Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2019) 27: 2666 – 2680

doi:10.3906/elk-1801-18

Path-oriented random testing through iterative partitioning (IP-PRT)

Esmaeel NIKRAVAN*, **Saeed PARSA**
School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

Received: 02.01.2018 • **Accepted/Published Online:** 11.04.2018 • **Final Version:** 26.07.2019

Abstract: Path-oriented random testing aims at generating a uniformly distributed sequence of test data from a program input domain space to traverse a desired execution path of the program. To this aim, this article proposes a new algorithm to refine a program inputs domain space from invalid subdomains not covering the path. The validity of the subdomains is checked by a constraint propagation method against the path constraints (PCs). The proposed algorithm uses a divide-and-conquer technique to iteratively split the inputs domain into subdomains and each time refutes those subdomains that are inconsistent with the PCs. The remaining shrunken subdomains provide all possible test data covering the desired path. Obviously, the more accurate the input domain is, the more effective test data will result. Experiments show the proposed method outperformed other related methods on a set of classical benchmark programs.

Key words: Test data generation, random testing, path-oriented random testing, constraint propagation

1. Introduction

Automatic test data generation is concerned with the detection of program input data satisfying a given testing criterion. Currently, path coverage is the most applicable criterion for white-box testing [1,2]. To satisfy this criterion, test data should be generated in such a way that each path could be executed at least once. However, every practitioner knows that sometimes to detect a latent fault it will be required to execute the faulty path several times with different test data before the fault could be detected. Hence, selecting test cases that effectively detect faults at a minimum cost is an imperative task. To this aim, those values that do not cover the faulty path should be removed from the domain of input variables.

Among varying approaches to automatic test data generation, random testing (RT) is a simple and common method [3,4]. RT methods randomly select test data from a program input domain with a uniform probability distribution. Here, uniform probability distribution means that all the points in the input domain space have the same probability of being selected. Despite its simplicity and cost-effectiveness, many researchers believe that RT is not an efficient method in terms of coverage capability [5], while many studies showed that RT is an effective method in detecting faults not found by other methods [6].

The key advantage of RT generators over other methods is that it does not need any information about the program under test (PUT) and test data are generated without considering the specification or the structure of the PUT [7,8]. RT is applied by an approach known as path-oriented random testing (PRT) [9] to generate test data to cover a given execution path of the PUT. PRT initially uses backward symbolic execution [10] to derive the path constraints (PCs) and then divides the domain of all the input variables within the derived PCs into

*Correspondence: nikravan@comp.iust.ac.ir

k equally sized subdomains. Thereafter, the validity of each of the k subdomains against the PCs is checked by applying two known techniques of constraint propagation and constraint refutation. The constraint propagation algorithm is applied by PRT in order to examine values within each of the k subdomains to ensure that there is at least one value consistent with the PC. The difficulty is that the constraint propagation algorithm cannot detect subdomains with no values inconsistent with the PC. PRT removes all those subdomains not covering the PC. However, when selecting points from within the remaining subdomains, not every point necessarily satisfies the PC. Therefore, a major challenge with PRT is to select the appropriate value for the dividing parameter k. Obviously, the larger the value of k, the larger the number of subdomains to be evaluated against the PC. On the other hand, if k is small then the resultant subdomains could include many values inconsistent with the PC.

To address these problems and overcome the shortcomings of the PRT methods while retaining their advantages, this paper suggests a new PRT method, namely path-oriented random testing through iterative partitioning (IP-PRT). IP-PRT carries on with subdividing the subdomains, including at least one value consistent with the PC, until the number of invalid points in the program input domain space is reduced to a reasonable number. To keep track of the remaining valid subdomains, a Boolean hypercube in which each edge corresponds to an input variable domain is used. A random test data generator is invoked to select test data from the resulting subdomains.

The remaining parts of this paper are organized as follows. In Section 2, a motivating example is presented. Section 3 introduces a few basic terminologies, the RT method, and its improvements. In Section 4 our suggested test data generation method, IP-PRT, is elaborated. The outperformance of IP-PRT compared to others is demonstrated in Section 5, followed by threats to validity in Section 6. Finally, the conclusions of our research are presented in Section 7.

2. Motivating example

Consider the problem of generating test data to satisfy a given execution PC, $x \times y \leq 4$, where the input variables x and y are restricted to the interval 0…15. The input domain space of the problem is illustrated in Figure 1. As shown in Figure 1, the input domain space is a $\{0, \ldots, 15\} \times \{0, \ldots, 15\}$ plain in which the gray region includes all valid points (x_i, y_i) satisfying the PC, $x \times y \leq 4$. The gray region contains 39 points, covering about 15% of the whole input domain space. Therefore, when using a uniformly distributed random test data selection scheme, the probability of getting an invalid input value not satisfying the PC will be 85%.

This probability may be reduced by minimizing the number of invalid values not satisfying the PC in the input domain space. To this aim, as suggested in this paper, the following steps could be taken.

1. Each dimension of the input space is subdivided into k equally sized subregions, where k is a power of 2.
2. The validity of each subregion is examined against the PC by a constraint propagation algorithm.
3. All those subregions that contain at least one point satisfying the PC are kept and the remaining ones, which do not include any valid points, are refuted.
4. Repeat Step 1 until the number of iterations exceeds a threshold value.

