measure results after the RBCVT process was applied. Again the improvements in testing effectiveness, after the RBCVT process was applied, are lower than the corresponding block pattern results. However, in contrast with strip pattern, the RBCVT is statistically significantly superior to all approaches at all failure rates. In addition, the impact of the RBCVT procedure on the test case generation effectiveness regarding point pattern, as indicated by the effect sizes in Table 4, are larger than the equivalent results for the strip pattern.

In Table 4, one can observe that in contrast with the block and strip patterns, the maximum enhancement in testing effectiveness after the RBCVT process does not belong to the RT for all failure rates. EAR has the largest improvement for $\theta = 10^{-2}$, and RT for other failure rates. To further characterize the point pattern results regarding the RBCVT procedure, Fig. 14 provides a graphical representation of the effect sizes in Table 4. This figure indicates that the impact of the RBCVT process regarding the point pattern has a reducing trend as the failure rate reduces for all approaches.

Similar to the previous discussion in Sections 7.2 and 7.3, since the variation among the RBCVT results is quite low, the average RBCVT results is considered as a base for comparison of all the test case generation methods. Fig. 15 presents a comparison among all the approaches against RT with respect to the point pattern. Again we can observe that the RBCVT method has the highest testing effectiveness. It is worth noting that in contrast with previous patterns, all

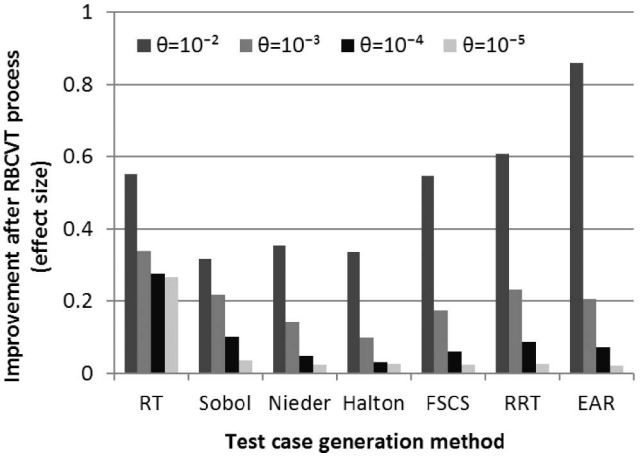

Fig. 14. Improvement of test case generation methods with respect to RBCVT process at different failure rates regarding the point failure pattern.

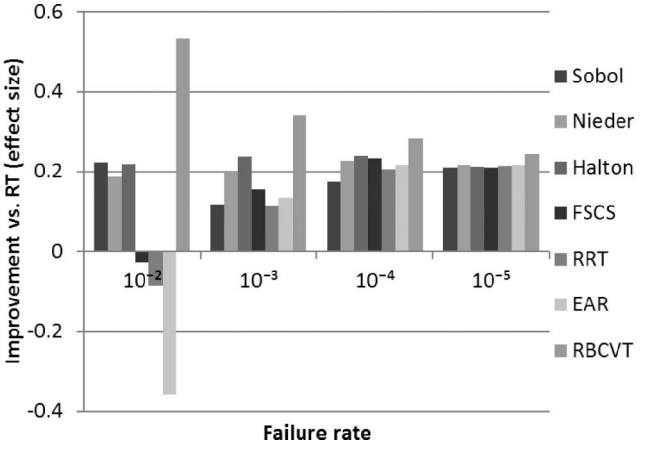

Fig. 15. P-measure testing effectiveness for point pattern simulations of FSCS, RRT, EAR, RBCVT, Sobol, Niederreiter, and Halton against the RT.

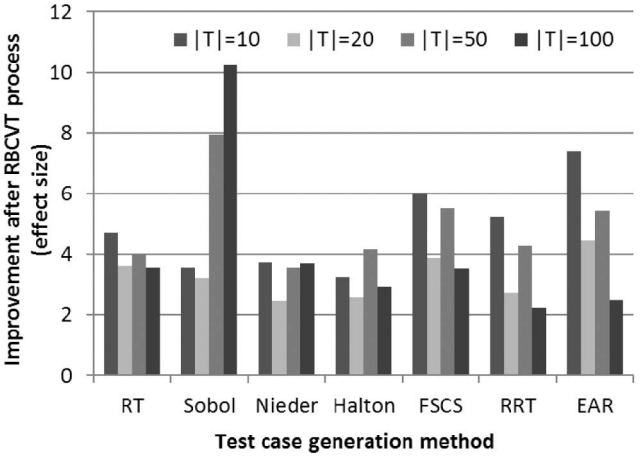

Fig. 16. Improvement of test case generation methods after the application of RBCVT with respect to the mutants' framework.

the ART approaches at $\theta = 10^{-2}$ have generated test cases with lower effectiveness than RT, while the QRT approaches have superior testing effectiveness compared to RT at all the studied failure rates.

