Quasi-Random Testing

Tsong Yueh Chen, Member, IEEE, and Robert Merkel

Abstract—Our paper proposes an implementable procedure for using the method of quasi-random sequences in software debug testing.

In random testing, the sequence of tests (if considered as points in an \(d\)-dimensional unit hypercube) will give rise to regions where there are clusters of points, as well as underpopulated regions. Quasi-random sequences, also known as low-discrepancy or low-dispersion sequences, are sequences of points in such a hypercube that are spread more evenly throughout. Based on the observation that program faults tend to lead to contiguous failure regions within a program’s input domain, and that an even spread of random tests enhances the failure detection effectiveness for certain failure patterns, we examine the use of quasi-random sequences as a replacement for random sequences in automated testing.

Because there are only a small number of quasi-random sequence generation algorithms, and each of them can only generate a small number of distinct sequences, the applicability of quasi-random sequences in testing real programs is severely restricted. To alleviate this problem, we examine the use of two types of randomized quasi-random sequences, which are quasi-random sequences permuted in a nondeterministic fashion in such a way as to retain their low discrepancy properties. We show that testing using randomized quasi-random sequences is often significantly more effective than random testing.

Index Terms—Adaptive random testing, automated testing, low-discrepancy sequence, low-dispersion sequence, quasi-random sequence, random testing.

ACRONYM

<table>
<thead>
<tr>
<th>ART</th>
<th>Adaptive Random Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRT</td>
<td>Restricted Random Testing</td>
</tr>
<tr>
<td>QRT</td>
<td>Quasi-Random Testing</td>
</tr>
<tr>
<td>AOR</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>SVR</td>
<td>Scalar Variable Replacement</td>
</tr>
<tr>
<td>CR</td>
<td>Constant Replacement</td>
</tr>
<tr>
<td>OTH</td>
<td>Other program modification</td>
</tr>
</tbody>
</table>

NOTATION

\(d\)  
Number of dimensions in unit cube
\(N\)  
Number of points in unit cube
\(x_i\)  
Point in unit cube

\(I_d\)  
d-dimensional unit cube
\(D(J; N)\)  
Local discrepancy of subinterval \(J\)
\(A(J)\)  
Number of points in subinterval \(J\)
\(V(J)\)  
Volume of \(J\)
\(\Delta(N)\)  
Star discrepancy of a point set
\(B_d\)  
Constant factor in discrepancy sequence formula
\(C_d\)  
Constant factor in discrepancy sequence formula
\(t, d\)-sequence  
A type of sequence having low discrepancy properties, where \(t\) is a constant, and \(d\) is the dimensionality of the sequence
\(b\)  
Base used in computation of quasi-random sequence
\(E\)  
Elementary interval in base \(b\)
\((t, m, d)\)-net  
A set of points making up well-defined parts of a \((t, d)\)-sequence
\(\gamma, \gamma_i\)  
Random real numbers between 0, and 1 used in Cranley-Patterson rotation
\(A_i^j\)  
The \(j th\) base-\(b\) digit in the \(i th\) element of a quasi-random sequence
\(X_i^j\)  
The \(j th\) base-\(b\) digit in the \(i th\) element of a quasi-random sequence after scrambling
\(x_i^{jk}\)  
A complete digit permutation used in Owen scrambling
\(T_i^j\)  
The \(j th\) element in the \(i th\) input used for testing
\(O(g(n))\)  
The well-known “big-O” notation. If, for some constants \(c\) and \(M\), for all \(n > c\), \(|f(n)| \leq M|g(n)|\), then \(f(n)\) is \(O(g(n))\)

I. INTRODUCTION

RANDOM testing is a basic program testing technique which has been extensively used for reliability estimation, and debug testing. Examples of its application to debug testing include Java Virtual Machines [1], and SQL database servers [2]. Its use is often advocated in the context of reliability assessment, where the profile of test inputs is often selected according to an operational profile intended to match program usage. However, it can also be used for debug testing, in which the goal is to expose as many program failures as possible with the testing resources available. However, its use in this context has been subject to considerable debate, with some authorities arguing that the large number of tests required to reveal failures makes it unsuited to this task [3].