For instance, as shown in Figure 2a, with dividing each dimension of the initial input domain space in Figure 1 into two distinctive subdomains 0…7 and 8…15, we get the following 4 subdomains: $D_1 = (x \in 0 \ldots 7, y \in 8 \ldots 15)$, $D_2 = (x \in 0 \ldots 7, y \in 0 \ldots 7)$, $D_3 = (x \in 8 \ldots 15, y \in 8 \ldots 15)$, and $D_4 = (x \in 8 \ldots 15, y \in 0 \ldots 7)$.

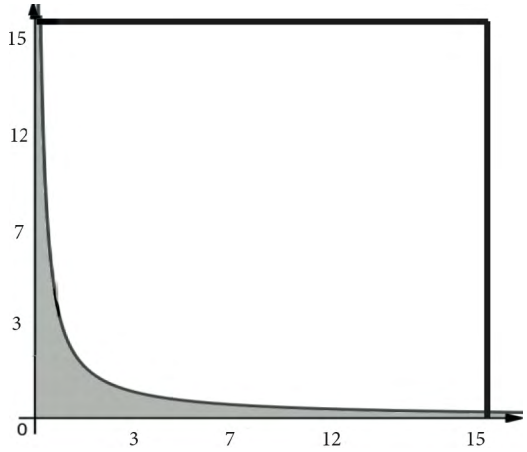


Figure 1. The domain of the predicate x × y ≤ 4.

As observed in Figure 2a, subregion D_3 has no intersection with the gray region. Therefore, as shown in Figure 2b, D_3 could be completely eliminated from the input domain space. After removing D_3 from the input domain space, the number of invalid points will be reduced to 153 and the probability of getting invalid points not satisfying the PC will be decreased from 85% to 79.7%. The remaining subregions, D_1, D_2, and D_4, despite having many invalid points, are considered valid since they include a few gray points. In order to further refine the subregions, as shown in Figure 2c, this time each dimension of the input domain space is subdivided into 4 subdomains.

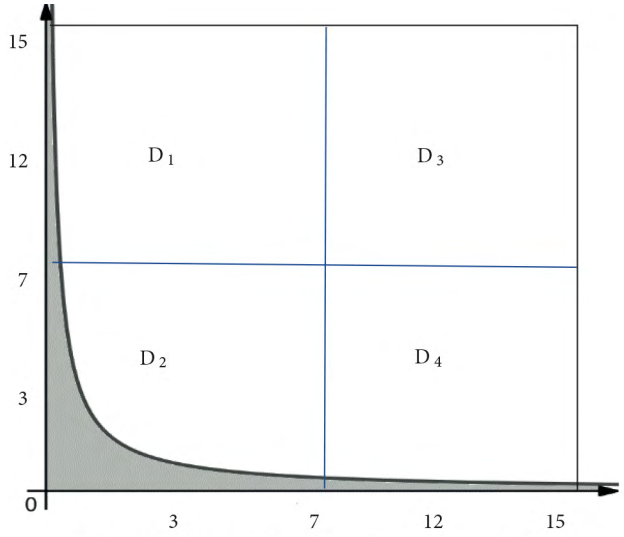
After removing the D5, D_6, D_7, D_9, and D_{11} subregions from the input domain space, the number of invalid points is reduced to 73 and the probability of getting invalid points not satisfying the PC is decreased from 85% to 65.1%. One may carry on with this process of removing the invalid subregions until an acceptable probability is reached.

3. Background

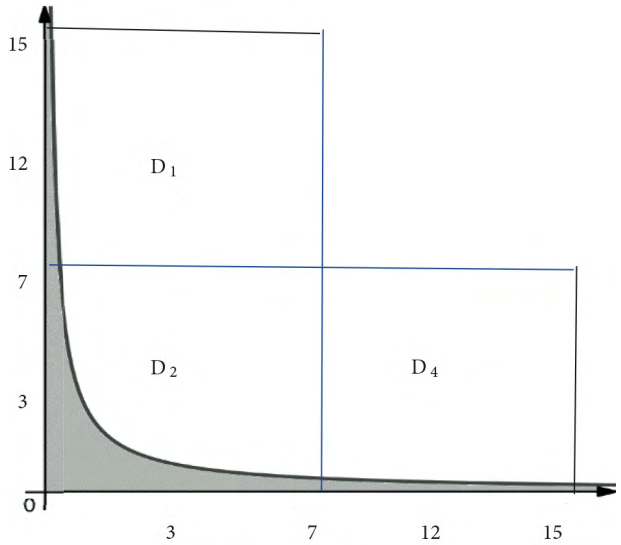
In this section, first various methods that have been proposed to improve the performance of RT are introduced (Section 3.1). In the next section, the use of symbolic execution to derive the constraints of a desired path is explained (Section 3.2). Thereafter, the details of constraint propagation are discussed (Section 3.3). Finally, the PRT method is detailed (Section 3.4).