7.5 Mutants' Testing Results

The testing effectiveness of all the studied approaches with respect to the real software testing framework based on mutation is represented in Table 5. The results demonstrate the significant improvement after the RBCVT approach is applied. One can observe that, in each case, the amount of improvement in terms of effect size is larger than the highest Cohen's description (Large). Further, the effect size is larger than two in all cases, leading to less than 18.9 percent overlap between the statistics of each method and its corresponding result after the application of RBCVT, according to Table 1.

Fig. 16 indicates the improvement of each approach after the RBCVT process in terms of effect size. In contrast with the simulation framework, no particular increasing/decreasing trend has been observed in this figure.

Similar to the simulation framework results, Fig. 17 provides a comparison among all of the approaches where

TABLE 5
The P-Measure Testing Effectiveness for All Approaches Including the Corresponding Results After the RBCVT Process with Respect to the Mutants' Framework

$\|T\|$	Method	Before the RBCVT process		After the RBCVT process		Effect size	Z-score	Significance
		Mean	std	Mean	std			
10	RT	0.0112	0.0021	0.0243	0.0033	4.7137	62.99	0.0000
	Sobol	0.0083	0.0046	0.0219	0.0029	3.5604	29.76	0.0000
	Niederreiter	0.0111	0.0018	0.0221	0.0038	3.7172	60.74	0.0000
	Halton	0.0115	0.0020	0.0223	0.0042	3.2508	52.99	0.0000
	FSCS	0.0130	0.0026	0.0326	0.0038	5.9945	75.83	0.0000
	RRT	0.0122	0.0020	0.0267	0.0034	5.2340	71.47	0.0000
	EAR	0.0338	0.0033	0.0632	0.0046	7.3876	89.35	0.0000
20	RT	0.0215	0.0029	0.0336	0.0037	3.6197	41.69	0.0000
	Sobol	0.0150	0.0065	0.0316	0.0033	3.2266	25.58	0.0000
	Niederreiter	0.0229	0.0028	0.0308	0.0036	2.4544	28.49	0.0000
	Halton	0.0223	0.0025	0.0303	0.0036	2.5746	31.92	0.0000
	FSCS	0.0244	0.0036	0.0404	0.0046	3.8819	44.91	0.0000
	RRT	0.0243	0.0032	0.0356	0.0049	2.7345	35.28	0.0000
	EAR	0.0488	0.0049	0.0720	0.0055	4.4511	47.25	0.0000
50	RT	0.0517	0.0049	0.0706	0.0045	4.0120	38.47	0.0000
	Sobol	0.0322	0.0053	0.0695	0.0040	7.9235	69.99	0.0000
	Niederreiter	0.0521	0.0035	0.0691	0.0058	3.5384	48.88	0.0000
	Halton	0.0515	0.0031	0.0690	0.0051	4.1634	57.34	0.0000
	FSCS	0.0562	0.0046	0.0845	0.0056	5.5187	61.28	0.0000
	RRT	0.0569	0.0045	0.0760	0.0044	4.2793	42.18	0.0000
	EAR	0.0817	0.0055	0.1122	0.0058	5.4239	55.54	0.0000
100	RT	0.0925	0.0050	0.1093	0.0043	3.5618	33.26	0.0000
	Sobol	0.0549	0.0055	0.1066	0.0046	10.2313	94.02	0.0000
	Niederreiter	0.0871	0.0057	0.1075	0.0053	3.6866	35.62	0.0000
	Halton	0.0917	0.0035	0.1070	0.0064	2.9412	43.26	0.0000
	FSCS	0.0988	0.0052	0.1182	0.0057	3.5278	36.96	0.0000
	RRT	0.1034	0.0052	0.1143	0.0044	2.2426	20.65	0.0000
	EAR	0.1259	0.0086	0.1445	0.0062	2.4885	21.78	0.0000

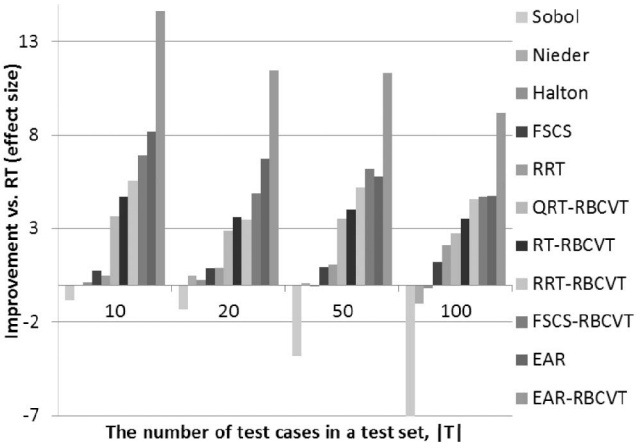

Fig. 17. P-measure testing effectiveness of each test case generation approach against RT with respect to the mutants' framework.