Several independent studies have shown that program faults tend to create contiguous regions within a program’s input
domain in which testing will reveal failure. White & Cohen [4] provide an example of how contiguous failure regions can be formed by common program faults known as domain errors, where the conditions deciding the program’s execution path contain a fault. A simple example is when a constant in a path condition is changed, which can result in a contiguous “strip” where the wrong execution path is chosen, and thus an incorrect output will be computed. Bishop [5] describes the prevalence of “blob” defects in a study of software implementing a nuclear reactor control function. Bishop further observes that software can be viewed as a composition of functions. Consider what happens if a failure occurs in one such function (such as a domain error as described above). In this case, the region of failure is often continuous with respect to the actual input variables of that function. If the upstream, and downstream functions relative to the faulty function are continuous, the resultant failure region will be continuous with respect to the original input variables. All 30 defects examined in his study resulted in continuous “blob” regions of failure. It is not surprising that contiguous failure regions are so common. For a failure region to be non-contiguous, it must occur such that every input parameter must be at a particular value to trigger the failure! Obviously, this is not common.

Given that contiguous failure regions are common, it should also be common for non-failure regions to be contiguous. If a test $t$ is conducted, and does not reveal failure, a test spread away from $t$ is, in general, more likely to detect a failure than one close to it. Based on this intuition, a new approach called Adaptive Random Testing (ART) has been developed to improve the failure-detection effectiveness of random testing [6–8]. Several variants of this approach have been empirically evaluated. They use significantly less test cases to detect the first failure (a metric known as the F-measure), showing reductions of up to 50% compared with random testing. However, they have a significant drawback: as currently implemented, the time required to generate a sequence of $n$ test cases is proportional to $n^2$, making the methods unsuitable for generating very large sequences of test cases.

To reduce these costs, lower-overhead approaches have been proposed. One approach has been to use a less rigorous criterion for enforcing separation between test cases. For instance, Chen et al. [9], proposed Bisection-ART, in which the input domain is dynamically bisected, and test cases are generated in the resulting dynamic partitions, with recursive division until failure is detected.

In this paper, we propose an alternative approach. Rather than trying to filter or restrict randomly selected test cases to conform to the “separateness” requirements, we propose the use of a class of sequences that intrinsically possess such a property, namely quasi-random sequences. These sequences have been widely used in mathematics for a number of purposes, most notably in speeding up the numerical evaluation of multidimensional integrals. They take linear time, and negligible storage to generate.

We organize our paper as follows: in Section II, we introduce the definition of quasi-random sequences, and briefly outline some of the sequence generation methods used. Section III studies the effectiveness of quasi-random sequences in detecting simulated contiguous failure patterns. We introduce the technique of randomized quasi-random sequences in Section IV, which makes possible the generation of arbitrarily many different quasi-random sequences as is required in some practical testing situations, and especially for the empirical evaluation of the method using a small number of faulty programs. We perform such a study, showing that the failure-detection effectiveness of testing with quasi-random sequences is much better than random, and comparable to that of ART. Section V sums up our findings, and examines potential future studies.

II. QUASI-RANDOM SEQUENCES

In the mathematical literature, the terms “low-discrepancy sequence,” and “low-dispersion sequence” are used to specifically describe point sequences that have a low discrepancy, and low dispersion, respectively. In practice, sequences that possess one property tend to display the other. These low-discrepancy, and low-dispersion sequences exhibit some, but not all of the properties of random sequences, and hence are described as “quasi-random sequences”.

Intuitively, a low-discrepancy sequence is a sequence of points in a $d$-dimensional half-open unit cube with the property that in any given subinterval, points will be reasonably evenly distributed. For $N$ points $x_1, \ldots, x_n$ in the $d$-dimensional half-open unit cube $I^d = [0, 1)^d$, $d \geq 1$, and a subinterval $J$ of $I^d$, we define the local discrepancy as

$$D(J; N) = \left| \frac{A(J)}{N} - V(J) \right|$$

where $A(J)$ is the number of $x_i$ in $J$, and $V(J)$ is the volume of $J$ [10].

Fig. 1 shows a unit square (a 2-dimensional unit hypercube) containing six points. As an example, we select three sub-regions (indicated by the shaded areas), and consider their discrepancy. Region (a), the bottom half of the unit square, containing three of the six points, has zero discrepancy as half the area contains half of the points. Region (b) contains zero points, but has...
an area of $0.25 \times 0.2 = 0.05$, so the discrepancy would be 0.05. Region (c) contains $1/6 \approx 0.1666$ of the points, and has an area of 0.0001, so the discrepancy is nearly $1/6$; the region could be made arbitrarily small but still containing the point, and thus the discrepancy would asymptotically approach $1/6$.