3.1. Improvements in RT

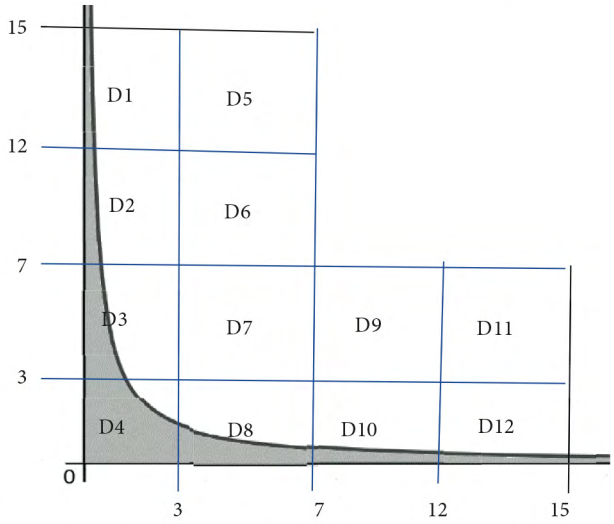
RT is a simple and low-cost software testing technique [3,4]. It randomly selects test cases from the whole input domain space. The test data selection process is continued as long as useful inputs are found. RT is simple and easy to implement, but it may fail to find test data to satisfy the desired coverage criterion. The main reason is that random techniques are blind in the sense that they do not incorporate the program control flow and structure into the process of test data generation. With this aim, search-based test data generation techniques have been developed to improve the efficiency of RT [11]. Search-based test data generation consists of inspecting the input domain of PUT for test data satisfying a selected test data adequacy criterion. For this reason, the focus has been on the use of metaheuristic search and evolutionary algorithms such as hill climbing, simulated annealing, tabu search, and genetic algorithms [1,11–14]. Each of these search-based algorithms is



a. Partitioning the square input domain space into a 2×2 grid.



b. The subdomain D3 can be refuted.



c. Partitioning the input domain space into a finer 4×4 grid for further pruning.

Figure 2. Input domain partitioning: a) partitioning the square input domain space into a 2 × 2 grid; b) the subdomain D3 can be refuted; c) partitioning the input domain space into a finer 4 × 4 grid for further pruning.

strongly dependent on the domain of the problem under consideration because they use heuristics or knowledge related to the problem domain. Each of these algorithms has its own advantages and disadvantages compared to the other algorithms. Search-based approaches often generate test data according to a test adequacy criterion, encoded as a fitness function that is used as an objective for the search [2]. The search-based approach is very generic because different fitness functions can be defined to capture different test data generation objectives. A major difficulty with the heuristic-based techniques is that they do not provide the same results for different runs. Moreover, a heuristic-based approach may take a relatively long execution time before finding reasonable results satisfying its objectives.

In order to speed up the heuristic-based search methods, Chen et al. proposed a method named adaptive random testing (ART) [7]. ART improves heuristic-based and random test data generation methods by focusing

on generating test data that spread them more. ART is based on the observation that failure-causing inputs are most often very close to each other and could be gathered in one or more regions of the program input domain space. In other words, failure-causing inputs are denser in some areas than others. Thus, if previously executed test cases are not failure-causing values, new test cases should be chosen that are very different from them. Accordingly, test data should be uniformly distributed across the input space.

ART methods use a variety of distance calculations, with corresponding computational overhead. Newly proposed methods like partitioning-based ART [15–21] try to decrease the computational overhead while maintaining the performance. For example, the authors of [21] proposed a new ART approach with the aim of decreasing the distance calculation computational overhead while distributing test cases evenly. In [15], the authors proposed an innovative divide-and-conquer approach to improve the efficiency of ART algorithms while maintaining their performance in terms of effectiveness. They made use of the intuition of breaking up a large problem into smaller subproblems and specified a threshold to limit the computational growth when a large number of previously executed test cases are involved in an ART algorithm.

The main objective underlying all the above-mentioned methods is to generate test data in such a way that each path could be executed at least once. However, every practitioner knows that sometimes to detect a latent fault it will be necessary to execute the faulty path several times with different test data before the fault can be detected. In this context, automatic detection of valid input subdomains for executing a given path is undeniable. In this respect, it will be possible to generate as much test data as required.

3.2. Symbolic execution

In symbolic execution, a PUT is executed using symbolic values instead of concrete values for input variables [22–24]. In this method, each execution path is associated with a PC that is the conjunction of all the predicate interpretations that are taken along the path. A predicate is a Boolean expression associated with a conditional statement that determines which fork of the condition will be traversed. A path is executable if and only if the related PC is satisfiable. To find out whether a subdomain has any value that could satisfy the PC, the constraint propagation technique can be used.

3.3. Constraint propagation

Constraint propagation is a deductive activity performed by a propagation system for a problem solver [9,25]. This technique can easily be used to assess the validity of a subdomain D against a PC. To this aim, the PC is ANDed with the constraint that determines the boundary of D and then the added constraint is checked for a solution or contradiction. When there could be a solution, the subdomain D is considered as a valid subdomain, and in case of a contradiction (inconsistency), the subdomain D would be an invalid subdomain. A contradiction occurs when two constraints cannot both be true at the same time and in the same condition. The constraint propagation technique introduces the given constraints into a propagation queue. Then an iterative algorithm manages each constraint one by one in this queue by filtering the domains of variables within the PC for their inconsistent values. For instance, consider again subdomain D3 in Figure 2a. As can be seen from the figure, this subdomain is restricted by the constraints $8 \leq x \leq 15$ and $8 \leq y \leq 15$. To examine its validity, satisfying the constraint $x \times y \leq 4$, at first two constraints, ($x \times y \leq 4$) and ($8 \leq x \leq 15$), are joined to obtain a new constraint, $\left(\frac{4}{15} \leq y \leq \frac{4}{8}\right)$, for y. Thereafter, this constraint is joined with ($8 \leq y \leq 15$), which is a contradiction. Therefore, as expected, it is inferred that subdomain D3 is an invalid subdomain to satisfy $x \times y \leq 4$.