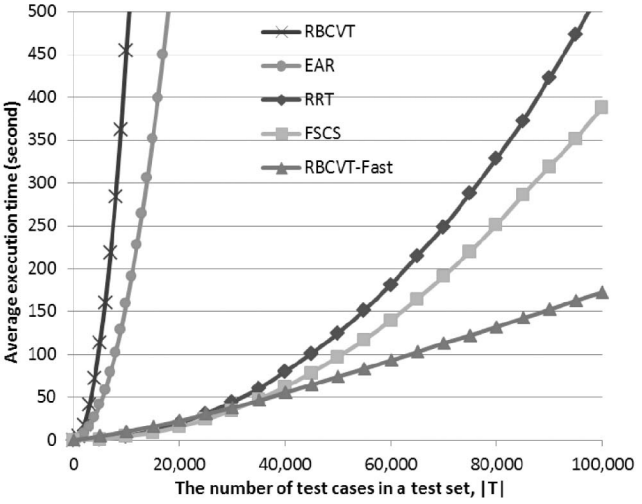

Fig. 18. Empirical test set generation runtime for RBCVT, RBCVT-Fast, FSCS, RRT, and EAR.

the RT effectiveness is considered as a reference, i.e., Fig. 17 represents the effect size of each strategy against RT. In contrast with the simulation framework, the P-measure results, after the application of RBCVT, are not similar in all cases. Only in the case of QRTs is a sizable reduction of the variation observed among RBCVT results. Accordingly, in Fig. 17 RBCVT results with QRTs as generators are combined as QRT-RBCVT, while RBCVT with other inputs is represented separately.

Test case generation approaches in Fig. 17 are sorted based on their performance where the EAR-RBCVT is the approach with highest efficiency and Sobol has the worst results in term of testing efficiency. Finally, as demonstrated in Fig. 17, QRT methods revealed degraded performance compared to RT in most of the cases, whereas other test case generation approaches outperformed RT.

7.6 Empirical Runtime Analysis

In addition to effectiveness, the computational complexity of an algorithm is an important factor in practical applications. In this research, different algorithms have been used as a basis to study the RBCVT method and in this section the runtime of these methods, as well as RBCVT, is investigated.

All the simulations within this study were conducted using Java (JDK 7, 64 bit). We implemented the RBCVT, FSCS, RRT, and EAR in Java and the Martingale stochastic library [65] has been used to generate the Sobol, Halton, and Niederreiter quasi-random sequences.[1] Besides, the Java native pseudorandom function has been employed for the RT test case generation. The hardware platform, where the simulation process has been executed, was an Intel dual-core Processor E6300 (2.8 GHz) with 8 GB of RAM.

To demonstrate the computational costs associated with each algorithm, an empirical runtime investigation has been performed. The parameters associated with each approach are the same as used during the evaluation, described in Section 6.2. Fig. 18 represents the test set generation runtime for the FSCS, RRT, EAR, RBCVT, and RBCVT-Fast in seconds. The runtime of the RT and QRT approaches has not been included in this figure due to their significantly lower runtime compared to the RBCVT and ART methods. The presented runtime values are the average runtime of $M_t = 100$ test set generation for each approach with each test set length ($0 < |T| \leq 100{,}000$). As indicated in this figure, the nonoptimized RBCVT has the largest runtime compared to all other methods and is within the order of quadratic time as calculated in Section 5.1.1. In accordance with runtime analysis in Section 5.2.3, RBCVT-Fast runtime is linear, based on empirical values observed in Fig. 18. Fig. 18 also demonstrates that RBCVT-Fast has the best runtime compared to the nonoptimized RBCVT and all the investigated ARTs, for $|T| \geq 30{,}000$. In addition, the computational complexity of 170 seconds for generating 100,000 test cases suggests 1.7 milliseconds for each test case in the proposed RBCVT-Fast calculation approach. It is worthwhile to note that similarly to ARTs, we can apply the mirroring technique [24] to RBCVT to reduce the execution times further if it is required.

8 Degree of Randomness Analysis

Besides the even distribution of the test cases within a test set, another important aspect of test case generation algorithms is their ability to generate a sequence of test cases which are random. Requiring random test cases has two different implications in this context:

- Randomness within a test set indicates the randomness among the individual test cases within a test set. A high degree of randomness in test cases is better since it provides the ability to generate uncorrelated test cases, which is essential for software testing applications. Uncorrelated test cases are critical to avoid systematic poor performance in certain situations (that is, a nonrandom set of test cases could significantly correlate with a current set of defects).
- Randomness between multiple test sets, which represents absence of correlation between two, or more, different sequences of test cases, resulting from different runs of the corresponding test case

1. In order to allow for the repeatability of the experiments contained within this paper, source code is available at www.steam.ualberta.ca/main/Papers/RBCVT.

generation algorithm. This is a critical feature of test case generation algorithm since software testing applications require uncorrelated sequences of test cases. Executing a sequence of test cases will hopefully result in the discovery of a number of defects. After correction, we may elect to execute another set of tests; ideally, the tester wants the option to execute either the previous set or a new set of test cases. Alternatively, if no or few defects were discovered, the tester will often want the option of executing another new and, by definition, different set of test cases in an attempt to discover more defects.