The star discrepancy of the points $x_1, \ldots, x_n$ is defined as

$$
\Delta(N) = \sup_J |D(J; N)|
$$

where the supremum is extended over all half-open subintervals $J = \prod_{i=1}^{d} [0, n_i)$ of $I^d$.

Low-discrepancy sequences, as defined in [10], are sequences of points in $I^d$ where for all $N \geq 2$

$$
\Delta(N) \leq B_d (\log N)^{d-1} + O \left( (\log N)^{d-2} \right),
$$

and

$$
\Delta(N) \leq C_d (\log N)^{d} + O \left( (\log N)^{d-1} \right)
$$

where the constants $B_d$ and $C_d$ are as small as possible. It is not difficult to construct sequences so that, for some sequence lengths, the discrepancy is low. It is much more challenging, however, to construct sequences such that the discrepancy remains low for all $N$.

Several methods for constructing low-discrepancy sequences have been Proposed; Niederreiter [10] provides a good reference. All of these methods can be described in terms of what Niederreiter calls $(t, s)$-sequences; we will however use the term $(t, d)$-sequence, to avoid the use of $s$—which is used for a special purpose in this journal. First, he defines an elementary interval in base $b$ as an interval of the form

$$
E = \prod_{i=1}^{d} [a_i b^{-d_i}, (a_i + 1)b^{-d_i})
$$

where integers $d_i \geq 0$, and integers $0 \leq a_i < b^{d_i}$ for $1 \leq i \leq d$.

Intuitively, an elementary interval is a rectangular (or $n$-dimensional equivalent) sub-region precisely expressible in that base-$b$ number system (so for base 10, the elementary intervals would be those precisely expressible in decimal-point notation). Next, Niederreiter defines the concept of a $(t, m, s)$-net (again, we will use $(t, m, d)$-net) in base $b$. Let $0 \leq t \leq m$ be integers, and $d$ be the dimensionality of the desired point set. A $(t, m, d)$-net in base $b$ is a point set of $b^m$ points in $I^d$ such that $D(E; b^m) = b^{-t}$ for every elementary interval $E$ in base $b$ with $V(E) = b^{-m}$.

In other words, every elementary interval (aligning on the digit boundaries), which can take any proportions but must have a specified area, must contain the same number of points. So if $b = 2$, $t = 1$, $d = 2$, and $m = 2$, then the requisite net would have 4 points. For every elementary interval of size $2^{1-2} = 2^{-1} = 1/2$, there must be precisely $2^{1} = 2$ points in it. Note that the local discrepancy of any such elementary interval will therefore be zero.

An example $(1, 2, 2)$-net in base 2 is depicted in Fig. 2. The relevant elementary intervals are the four rectangles \{ACDE, DEFH, ABFG, BCGH\}. Of the four points in the net, in each elementary interval, there are precisely 2 points.

Let $t \geq 0$ be an integer. A (potentially infinite) sequence $x_1, x_2, \ldots$, in $I^d$ is called a $(t, d)$-sequence in base $b$ if, for all integers $k \geq 0$, and $m > t$, the point set \{x_{kb^m+1}, x_{kb^m+2}, \ldots, x_{kb^{m}+k+1} \}_{y=1} \in b$ is a $(t, m, d)$-net in base $b$. Consider the following example: If $t = 1$, $d = 2$, $b = 2$, any sequence meeting the definition would require the point set \{x_1, x_2, x_3, x_4\} to be a $(1, 2)$-net, as would \{x_5, x_6, x_7, x_8\}, \{x_9, \ldots, x_{12}\}, and so on. \{x_1, x_2, \ldots, x_8\}, \{x_9, \ldots, x_{16}\}, and so on would have to form $(1, 3, 2)$-nets. Successively larger groups would have to form nets with a bigger value for $m$.