3.4. Path-oriented random testing

PRT [9,25] works like random testing and generates test data randomly to execute a given path according to a uniform probability distribution over the program's input domain. This method first derives the PCs corresponding to a selected path using backward symbolic execution and then separates each variable domain into k equal subdomains. If the domain cannot be divided by k it should be enlarged until it can. By iterating this process over all the n input variables, the input domain is partitioned into k^n subdomains. Then the separated subdomains are checked, one by one, for satisfiability using constraint propagation. The subdomains that could not satisfy the PCs would be refuted. Obviously, removing any portion of the invalid data from the input domain will decrease the number of rejected test data that deviate to execute the desired path. As a result, a uniform random sequence for the input domain can be built by first generating a uniform random sequence over obtained subdomains and then picking up a single tuple in each subdomain at random.

4. The proposed method

The inspiration behind our suggested algorithm for automatically generating a sequence of valid test data exercising a desired path comes from the difficulties and defects observed in the PRT method. As mentioned in Section 3.4, considering n input variables for a PUT the PRT method divides the input domain into k^n subdomains and checks their satisfiability against the PC by a constraint propagation algorithm, which is a time-consuming process. Certainly, the less the need for invocation of the constraint propagation algorithm, the less the CPU usage will be. This can be achieved through an iterative approach that successively partitions the input domain from a coarse scheme to a finer one. Doing this speeds up the suggested algorithm by reducing the number of invocations of the constraint propagation algorithm. The reason is that in each iteration a large portion of the invalid test data might be omitted and would not be assessed in the next iterations. For instance, consider again the motivating example and assume k = 16. In the PRT method $16^2 = 256$ subdomains would be created, which need to be examined against PC for the satisfiability, but with the iterative technique, this would be decreased to 105. This is shown in Figure 3.

As shown in the figure, it is assumed that the input domain is partitioned iteratively into 4, 16, 64, and 256 subdomains. The number of subdomains that need to be checked for satisfiability in each partitioning scheme is shown in Table 1.

Table 1. The number of subdomains that need to be checked in each partitioning scheme.

Partition scheme	Number of subdomains that need to be checked
Initially	1
2×2	4
4×4	12
8×8	28
16×16	60

At first, the whole input domain is checked for satisfiability, and then the input domain is partitioned into 4 subdomains, all of which are checked against the PC, and only one subdomain is refuted for further consideration. Considering the refuted subdomain, when the input domain space is partitioned into 16 subdomains, only 12 subdomains need to be checked. This value would be changed to 28 and 60 when the input domain is partitioned into 64 and 256 subdomains, respectively. In total, $1 + 4 + 12 + 28 + 60 = 105$ subdomains would be checked, which is a remarkable result compared to the PRT method.