How we can measure randomness? Kolmogorov complexity provides a new class of distances appropriate for measuring similarity relations between sequences [39], [40]. The Kolmogorov complexity of a piece of information ($\delta(data)$) is the length of the ultimate lossless compressed version of the corresponding information [39]. In fact, the ultimate compressor does not exist. Thus, we have to use the lower bound of what a real-world compressor can achieve [39]. Within this study, the Lempel-Ziv-Markov chain Algorithm (LZMA) [66] is used to calculate $\delta(.)$ since it is believed that it is one of the best lossless compressors available. Before we can use a test set (T) as input to LZMA we need to preprocess the test set to convert it to a set of Integer values. Assuming a test set as

$$T = \{\{x_1, y_1\}, \{x_2, y_2\}, \ldots, \{x_{|T|}, y_{|T|}\}\}, \tag{19}$$

where $\{x_i, y_i\}$ denotes a 2D test case (t_i), the preprocessing function is defined as

$$T = \{x'_1, y'_1, x'_2, y'_2, \ldots, x'_{|T|}, y'_{|T|}\}, \tag{20}$$

where x'_i and y'_i denote the scaled integer representation of x_i and y_i, respectively.

Accordingly, to analyze within a test set randomness, $CR(T) = \frac{\delta(\varphi(T))}{|\varphi(T)|}$ is used, which indicates the compression ratio with respect to T. A compression ratio of one denotes a totally random test set, while fewer compression values denote repetitive patterns within the test set. Theoretically, $0 \leq CR(T) \leq 1$. However, since LZMA is not a perfect compressor, a small (unknown) additive offset exists in the estimation of $CR(T)$.

To investigate randomness between test sets, we used the NCD [39], indicating the similarity between two test sets. NCD is defined as [39]

$$NCD(T_i, T_j) = \frac{\delta(\varphi(T_{ij})) - min\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}}{max\{\delta(\varphi(T_i)), \delta(\varphi(T_j))\}}, \tag{21}$$

where T_{ij} is formed from the concatenation of T_i and T_j. When $NCD(T_i, T_j) = 0$, T_i and T_j are identical, whereas $NCD(T_i, T_j) = 1$ represents complete dissimilarity (these relationships assume perfect compression). The length of the test set should be large enough to be compressed effectively by LZMA. Thus, within this study the length of each test set is selected as an arbitrary large number, specifically as $|T| = 100{,}000$.

Table 6 represents the results of $CR(T)$ and $NCD(T_i, T_j)$ for RT, FSCS, RRT, and EAR approaches before and after the RBCVT process. QRT approaches have not been included since they use a deterministic algorithm producing a unique test set. The reported values in Table 6 are the average of 100 measurements, which indicates similar results before and after the RBCVT process regarding all studied approaches (in all situations, the variation between trials was negligible). These results suggest no degradation by RBCVT on the input points regarding randomness. In addition, all the ART methods perform similarly to RT with respect to degree of randomness.

TABLE 6
$CR(T)$ and $NCD(T_i, T_j)$ for RT, FSCS, RRT, and EAR Before and After the RBCVT Process

method	$CR(T)$		$NCD(T_i, T_j)$	
	Before RBCVT	After RBCVT	Before RBCVT	After RBCVT
RT	1.0134	1.0135	0.99953	0.99954
FSCS	1.0142	1.0143	0.99954	0.99953
RRT	1.0143	1.0144	0.99953	0.99956
EAR	1.0143	1.0142	0.99953	0.99955

9 Conclusion and Discussion

In this research, the novel RBCVT method has been proposed to the domain of software testing with the aim of increasing the effectiveness of test case generation approaches. The RBCVT method cannot be considered as an independent approach since it requires an initial set of input test cases. This method is developed as an add-on to the previous ART and QRT methods, enhancing the testing effectiveness by more evenly distributing test cases across the input space. In addition, the applied probabilistic approach for RBCVT generation allows different sets of output to be produced from the same set of inputs, which makes RBCVT an appropriate method for software testing applications.

The computational cost of a test case generation algorithm should be carefully considered in a practical application. In this research, we optimized the probabilistic computational algorithm of the RBCVT approach. The proposed search algorithm reduces the RBCVT computational complexity from a quadratic to a linear time order regarding the size of the test set, while ART methods still suffer from high runtime order. In this regard, the computational cost of RBCVT is quite feasible with respect to practical applications. It is worthwhile to state that since the RBCVT approach requires initial test cases, the computational cost of the input test set generation is added to the RBCVT calculation cost. Since the results provided in Tables 2, 3, 4, and 5 indicate, on average, "similar" results for RBCVT with different types of generators, we can select the RT method, which is linear and adds a low computational overhead onto the RBCVT execution. Therefore, with a concatenation of the RT and the RBCVT-Fast methods, we can produce a linear algorithm with respect to computational complexity, although in some specific situations this may lead to a slight reduction of algorithmic effectiveness. The principle contribution of this research is utilizing CVT to develop an innovative test cases generation approach, in

particular RT-RBCVT-Fast with linear order of computational complexity similar to RT.