The discrepancy of $(t, d)$-sequences is lower than any other sequence yet known, and is bounded according to the formulae in Niederreiter ([10], p. 53). Niederreiter also provides a method for efficiently generating a $(t, d)$-sequence. A complete method for any prime base is provided by Bratley et al. [11], who further show that base-$2$ sequences can be generated much more quickly (using the fact that base-2 arithmetic is the native form used by computers). They conducted an empirical study of quasi-random sequences for quasi-Monte Carlo integration, showing that the base-$2$ sequences perform more effectively than other bases for this purpose.

While the theory behind Niederreiter’s method is very complex, its implementation is reasonably simple, and efficient. Generating the next $N$-dimensional point in the sequence uses constant time, and negligible storage. An implementation of Niederreiter’s method for base 2 sequences is available as part of the GNU Scientific Library [12]. The implementation in this library is a direct C translation of the original FORTRAN code written by Bratley et al. [11].

III. SIMULATION STUDY

An experiment was conducted to determine whether tests specified by a quasi-random sequence could detect failures more effectively than randomly selected tests.

Both simulation, and empirical studies have been used to assess the effectiveness of new testing methods. Empirical studies, where the effectiveness of methods in detecting faults in real programs is examined, have clear, obvious advantages over simulation studies. However, the nature of the quasi-random sequence generator poses a challenge to the design of appropriate empirical studies.
In previous empirical studies [6], a mean F-measure for a testing method, for a particular program under test, has been estimated by repeatedly running the method, using different sequences of test cases based on different random seeds for each run. This will not work for testing based on quasi-random sequences. Bratley, Fox, and Niederreiter’s implementation [11] can only generate one unique sequence for a particular base, and given that only the base-2 generator was available, only one unique sequence could be generated.

Therefore, as a preliminary investigation, the effectiveness of quasi-random sequences was measured by using a simulated failure pattern that could be placed in random locations over multiple runs. While this is undoubtedly less realistic than using real program failures, it at least lets some statistical analysis be done, and so we could assess whether the method has any potential, thus justifying further more comprehensive investigation.

A. Method

For each experimental trial, a simulated program input domain of a unit square was created. A square “failure region” of a specified size was randomly placed inside the input domain. The implementation of the Niederreiter sequence in the GNU Scientific Library [12] was used to generate “tests” within the input domain, until a point was generated within the simulated failure region, simulating the detection of the “failure”.

To estimate the effectiveness of testing, the number of “tests” required to detect the failure region was recorded. We use the F-measure rather than alternatives, such as the P-measure (probability of at least one failure being detected), or E-measure (expected number of failures in a test set), for two reasons. Firstly, it avoids having to subjectively choose the number of tests in a test set, as is required for the E-measure, and P-measure. Secondly, the E-measure is not appropriate for use with adaptive testing methods that use the results of previously executed test cases. Details can be found in [13]. For these reasons, we have used the F-measure on our previous studies of adaptive random testing, and therefore we continue with the practice here.

Four different experimental conditions were tried, with the only varying parameter being the size of the simulated failure regions. They were of size 0.01, 0.002, 0.001, and 0.0002. For random testing with replacement, the expected mean F-measure for those experimental conditions was 100, 500, 1000, and 5000 respectively.

B. Results

The results of our simulation are presented in Table I. The F-measure is reduced by between 21% to 33% when compared to random testing.

It is clear from these results that testing using quasi-random sequences offers the possibility of considerable improvements in effectiveness over random testing. However, without an ability to somehow “randomize” the sequences, there is no way to evaluate the effectiveness of the methods empirically with a small number of test programs. Moreover, for practical testing, the availability of a large number of sequences to test with is very useful.

<table>
<thead>
<tr>
<th>Failure Pattern Size</th>
<th>F-Measure for Random Testing (F_{rt})</th>
<th>F-Measure for Quasi-Random Testing (F_{qrt})</th>
<th>F_{qrt}/F_{rt} (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>100</td>
<td>67.0</td>
<td>67.0%</td>
</tr>
<tr>
<td>0.002</td>
<td>500</td>
<td>358.0</td>
<td>71.6%</td>
</tr>
<tr>
<td>0.001</td>
<td>1000</td>
<td>788.8</td>
<td>78.9%</td>
</tr>
<tr>
<td>0.0002</td>
<td>5000</td>
<td>3847</td>
<td>76.9%</td>
</tr>
</tbody>
</table>

IV. Randomized Low Discrepancy Sequences

Niederreiter sequences have a significant limitation for our purposes; as discussed in the previous section, only a small number of different sequences can be generated using these methods. This is somewhat problematic when applying the sequences to software testing; it poses an even greater problem for empirically assessing their effectiveness in the context of testing. Generally, past empirical studies have applied testing methods repeatedly to programs seeded with the same errors (and thus exhibiting the same failure behavior), with a different random number generator seed, and then calculating average performance.