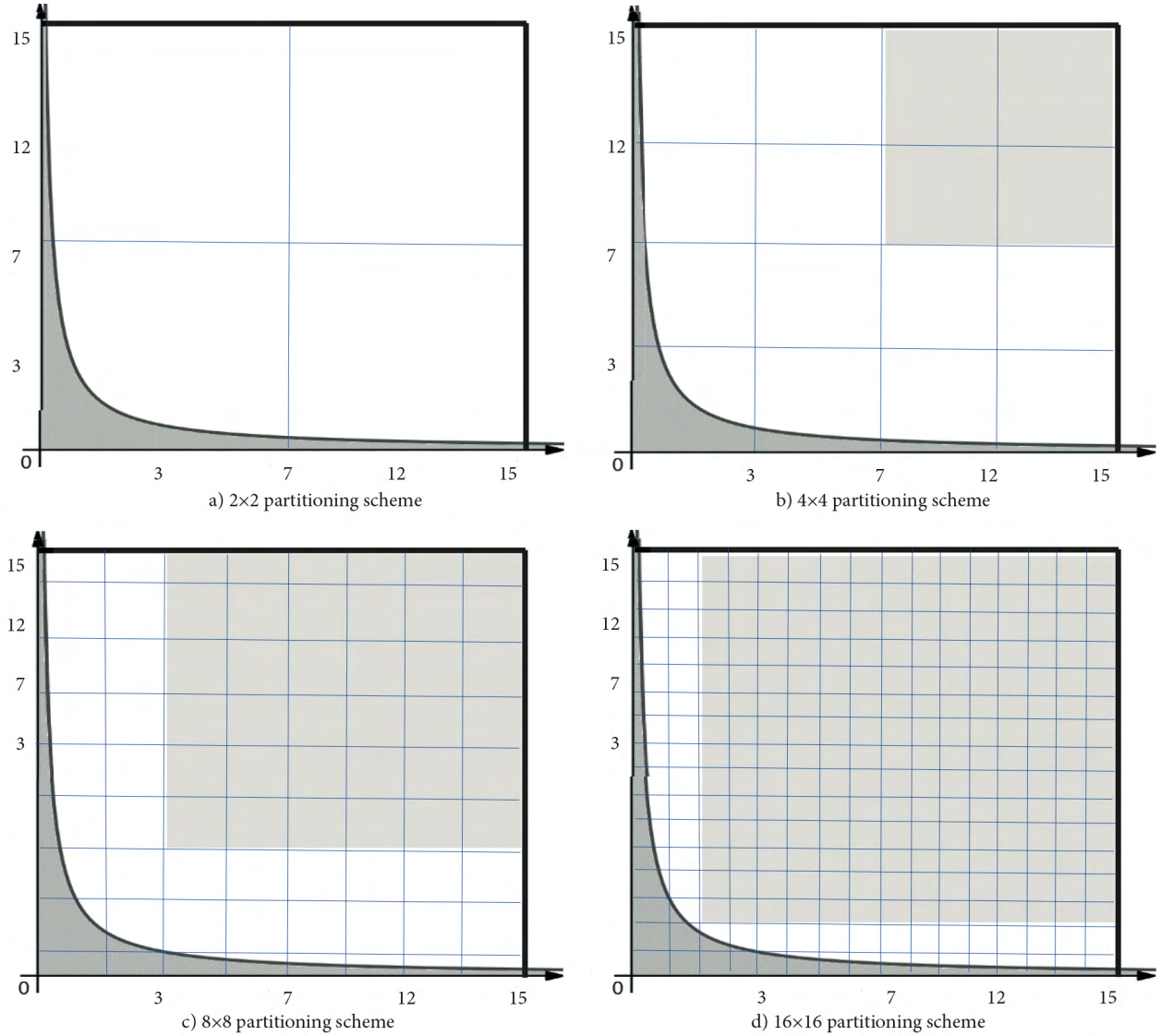


Figure 3. The valid subdomains in the various partitioning scheme: a) 2 × 2 partitioning scheme; b) 4 × 4 partitioning scheme; c) 8 × 8 partitioning scheme; d) 16 × 16 partitioning scheme.

4.1. Overview of the IP-PRT algorithm

In IP-PRT, the input domain is divided into a regular grid of equally sized cells. The grid cells are categorized as valid and invalid cells according to their relative subdomain validity. The related subdomains of each valid cell are checked for satisfiability over the PC using constraint propagation. If adding a subdomain boundary to the PC leads us to a contradiction, then the subdomain is considered as an invalid subdomain and the related cell is marked as invalid. After checking all the subdomains, the current partitioning scheme will be discarded and a finer partitioning scheme will be applied, and then all the invalid cells will be mapped into the new partition. This process can be repeated until the testing resources are exhausted or the number of iterations exceeds a threshold value.

4.2. Grid coordinates used in IP-PRT

In IP-PRT, the whole input domain is iteratively divided into equally sized subdomains until the size of the grid cells reaches a certain threshold value, specified by the tester. The coordinates of the subdomains for each variable are kept in a grid cell, representing the domain space. In addition to the coordinates, in each cell, the validity of the corresponding subdomain, determined by applying a constraint propagation algorithm, is recorded.

Assume there is a function p(int x, int y) where the parameters x and y are bounded as $0 \leq$ x,y $<$ M and suppose the input domain is partitioned by a $k \times k$ grid, where k is a positive integer given by the tester. Let $C = \frac{M}{K}$ indicate the size of each grid cell, and then the boundaries of the grid cell, GridCells(i, j), could be computed by applying the following relations:

$$(i-1) \times C + 1 \leq x \leq i \times C and (j-1) \times C + 1 \leq y \leq j \times C.$$

For instance, as shown in Figure 4, the grid cell (3, 2) refers to the subdomain in which the boundaries of input variables x and y are 2C + 1 $\leq$ x $\leq$ 3C and C + 1 $\leq$ y $\leq$ 2C

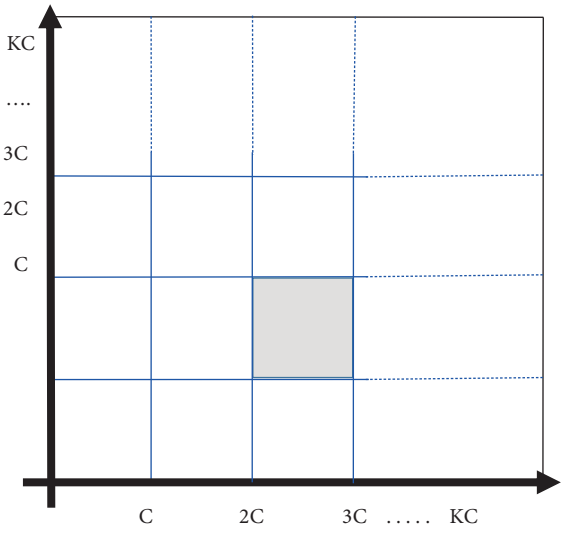


Figure 4. The coordinates of cell (3,2).

4.3. A divide-and-conquer algorithm

We present an algorithm that generates a sequence of random test data with a uniformly distributed probability. The algorithm takes as inputs a set of variables along with their variation domain, PC a constraint set corresponding to the PCs of the selected path, N the length of the expected random sequence, and a threshold that determines the number of iterations. The threshold is taken to control the depth of the iterations.