Another important issue regarding the computational cost of test case generation (t_g) is its relation to the required time for the execution of that test case in a testing process (t_e). It is desirable to have $t_g < t_e$ since they can be parallel processes. When t_e is small (very small programs), one would be better off running more test cases instead of generating more efficient test cases, similar to [57]. In such a case, RBCVT or ART approaches are not cost effective due to their relatively high runtime. But industrial software's execution runtime is usually large enough to have adequate time for t_g. Furthermore, we view automatic test case generation for large volumes of test case generation as effectively an "offline" process. A program generates the test cases, a test harness executes the test cases, and the results are stored in a log file for the tester to consider either with or without the assistance of a test oracle system.

An extensive experimental study has been performed and the results demonstrate that RBCVT is significantly superior to all approaches for the block pattern in simulation framework at all failure rates as well as the studied mutants at all test set sizes. Although the magnitude of improvement in testing effectiveness results is higher for the block pattern compared to the point pattern, the results demonstrate statistically significant improvement in the point pattern. In contrast, ART methods have indicated less effectiveness than RT regarding point patterns at $\theta = 0.01$ (demonstrated in Fig. 15). Although RBCVT's performance regarding strip pattern is statistically significant compared to the other approaches at $\theta = 10^{-2}$, the impact of RBCVT versus the other approaches tends to zero as the failure rate decreases. In fact, in the case of strip pattern, the impacts of all of the approaches reduce to the performance of RT as the failure rate decreases; this is demonstrated in Fig. 13. In contrast, in block and point patterns, the performance of all the approaches verses RT usually stays constant or even increases as the failure rate reduces, Figs. 11 and 15. It is believed that these conclusions are stable regardless of the failure rate, and hence, simulating lower failure rates than studied in this research is not required. This fact is also verified in [67]. Randomness of test cases is an important factor with respect to software testing. Accordingly, the investigation of randomness in Section 8 demonstrates that RT, all ART methods, and all corresponding RBCVT methods possess an appropriate degree of randomness.

Although in real-life applications test cases' dimension can be large, in most cases they belong to an acceptable range. Test case generation often seeks to generate values with a specific purpose rather than generating test cases to exercise the entire system. The large size of the input space for modern software systems tends to imply that this "scatter gun" approach is ineffective. Instead, the tester will often have a specific testing objective and will attempt to generate a specific set of test cases under specific circumstances that answer this question. That is, the tester tends to test aspects of the system or subcomponents of the system rather than blindly "attacking" the entire system. As an example, in unit testing, the program under test is usually small, so the number of input and output variables is limited, as is the number of dimensions. For instance, Ciupa et al. [34] conducted an empirical study on several real-world small routines using unit testing. Briand and Arcuri [57] have considered 11 programs, basic mathematical functions that appear in the ART literature [20], for empirical analysis. The generated test cases in these papers do not exceed four dimensions. Furthermore, some techniques, like range coding [68], exist to reduce the dimension of the input space, especially when collections are considered as the input to the software under the test. As a result, where we do not have large dimensions, the linear RBCVT-Fast approach dominates over ART approaches regarding computational cost.

Although the results of this research improve software testing effectiveness, the RBCVT approach still has room for improvement since the maximum theoretical P-measure testing effectiveness is one. Therefore, one can clearly observe from the simulation study that more effective algorithms than the algorithms introduced within the literature can be developed in the future. Future research within this domain should focus on increasing software testing effectiveness and lowering computational complexity.

Finally, although further studies are required to validate the use of RBCVT in real-life applications, RT-RBCVT, ART-RBCVT, and QRT-RBCVT have been demonstrated to have superior performance against the RT, ART, and QRT methods, respectively. Consequently, software testing practitioners can use RBCVT to enhance the existing strategies within their software testing toolbox. The use of RBCVT in software testing is straightforward since RBCVT can be included in the previous methods as an add-on.

Acknowledgments

The authors would like to thank Dr. Arcuri and Dr. Briand for copies of the code used in their paper [57].