This restriction is also of practical importance in numerical multidimensional integration, which is the major current application of quasi-random sequences. While they are often more accurate than methods using random sequences, it is difficult to assess the accuracy of the results obtained (for example, see [14]). The notion of "scrambling" was therefore proposed to solve this problem [14]. In this section, we investigate two scrambling techniques, and evaluate their effectiveness in the context of software testing.

A. Cranley-Patterson Rotation

One of the simplest methods for scrambling quasi-random sequences is Cranley-Patterson rotation [15]. Kollig & Keller [16] provide a clearer description of the method: to rotate a d-dimensional sequence, randomly generate $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_d)$, where $\forall i, 0 \leq \gamma_i < 1$. For every point $p_i$ in the quasi-random sequence, the corresponding rotated point $p_{\gamma i}$ is defined as

$$ p_{\gamma i} = (p_{i1} + \gamma_1 \mod 1, p_{i2} + \gamma_2 \mod 1, \ldots, p_{id} + \gamma_d \mod 1). $$

Kollig & Keller [16] note that applying a Cranley-Patterson rotation to a $(t, m, d)$-net is not guaranteed to preserve its low discrepancy. Given that the intervals between the majority of points will not change, it is reasonable to expect that the discrepancy will not be substantially affected most of the time.

A further advantage is that the method is extremely fast, simple, and easy to implement.

B. Owen’s Scrambling Method

Owen [14], proposed a more elaborate scrambling method where the resulting scrambled net remains a $(t, m, d)$-net, thus avoiding the theoretical issues with Cranley-Patterson rotation.
Owen [14] defines randomized low-discrepancy sequences as follows: Let \( (A_k) \) be a \((t,m,d)\)-net, or a \((t,d)\)-sequence in base \( b \). The \( i \)th term in the sequence is written

\[
A_i^j = \sum_{k=1}^{\infty} a_{ijk} b^{-k}
\]

(6)

where \( 0 \leq a_{ijk} < b \) for all \( i, j, k \).

A randomized version of \((A_k)\) is a sequence \((X_i)\) where elements \(X_i = (X_i^1, \ldots, X_i^d)\) are defined as

\[
X_i^j = \sum_{k=1}^{\infty} x_{ijk} b^{-k}
\]

(7)

with \( x_{ijk} \) defined in terms of random permutations of the \( a_{ijk} \)

\[
x_{ij1} = \pi_j(a_{ij1}) \\
x_{ij2} = \pi_{j,1}(a_{ij2}) \\
x_{ij3} = \pi_{j,1,2}(a_{ij3}) \\
\vdots \\
x_{ijk} = \pi_{j,1,\ldots,i-1}(a_{ijk})
\]

Each permutation \( \pi \) is uniformly distributed over the \( b! \) permutations of \( \{0, 1, \ldots, b-1\} \), and the permutations are mutually independent. \( \pi_j \) permutes the first digit in the base \( b \) expansion of \( A_i^j \) for all \( j \). The second digit is permuted by \( \pi_{j,1} \), so the permutation applied to the second digit depends on the value of the first digit. The permutation applied to the \( k \)th digit depends on the values of the preceding \( k-1 \) digits. Owen shows that, if a \((t,d)\)-sequence is randomized using the procedure described above, then the resultant randomized sequence is also a \((t,d)\)-sequence.

C. Implementation

While implementation of Owen’s method appears straightforward, the number of permutations required by a naive implementation makes the storage requirements extremely high; if all 52 bits of precision in the fractional component of a double-precision floating point were used, \( 2^{52} \) permutations would need to be calculated, and stored! This is clearly infeasible.

In practice, the number of digits in the base \( b \) representation is restricted to a limited number. In base 2, a number of other performance-enhancing strategies can be employed. Notably, there are only two permutations in base 2; and thus a permutation can be represented as a single bit, and applied using a bitwise exclusive-or.