To perform IP-PRT, first we need to determine the number of grid cells in partitioning the input domain of each variable. If it is at the early stage of the partitioning, a coarse grid is appropriate because a large portion of the invalid subdomains might be omitted at the beginning. Hence, the algorithm starts with a coarse grid. In the next iterations, the current partitioning scheme will be discarded and a finer partitioning scheme will be applied to partition the input domain all over again. The proposed algorithm is elaborated for a 2-dimensional square input domain with a size of M $\times$ M. Extension to input domains of higher dimensions is straightforward.

The Boolean matrix GridCells is used to represent the partitioning grid. Each entry of the grid corresponds to a grid cell. If a cell entry corresponds to an invalid subdomain, then it will be assigned a value of F; otherwise, it corresponds to a valid subdomain cell and will be assigned a value of T. In the algorithm, k indicates the number of subdomains in each iteration of the partitioning scheme. For example, k = 2 indicates a 2 × 2 partitioning grid.

Algorithm: IP-PRT

Input: x, y, PC, N, threshold
Output: $t_1, \ldots, t_N$ or ϕ (infeasible path)

```
1.  T:= ϕ;
2.  k:=1;
3.  while k ≤ threshold do
4.      Discard (release) the Boolean matrix GridCells. Set k = k × 2.
5.      Construct a k × k Boolean matrix, GridCells, and for each cell GridCells[i,j] in the previous
        partition scheme with value F assign F to all its entries in the new partitioned scheme that
        correspond to cells GridCells [(i–1) × 2 + 1,(j–1) × 2 + 1], GridCells [(i–1) × 2 + 1,(j–1) ×
        2 +2], GridCells [(i–1) × 2 + 2,(j–1) × 2 +1], GridCells [(i–1) × 2 + 2,(j–1) × 2 + 2];
6.      for i: = 1 to k do
7.          for j: = 1 to k do
8.              if GridCells [i,j] = T then
9.                  if D_{i,j} is inconsistent with respect to PC then
10.                     GridCells [i,j]: = F;
11.                 end if;
12.             end if;
13.         end for;
14.     end for;
15. end while;
16. Let D1', . . . , Dp' be the relative subdomains of the GridCells for which its cells have value T;
17. if p ≥ 1 then
18.     while N > 0 do
19.         Pick up uniformly D at random from D1', . . . , Dp'
20.         Pick up uniformly t at random from D;
21.         if PC is satisfied by t then
22.             add t to T;
23.             N: = N–1;
24.         end if
25.     end while
26. end if
return T;
```

Variable k is considered as a division parameter and determines the partitioning scheme in each iteration. First, the algorithm partitions the hypercuboid into a 2 × 2 grid and then each relative subdomain in the grid cells is checked for nonsatisfiability. In the next iteration the division parameter k is changed to 2 × k, the previous partitioning scheme is discarded, the input domain is partitioned again using a 4 × 4 grid, and all the invalid cells are mapped into the new partition scheme. This process can be repeated until the size of the subdomains is small enough (the number of iterations exceeds the threshold value).

Secondly, a uniform random test data generator is built from the valid cells by first picking up a subdomain and then picking up a tuple inside this subdomain. If the selected tuple does not satisfy the PC, then it is simply rejected. This process is repeated until a sequence of N test data is generated.

5. Experimental results and discussion

To assess the effectiveness of the IP-PRT algorithm, we have implemented a prototype and evaluated it on a set of well-known benchmark examples, comparing RT and PRT in terms of the achieved generated test data and CPU time consumption by regularly increasing the desired length of the random test data. For the implemented prototype to solve the constraints, the open source constraint solver Choco[1] is exploited [33].

Our RT implementation iteratively generates new test data randomly, with a uniform distribution probability, and accepts them provided that they satisfy the PC.

In PRT and IP-PRT implementation, after partitioning the input domain and refuting the invalid subdomains, first a subdomain is picked from the remaining subdomains and then a tuple is selected inside these subdomains. Selection of the subdomains and tuples within the subdomains is random with uniform distribution probability. Furthermore, PRT and IP-PRT are evaluated with several distinct values of k. Considering that the achieved subdomains in both methods are identical, they have the same conditions for generating test data. Hence, as expected, the number of generated test data with these two methods is nearly the same value. According to this, in the experimental results only the number of test data generated by the IP-PRT method is shown.

To apply the methods over the benchmark, first a list containing of all the execution paths for each benchmark is created. Then, in a loop, the paths are chosen one by one and given to the methods. Each method performs 20 independent runs over the given path. For the benchmark, which has loops, we have selected a path such that each loop iterates between 4 and 20 times. As a result, the average value of 20 runs over all paths is considered.

All experiments were run on a 64-bit, 2.10 GHz Intel CoreTM i7 computer running Microsoft Windows 7 with 8 GB memory.

5.1. Programs to be tested

To evaluate IP-PRT we perform experiments over three widely used programs, *remainder*, *trityp*, and *Middle*, in the research area of software testing. The *remainder* program takes as input two integers and returns the remainder by dividing the input values using only subtractions and comparisons. The *trityp* program takes three nonnegative integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene, isosceles, equilateral, or illegal. The *Middle* program takes 3 input variables and returns the variable having a value between the other two. If two variables have the same value, then the third one is reported as a middle. First of all, we show the results achieved with the motivating example.