References

[1] V. Agrawal, "When to Use Random Testing," *IEEE Trans. Computers,* vol. 27, no. 11, pp. 1054-1055, Nov. 1978.
[2] J.W. Duran and S.C. Ntafos, "An Evaluation of Random Testing," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 438-444, July 1984.
[3] P. Loo and W. Tsai, "Random Testing Revisited," *Information and Software Technology,* vol. 30, no. 7, pp. 402-417, 1988.
[4] P. Schneck, "Comment on 'When to Use Random Testing'," *IEEE Trans. Computers,* vol. 28, no. 8, pp. 580-581, Aug. 1979.
[5] D.R. Slutz, "Massive Stochastic Testing of SQl," *Proc. 24th Int'l Conf. Very Large Data Bases,* pp. 618-622, 1998.
[6] H. Bati, L. Giakoumakis, S. Herbert, and A. Surna, "A Genetic Approach for Random Testing of Database Systems," *Proc. 33th Int'l Conf. Very Large Data Bases,* pp. 1243-1251, 2007.
[7] T. Yoshikawa, K. Shimura, and T. Ozawa, "Random Program Generator for Java Jit Compiler Test System," *Proc. Third Int'l Conf. Quality Software,* pp. 20-23, 2003.
[8] C. Pacheco, S.K. Lahiri, and T. Ball, "Finding Errors in .Net with Feedback-Directed Random Testing," *Proc. Int'l Symp. Software Testing and Analysis,* pp. 87-96, 2008.
[9] P. Godefroid, A. Kiezun, and M.Y. Levin, "Grammar-Based Whitebox Fuzzing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation,* pp. 206-215, 2008.
[10] J.E. Forrester and B.P. Miller, "An Empirical Study of the Robustness of Windows NT Applications Using Random Testing," *Proc. Fourth Conf. USENIX Windows Systems Symp.,* vol. 4, pp. 6-6, 2000.
[11] B.P. Miller, G. Cooksey, and F. Moore, "An Empirical Study of the Robustness of Macos Applications Using Random Testing," *Proc. First Int'l Workshop Random Testing,* pp. 46-54, 2006.

[12] S. Lipner and M. Howard, "The Trustworthy Computing Security Development Lifecycle Document (SDL)," http://msdn.microsoft.com/en-us/library/ms995349.aspx, 2005.

[13] P. Godefroid, "Random Testing for Security: Blackbox vs. Whitebox Fuzzing," *Proc. Second Int'l Workshop Random Testing: Co-Located with IEEE/ACM 22nd Int'l Conf. Automated Software Eng.*, p. 1, 2007.

[14] M. Marre and A. Bertolino, "Using Spanning Sets for Coverage Testing," *IEEE Trans. Software Eng.*, vol. 29, no. 11, pp. 974-984, Nov. 2003.

[15] A. Bertolino and M. Marre, "Automatic Generation of Path Covers Based on the Control Flow Analysis of Computer Programs," *IEEE Trans. Software Eng.*, vol. 20, no. 12, pp. 885-899, Dec. 1994.

[16] L. White and E. Cohen, "A Domain Strategy for Computer Program Testing," *IEEE Trans. Software Eng.*, vol. 6, no. 3, pp. 247-257, May 1980.

[17] P. Ammann and J. Knight, "Data Diversity: An Approach to Software Fault Tolerance," *IEEE Trans. Computers*, vol. 37, no. 4, pp. 418-425, Apr. 1988.

[18] G.B. Finelli, "NASA Software Failure Characterization Experiments," *Reliability Eng. and System Safety*, vol. 32, nos. 1/2, pp. 155-169, 1991.

[19] P. Bishop, "The Variation of Software Survival Time for Different Operational Input Profiles (or Why You Can Wait a Long Time for a Big Bug to Fail)," *23rd Int'l Symp. Fault-Tolerant Computing Digest of Papers*, pp. 98-107, june 1993.

[20] C. Schneckenburger and J. Mayer, "Towards the Determination of Typical Failure Patterns," *Proc. Fourth Int'l Workshop Software Quality Assurance: in Conjunction with the Sixth ESEC/FSE Joint Meeting*, pp. 90-93, 2007.

[21] 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.

[22] C. Jones, "Software Quality in 2011: A Survey of the State of the Art," http://www.asq509.org/ht/a/GetDocumentAction/id/62711, 2011.

[23] T. Chen, D.H. Huang, T. Tse, and Z. Yang, "An Innovative Approach to Tackling the Boundary Effect in Adaptive Random Testing," *Proc. 40th Ann. Hawaii Int'l Conf. System Sciences*, p. 262a, 2007.

[24] F.-C. Kuo, "An Indepth Study of Mirror Adaptive Random Testing," *Proc. Ninth Int'l Conf. Quality Software*, pp. 51-58, 2009.

[25] T. Chen, H. Leung, and I. Mak, "Adaptive Random Testing," *Proc. Ninth Asian Computing Science Conf.: Advances in Computer Science*, M. Maher, ed., pp. 3156-3157, 2005.

[26] T. Chen and R. Merkel, "Efficient and Effective Random Testing Using the Voronoi Diagram," *Proc. Australian Software Eng. Conf.*, pp. 300-308, 2006.

[27] T. Chen, R. Merkel, P. Wong, and G. Eddy, "Adaptive Random Testing through Dynamic Partitioning," *Proc. Fourth Int'l Conf. Quality Software*, pp. 79-86, 2004.

[28] T.Y. Chen, T.H. Tse, and Y.T. Yu, "Proportional Sampling Strategy: A Compendium and Some Insights," *J. Systems and Software*, vol. 58, no. 1, pp. 65-81, 2001.