A number of implementations of this scrambling are freely downloadable for non-commercial use [17]–[19]. SamplePack [18] was chosen for our experiments, as it was readily available, and compatible with our existing work. Cranley-Patterson rotation is not directly available from this package, but was implemented easily using SamplePack’s unscrambled generator.

D. Empirical Study

We conducted an empirical study to examine the effectiveness of testing using “scrambled” low-discrepancy sequences. Both Cranley-Patterson rotation, and Owen scrambling were studied. Their effectiveness was compared to ART, and random testing.

We follow the basic experimental procedure of Chen et al. [6]. In their study of ART [6], they used as a study basis twelve small numerical programs drawn from the “Numerical Recipes” [20], and the “Collected Algorithms of the ACM” [21]. These programs each take a fixed length vector as input (varying in length between 1, and 4 for each program), with each vector element within specified bounds; and each program outputs a number. The original programs (most originally written in FORTRAN) were translated line by line into C++ for convenience. As the programs were numerical routines that used scalar data structures, the translated versions are all but identical to the originals, differing only in the details of syntax, and produce the same output. These programs were seeded with a number of errors. The errors were of a number of different types, including arithmetic operator replacement (AOR), relational operator replacement (ROR), scalar variable replacement (SVR), constant replacement (CR), and other (OTH). As a test oracle, the results from running the error-seeded program were compared with that of the unaltered program, with an error reported if the results differ. We use the same test programs so as to be able to directly compare the results with our previous experiments (see Table II). Details of the program input domains, types of errors, and failure rate can be found in [6].

In an experimental run, the randomized low-discrepancy generator that generates vectors of the appropriate dimensionality is initialized using a random number generator as a source. As elements \(X_i = (X_i^1, \ldots, X_i^d)\) in the low-discrepancy sequence are points in the half-open unit cube in \( d \) dimensions, they are transformed into test points \( T_i = (T_i^1, \ldots, T_i^d)\) using the mapping \( T_i^k = X_i^k \times range_{ek} + \text{m}_{in}_{ek} \), where \( \text{m}_{in}_{ek} \) is the lower permitted bound for the \( k \)th dimension of the input vector, and \( range_{ek} \) is the corresponding permitted range. The program under test is then executed with the test point until a failure is detected. The number of tests required to detect a failure (F-measure) is then recorded. 5000 runs were conducted for each program under test.

In terms of performance, the time taken to generate a randomized quasi-random sequence, using Owen scrambling or Cranley-Patterson rotation, of the lengths required was negligible on a 1.3GHz Pentium-M laptop.

E. Results

Table III shows the F-measure for the 12 test programs for Quasi-random testing using Owen scrambling, Quasi-random using Cranley-Patterson rotation, ART, and random testing. Note that the ART, and random testing data are taken from Chen et al. [6].

For 9 out of the 12 test programs, the average F-measure using quasi-random sequences with Owen scrambling was clearly much lower than that previously measured for random testing. However, the improvement was smaller than that reported for ART. In the remaining three cases, the programs
CHEN AND MERKEL: QUASI-RANDOM TESTING

TABLE II

DETAILS OF PROGRAMS USED IN EMPIRICAL STUDY (NOTE: D=DIMENSIONALITY OF PROGRAM INPUT DOMAIN)

<table>
<thead>
<tr>
<th>Prog Name</th>
<th>Input Domain</th>
<th>Error Type</th>
<th>Total Errors</th>
<th>Failure Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIRY</td>
<td>(-5000.0)</td>
<td>(5000.0)</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>BESSJ</td>
<td>(2.0, -1000.0)</td>
<td>(3000.0, 15000.0)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>BESSJ0</td>
<td>(-300000.0)</td>
<td>(300000.0)</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>CEL</td>
<td>(0.001, 0.001, 0.001)</td>
<td>(1.0, 300.0, 100000.0, 10000.0)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EL2</td>
<td>(0.0, 0.0, 0.0, 0.0)</td>
<td>(250.0, 250.0, 250.0, 250.0)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>ERFCC</td>
<td>(-300000.0)</td>
<td>(300000.0)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>GAMMQ</td>
<td>(0.0, 0.0)</td>
<td>(1700.0, 40.0)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>GOLDEN</td>
<td>(-100.0, -100.0, -100.0)</td>
<td>(60.0, 60.0, 60.0)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>PLGNDR</td>
<td>(10.0, 0.0, 0.0, 0.0)</td>
<td>(500.0, 11.0, 1.0)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>PROBKS</td>
<td>(-50000.0)</td>
<td>(50000.0)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SNCNDN</td>
<td>(-50000.0, -50000.0)</td>
<td>(5000.0, 5000.0)</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>TANH</td>
<td>(-50000.0)</td>
<td>(50000.0)</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