5.2. Experiments based on the motivating example

Table 2 reports the obtained results for the predicate $x \times y \leq 4$ of the motivating example while increasing the requested length of the random test data sequence from 10 to 300. In this table, the first column shows that the number of rejects of the RT method is 72 – 10 = 62 test data with CPU time 0.35 ms and the number of rejects of the PRT and IP-PRT methods when k = 2 is 43 with CPU time 0.01 ms, and so on.

1. **Experiments on the *trityp*, *Middle*, and *remainder* programs**

For the *trityp*, *Middle*, and *remainder* programs, first we extract a list of the paths with their associated path conditions that cover all the decisions of the program. For the program *remainder,* which has a loop, each path

[1] Choco is an open source Java constraint programming library: http://www.emn.fr/z-info/choco-solver/uploads/pdf/choco-presentation.pdf

Table 2. The required CPU time and length of the test data generated for predicate x $\times$y $\leq$ 4.

Requested		10	20	30	40	50	100	200	300
RT	Test data	72	138	203	269	334	661	1321	1976
	CPU time	0.35 ms	0.72 ms	1.07 ms	1.3 ms	1.7 ms	3.2 ms	7.5 ms	10.45 ms
k = 2	Test data	53	103	152	203	251	497	989	1482
	CPU time (PRT)	0.01 ms	0.01 ms	0.02 ms	0.02 ms	0.03 ms	0.07 ms	0.11 ms	0.28 ms
	CPU time (IP-PRT)	0.01 ms	0.01 ms	0.02 ms	0.02 ms	0.03 ms	0.07 ms	0.14 ms	0.31 ms
k = 4	Test data	31	60	88	117	146	290	577	864
	CPU time (PRT)	0.11 ms	0.14 ms	0.17 ms	0.19 ms	0.20 ms	0.28 ms	0.35 ms	0.43 ms
	CPU time (IP-PRT)	0.08 ms	0.08 ms	0.09 ms	0.09 ms	0.10 ms	0.13 ms	0.19 ms	0.27 ms
k = 8	Test data	17	33	48	64	80	159	317	474
	CPU time (PRT)	0.45 ms	0.49 ms	0.54 ms	0.58 ms	0.65 ms	0.84 ms	0.88 ms	0.93 ms
	CPU time (IP-PRT)	0.15 ms	0.16 ms	0.17 ms	0.17 ms	0.18 ms	0.24 ms	0.31 ms	0.47 ms
k = 16	Test data	10	20	30	40	50	100	200	300
	CPU time (PRT)	1.03 ms	1.09 ms	1.14 ms	1.19 ms	1.23 ms	1.79 ms	2.04 ms	2.29 ms
	CPU time (IP-PRT)	0.28 ms	0.28 ms	0.32 ms	0.37 ms	0.41 ms	0. 95 ms	1.35 ms	1.65 ms

is selected in such a way that the loop iterates between 4 and 20 times. The domain of input variables is confined in the range of [0…63] and we compare the methods while generating random test data of increasing lengths from 10,000 to 80,000. The experimental results are given in Table 3, Table 4, and Table 5 for the programs *trityp*, *Middle*, and *remainder*, respectively. The PRT and IP-PRT methods are compared with the three distinct values of the division parameter k. The results demonstrate that the proposed method outperformed RT to minimize the amount of the rejected test data with less CPU usage. Furthermore, the results show that the proposed method improves the PRT method in terms of CPU time consumption.

Table 3. Experimental results of the program *trityp*.

Requested		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000
RT	Test data	293,372	591,298	874,846	1,163,378	1,451,149	1,745,932	2,039,450	2,329,345
	CPU time	35.4 s	67.2 s	89.9 s	127.3 s	151.7 s	182.9 s	216.9 s	243.2 s
k = 4	Test data	144,927	289,512	432,782	582,120	731,003	869,305	1,014,498	1,159,410
	CPU time (PRT)	2.9 s	6.6 s	13.2 s	18.2 s	23.1 s	29.2 s	32.8 s	42.8 s
	CPU time (IP-PRT)	2.8 s	5.3 s	9.4 s	12.5 s	15.8 s	19.9 s	25.2 s	34.1 s
k = 8	Test data	78,125	156,195	234,405	312,530	390,604	468,850	546,775	625,010
	CPU time (PRT)	10.73 s	25.2 s	48.6 s	67.9 s	84.9 s	109.1 s	128.6 s	158.8 s
	CPU time (IP-PRT)	10.3 s	18.1 s	32.3 s	41.6 s	56.9 s	71.4 s	95.5 s	116.8 s
k = 16	Test data	39,835	79,581	119,621	159,303	199,265	239,143	278,784	318,715
	CPU time (PRT)	17.6 s	41.3 s	80.1 s	111.9 s	139.2 s	180.4 s	218.4 s	258.9 s
	CPU time (IP-PRT)	9.6 s	22.7 s	45.8 s	62.7 s	87.4 s	112.5 s	151.2 s	194.3 s

The obtained results for the three programs *trityp*, *Middle*, and *remainder* are summarized and presented in Table 6. In this table, the percentage of improvement with respect to CPU usage achieved by IP-PRT as compared to PRT is presented. The results reveal that IP-PRT outperforms PRT significantly in all cases. In particular, it is found that higher improvement is achieved when the division parameter, k, is increased. For instance, in the case of the *trityp* program, by using IP-PRT, up to 15.1% improvement may be obtained when

Table 4. Experiments results of the program *Middle*.