[29] T. Chen and D. Huang, "Adaptive Random Testing by Localization," *Proc. 11th Asia-Pacific Software Eng. Conf.*, pp. 292-298, 2004.

[30] T. Chen, D. Huang, and Z. Zhou, "Adaptive Random Testing Through Iterative Partitioning," *Proc. 11th Ada-Europe Int'l Conf. Reliable Software Technologies*, L. Pinho and M. González Harbour, eds., pp. 155-166, 2006.

[31] J. Mayer, "Lattice-Based Adaptive Random Testing," *Proc. IEEE/ACM 20th Int'l Conf. Automated Software Eng.*, pp. 333-336, 2005.

[32] J. Mayer, "Adaptive Random Testing by Bisection and Localization," *Proc. Fifth Int'l Conf. Formal Approaches to Software Testing*, W. Grieskamp and C. Weise, eds., pp. 72-86, 2006.

[33] A. Tappenden and J. Miller, "A Novel Evolutionary Approach for Adaptive Random Testing," *IEEE Trans. Reliability*, vol. 58, no. 4, pp. 619-633, Dec. 2009.

[34] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer, "Artoo: Adaptive Random Testing for Object-Oriented Software," *Proc. ACM/IEEE 30th Int'l Conf. Software Eng.*, pp. 71-80, May 2008.

[35] T.Y. Chen and R. Merkel, "Quasi-Random Testing," *IEEE Trans. Reliability*, vol. 56, no. 3, pp. 562-568, Sept. 2007.

[36] H. Chi and E.L. Jones, "Computational Investigations of Quasirandom Sequences in Generating Test Cases for Specification-Based Tests," *Proc. 38th Conf. Winter Simulation*, pp. 975-980, 2006.

[37] K.P. Chan, T.Y. Chen, F.-C. Kuo, and D. Towey, "A Revisit of Adaptive Random Testing by Restriction," *Proc. 28th Ann. Int'l Computer Software and Applications Conf.*, vol. 1, pp. 78-85, 2004.

[38] P. LEcuyer and C. Lemieux, "Recent Advances in Randomized Quasi-Monte Carlo Methods," *Modeling Uncertainty*, pp. 419-474, Springer, 2005.

[39] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi, "The Similarity Metric," *IEEE Trans. Information Theory*, vol. 50, no. 12, pp. 3250-3264, Dec. 2004.

[40] M. Li and P. Vitanyi, *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, 2008.

[41] K. Chan, T. Chen, and D. Towey, "Restricted Random Testing," *Proc. Seventh Int'l Conf. Software Quality*, J. Kontio and R. Conradi, eds., pp. 321-330, 2006.

[42] T.Y. Chen, F.-C. Kuo, and H. Liu, "Adaptive Random Testing Based on Distribution Metrics," *J. Systems and Software*, vol. 82, no. 9, pp. 1419-1433, 2009.

[43] J. Mayer and C. Schneckenburger, "An Empirical Analysis and Comparison of Random Testing Techniques," *Proc. ACM/IEEE Int'l Symp. Empirical Software Eng.*, pp. 105-114, 2006.

[44] K.P. Chan, T. Chen, and D. Towey, "Forgetting Test Cases," *Proc. 30th Ann. Int'l Computer Software and Applications Conf.*, vol. 1, pp. 485-494, Sept. 2006.

[45] I.M. Sobol, "Uniformly Distributed Sequences with Additional Uniformity Properties," *J. Computational Math. and Math. Physics*, vol. 16, pp. 236-242, 1976.

[46] J.H. Halton, "Algorithm 247: Radical-Inverse Quasi-Random Point Sequence," *Comm. ACM*, vol. 7, pp. 701-702, Dec. 1964.

[47] H. Niederreiter, "Low-Discrepancy and Low-Dispersion Sequences," *J. Number Theory*, vol. 30, no. 1, pp. 51-70, 1988.

[48] H. Faure, "Discrépance de Suites Associées à un Systeme de numération (en Dimension Un)," *Bull. Soc. Math. France*, vol. 109, no. 2, pp. 143-182, 1981.

[49] P. Peart, "The Dispersion of the Hammersley Sequence in the Unit Square," *Monatshefte für Mathematik*, vol. 94, pp. 249-261, 1982.

[50] C. Schlier, "On Scrambled Halton Sequences," *Applied Numerical Math.*, vol. 58, no. 10, pp. 1467-1478, 2008.

[51] B.L. Fox, "Algorithm 647: Implementation and Relative Efficiency of Quasirandom Sequence Generators," *ACM Trans. Math. Software*, vol. 12, pp. 362-376, Dec. 1986.

[52] H. Everett, D. Lazard, S. Lazard, and M. Safey El Din, "The Voronoi Diagram of Three Lines," *Discrete and Computational Geometry*, vol. 42, pp. 94-130, 2009.