TABLE III

MEAN F-MEASURE AND CONFIDENCE INTERVALS (CI) FOR SCRAMBLED QUASI-RANDOM, ART, AND RANDOM TESTING

<table>
<thead>
<tr>
<th>Program</th>
<th>$F_{o_{wen}}$</th>
<th>95% CI ($F_{o_{wen}}$)</th>
<th>$F_{C-P}$</th>
<th>95% CI ($F_{C-P}$)</th>
<th>$F_{art}$</th>
<th>95% CI ($F_{art}$)</th>
<th>$F_r$</th>
<th>95% CI ($F_r$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIRY</td>
<td>799</td>
<td>(784, 814)</td>
<td>806</td>
<td>(791, 821)</td>
<td>799</td>
<td>(779, 819)</td>
<td>1381</td>
<td>(1333, 1430)</td>
</tr>
<tr>
<td>BESSJ</td>
<td>603</td>
<td>(588, 617)</td>
<td>604</td>
<td>(590, 619)</td>
<td>467</td>
<td>(452, 481)</td>
<td>802</td>
<td>(723, 831)</td>
</tr>
<tr>
<td>BESSJ0</td>
<td>522</td>
<td>(511, 533)</td>
<td>423</td>
<td>(415, 432)</td>
<td>424</td>
<td>(413, 435)</td>
<td>734</td>
<td>(707, 760)</td>
</tr>
<tr>
<td>CEL</td>
<td>1697</td>
<td>(1666, 1728)</td>
<td>1699</td>
<td>(1668, 1731)</td>
<td>1608</td>
<td>(1552, 1663)</td>
<td>3065</td>
<td>(2955, 3175)</td>
</tr>
<tr>
<td>EL2</td>
<td>1019</td>
<td>(996, 1041)</td>
<td>1062</td>
<td>(1039, 1086)</td>
<td>686</td>
<td>(662, 711)</td>
<td>1431</td>
<td>(1378, 1484)</td>
</tr>
<tr>
<td>ERFCC</td>
<td>1092</td>
<td>(1070, 1113)</td>
<td>967</td>
<td>(951, 984)</td>
<td>1005</td>
<td>(980, 1029)</td>
<td>1804</td>
<td>(1739, 1868)</td>
</tr>
<tr>
<td>GAMMQ</td>
<td>1138</td>
<td>(1106, 1169)</td>
<td>1201</td>
<td>(1170, 1232)</td>
<td>1081</td>
<td>(1044, 1119)</td>
<td>1220</td>
<td>(1176, 1264)</td>
</tr>
<tr>
<td>GOLDEN</td>
<td>1832</td>
<td>(1781, 1883)</td>
<td>3899</td>
<td>(3785, 4015)</td>
<td>1830</td>
<td>(1766, 1894)</td>
<td>1861</td>
<td>(1794, 1928)</td>
</tr>
<tr>
<td>PLGNDR</td>
<td>2082</td>
<td>(2033, 2130)</td>
<td>2148</td>
<td>(2097, 2198)</td>
<td>1807</td>
<td>(1754, 1859)</td>
<td>2742</td>
<td>(2647, 2837)</td>
</tr>
<tr>
<td>PROBKS</td>
<td>1801</td>
<td>(1765, 1840)</td>
<td>1474</td>
<td>(1445, 1502)</td>
<td>1443</td>
<td>(1407, 1478)</td>
<td>2635</td>
<td>(2542, 2727)</td>
</tr>
<tr>
<td>SNCNDN</td>
<td>636</td>
<td>(618, 653)</td>
<td>625</td>
<td>(608, 642)</td>
<td>629</td>
<td>(606, 651)</td>
<td>636</td>
<td>(613, 660)</td>
</tr>
<tr>
<td>TANH</td>
<td>379</td>
<td>(372, 387)</td>
<td>293</td>
<td>(288, 299)</td>
<td>307</td>
<td>(299, 315)</td>
<td>558</td>
<td>(538, 577)</td>
</tr>
</tbody>
</table>

“GAMMQ,” “GOLDEN,” and “SNCNDN,” the improvement is insignificant, which was also the case for ART.