[53] Q. Du, V. Faber, and M. Gunzburger, "Centroidal Voronoi Tessellations: Applications and Algorithms," *SIAM Rev.*, vol. 41, no. 4, pp. 637-676, 1999.

[54] L. Ju, Q. Du, and M. Gunzburger, "Probabilistic Methods for Centroidal Voronoi Tessellations and Their Parallel Implementations," *Parallel Computing*, vol. 28, no. 10, pp. 1477-1500, 2002.

[55] A. Okabe, B. Boots, K. Sugihara, and S.N. Chiu, *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*, second ed. John Wiley & Sons, Inc., 2008.

[56] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-Tree: An Efficient and Robust Access Method for Points and Rectangles," *SIGMOD Record*, vol. 19, pp. 322-331, May 1990.

[57] A. Arcuri and L. Briand, "Adaptive Random Testing: An Illusion of Effectiveness?" *Proc. Int'l Symp. Software Testing and Analysis*, pp. 265-275, 2011.

[58] T.Y. Chen, F.-C. Kuo, and R. Merkel, "On the Statistical Properties of Testing Effectiveness Measures," *J. Systems and Software*, vol. 79, no. 5, pp. 591-601, 2006.

[59] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," *Proc. 29th Int'l Conf. Software Eng.*, pp. 75-84, May 2007.

[60] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation*, pp. 213-223, 2005.

[61] Y.-S. Ma, J. Offutt, and Y.R. Kwon, "Mujava: An Automated Class Mutation System," *Software Testing, Verification, and Reliability*, vol. 15, no. 2, pp. 97-133, 2005.

[62] J. Cohen, "A Power Primer," *Psychological Bull.*, vol. 112, no. 1, pp. 155-159, 1992.

[63] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences.* Lawrence Erlbaum, 1988.
[64] L.A. Becker, "Effect Size (ES)," no. 1993, http://web.uccs.edu/lbecker/Psy590/es.htm, 2000.
[65] M.J. Meyer, "Martingale Java Stochastic Library," http://martingale.berlios.de/Martingale.html, 2012.
[66] K. Morse Jr, "Compression Tools Compared," *Linux J.,* vol. 2005, no. 137, pp. 62-66, 2005.
[67] T. Chen, F. Kuo, and Z. Zhou, "On Favourable Conditions for Adaptive Random Testing," *Int'l J. Software Eng. and Knowledge Eng.,* vol. 17, no. 6, pp. 805-825, 2007.
[68] D. Salomon, *Data Compression: The Complete Reference,* vol. 10. Springer-Verlag, 2007.

Ali Shahbazi received the BSc and MSc degrees in electrical engineering from Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran, in 2007 and 2010, respectively. He is currently working toward the PhD degree in the Department of Electrical and Computer Engineering at the University of Alberta, Edmonton, Canada. He is a member of the STEAM Lab working on software testing. His research interests include software testing approaches in web-based systems, embedded systems, and mobile devices, as well as security of web-based systems. He is a student member of the IEEE.

Andrew F. Tappenden received the PhD degree in software engineering from the Department of Electrical and Computer Engineering at the University of Alberta. While there, he was a member of the STEAM group working on testing frameworks for pervasive computer devices. In recognition of his contributions to research, he was awarded the Andrew Stewart Memorial Graduate Prize from the University of Alberta as one of the top 25 graduate students of 2009. In 2010, he joined The King's University College as an assistant professor, and has continued his research in the development of verification methodologies for emerging technologies. His work is currently focused on the application of heuristics-based search algorithms and grammar-based testing in the automation of verification activities for service-oriented computing. He has authored several journal publications and conference articles covering a range of software engineering topics, including web engineering, software testing, agile development, machine learning, and the automation of software verification. He is a member of the IEEE.

James Miller received the BSc and PhD degrees in computer science from the University of Strathclyde, Scotland. During this period, he worked on the ESPRIT project GENEDIS on the production of a real-time stereovision system. Subsequently, he worked at the United Kingdom's National Electronic Research Initiative on Pattern Recognition as a principal scientist, before returning to the University of Strathclyde to accept a lectureship and, subsequently, a senior lectureship in computer science. Initially, during this period his research interests were in computer vision, and he was a co-investigator on the ESPRIT 2 project VIDIMUS. Since 1993, his research interests have been in web, software, and systems engineering. In 2000, he joined the Department of Electrical and Computer Engineering at the University of Alberta, Canada, as a full professor and in 2003 he became an adjunct professor in the Department of Electrical and Computer Engineering at the University of Calgary. He is the principal investigator in a number of research projects that investigate software verification, validation and evaluation issues across various domains, including embedded, web-based, and ubiquitous environments. He has published more than 100 refereed journal and conference papers on web, software, and systems engineering (see www.steam.ualberta.ca for details on recent directions). He recently served as the 2011 organizing chair for the IEEE International Symposium on Empirical Software Engineering and Measurement, and he sits on the editorial board of the *Journal of Empirical Software Engineering*. He is a member of the IEEE.