V. DISCUSSION

We have shown that low-discrepancy sequences are a promising method for the automatic generation of test sequences for debug testing. We have also shown that the use of “scrambling” allows the generation of many different sequences, enabling us to obtain useful estimates of the improvement over random testing on real programs, and allows different sequences to be tried, which is an important point for industrial use.

As a pilot study, only one of the low-discrepancy sequences available was tried, with two scrambling methods. While it would be surprising if different low-discrepancy sequences produced significantly different overall results, it would be straightforward, and worthwhile to check.

Despite the fact that it does not have the same theoretical soundness as Owen scrambling, Cranley-Patterson rotation seems to work reasonably well in practice for testing purposes. In fact, it appears to work significantly better than Owen scrambling for one-dimensional input domains, so in this case our study supports the use of this simpler method over the more complex method; however, we believe that the one-dimensional case is not common in testing practice. Given the applications for which the scrambled low-discrepancy sequences were designed, it would not be surprising if Owen scrambling resulted in a less than optimal sequence in this degenerate case. It
would be interesting to investigate further whether the poor performance of Cranley-Patterson rotation on the “GOLDEN” program was an isolated case.

The implementation of Owen scrambling used for this pilot study has significant limitations in practice, as it requires the number of points in the sequence to be specified in advance; and then allocates, and seeds the scrambler (with the permutations) based on the size. Though we can estimate an upper bound of the number of tests required according to the testing resources available, it would be of significant benefit to develop an incremental sequence generator that allocates, and generates permutations as needed. The details of such an implementation will take some effort. Alternatively, there are a number of other scrambling techniques, for instance “digit scrambling” [16]. Digit scrambling is a simplified version of Owen scrambling, where instead of using different permutations depending on previous digits, one permutation is used all the time. Experiments with such techniques, as well as a number of other scrambling methods, would seem to be an obvious next step.

Testing using quasi-random sequences seems particularly well-suited to cases where a very large number of tests are required, as the cost of generating \( n \) tests is \( O(n) \). This is in contrast to the \( O(n^2) \) costs of most previous implementations of ART. Random testing is often used to generate very large test case sequences, so a linear-time method of doing so with greater effectiveness is potentially very useful. The improvement in failure detection effectiveness is not as substantial as in ART, so it does not make that method obsolete.

It should be noted that our proposed approach is only directly applicable to programs whose input space has a one-to-one correspondence with the unit \( n \)-dimensional hypercube. We are currently looking into ways to extend the method to a broader range of programs through appropriate mapping functions. With ART, considerable attention needs to be paid to choosing appropriate metrics for “even spread,” so that the improved effectiveness over random testing outweighs the selection overhead. With quasi-random testing, the overhead is essentially no higher than random testing, so a simple mapping from the non-numeric input domain to the unit hypercube may still offer improvements over random testing.

We conjecture that the points generated by ART would, in themselves, form a sequence with considerably lower discrepancy than random points. In fact, it is plausible that empirically-measured discrepancy may prove a strong predictor of failure-detection effectiveness for contiguous failure patterns. Empirical investigation of this connection may prove useful in mathematical modeling of testing strategies, and their effectiveness.

Because there is a rich mathematical literature on low-discrepancy sequences, we believe that there are considerable opportunities for researchers to explore this literature for methods useful in testing.

REFERENCES


Tsong Yuen Chen received his BSc, and MPhil degrees from the University of Hong Kong; MSc degree, and DIC from the Imperial College of Science and Technology, and PhD degree from the University of Melbourne. He is currently a Professor of Software Engineering in the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. His research interests include software testing, debugging, software maintenance, and software design.

Robert Merkel received the BSc (Hons) from the University of Melbourne, and his PhD from Swinburne University of Technology. He is currently a post-doctoral research fellow at Swinburne in the Faculty of Information and Communication Technologies. His current main research interests include software testing, and program verification. Prior to his PhD study, he worked in industry on an open source software project.