Requested		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000
RT	Test data	31,706	62,831	94,086	125,067	157,154	188,811	220,063	252,188
	CPU time	4.2 s	8.0 s	11.6 s	15.7 s	19.3 s	23.9 s	28.1 s	33.7 s
k = 4	Test data	21,525	43,071	64,486	86,052	107,577	129,002	150,638	172,103
	CPU time (PRT)	1.2 s	2.1 s	4.1 s	4.9 s	5.6 s	6.3 s	7.2 s	7.9 s
	CPU time (IP-PRT)	1.0 s	2.1 s	3.7 s	4.5 s	5.2 s	6.1 s	6.9 s	7.1 s
k = 8	Test data	14,275	28,450	42,715	56,988	71,225	85,507	99,795	114,009
	CPU time (PRT)	2.3 s	4.5 s	6.1 s	7.9 s	9.3 s	10.1 s	11.9 s	12.8 s
	CPU time (IP-PRT)	1.6 s	2.8 s	4.8 s	6.7 s	7.9 s	8.8 s	10.1 s	10.9 s
k = 16	Test data	12,417	24,784	37,301	49,618	62,130	74,392	86,819	99,286
	CPU time (PRT)	3.5 s	6.9 s	11.1 s	13.2 s	15.1 s	16.8 s	17.9 s	19.1 s
	CPU time (IP-PRT)	2.3 s	4.1 s	8.1 s	10.8 s	11.9 s	12.8 s	14.2 s	15.6 s

Table 5. Experiments results of the program *Remainder*.

Requested		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000
RT	Test data	3,135,816	6,293,698	9,385,870	12,555,939	15,868,593	18,817,339	22,064,590	25,108,365
	CPU time	27.3 s	58.9 s	75.8 s	98.0 s	114.8 s	154.6 s	181.3 s	213.7 s
k = 4	Test data	135,107	261,817	409,109	491,200	612,908	721,522	895,463	972,101
	CPU time (PRT)	2.1 s	5.9 s	11.4 s	13.1 s	15.2 s	17.0 s	18.8 s	20.3 s
	CPU time (IP-PRT)	2.1 s	5.7 s	10.9 s	12.8 s	14.3 s	15.9 s	17.1 s	18.9 s
k = 8	Test data	65,340	110,234	189,320	278,901	312,009	391,023	442,981	569,012
	CPU time (PRT)	4.5 s	8.1 s	11.9 s	15.3 s	18.0 s	22.3 s	24.5 s	26.1 s
	CPU time (IP-PRT)	3.1 s	6.4 s	9.7 s	12.7 s	14.6 s	17.1 s	19.5 s	21.5 s
k = 16	Test data	34,781	71,240	92,194	105,672	128,923	192,903	231,982	290,823
	CPU time (PRT)	8.3 s	15.7 s	21.1 s	29.6 s	37.2 s	53.8 s	61.0 s	69.9 s
	CPU time (IP-PRT)	5.1 s	10.9 s	16.8 s	21.4 s	26.1 s	36.5 s	42.4 s	51.7 s

the value of k is 4 and the requested number of test data to be generated is 20,000. It is worth mentioning that the amount of improvement increases to about 45% when the value of k is increased to 16. A similar observation could be made in other cases. The main reason behind this improvement is that IR-PRT significantly reduces the number of invocations of the constraint propagation algorithm.

5.3. Discussion

In this section, we provide a discussion on the negative effects of eliminating input subdomains and the ability of IP-PRT in dealing with PUTs having large input domain spaces. Determining the validity of a subdomain with respect to a PC is dependent on the constraint propagation algorithm. Thus, similar to other related works [9,14,26], a valid (or invalid) subdomain may be considered as invalid (or valid) and refuted (or kept) by the proposed method. Elimination of a whole input subdomain because it consists of negative and positive test results may cause the test runner to miss some false positive errors, especially corner cases, in the proposed method. Please note that other related methods, like PRT, employing a constraint propagation algorithm also suffer from this issue, which may increase the number of invalid generated test data. Similar to PRT, our proposed algorithm is semicorrect, meaning that when it terminates, it is guaranteed to provide the correct

Table 6. The summarized results.

Programs		Requested							
		10,000	20,000	30,000	40,000	50,000	60,000	70,000	80,000
trityp	k = 4	3.4%	15.1%	28.7%	31.2%	31.6%	31.8%	23.1%	20.3%
	k = 8	4.0%	28.1%	33.5%	38.7%	32.9%	34.5%	25.7%	26.4%
	k = 16	45.4%	45%	42.8%	43.9%	37.2%	37.6%	30.7%	24.9%
Middle	k = 4	16.6%	0%	9.7%	8.1%	7.1%	3.1%	4.1%	10.1%
	k = 8	30.4%	37.7%	21.3%	15.1%	15.0%	12.8%	15.1%	14.8%
	k = 16	34.2%	40.5%	27.0%	18.1%	21.1%	23.8%	20.6%	18.3%
remainder	k = 4	0%	3.3%	4.3%	2.2%	5.9%	6.4%	9.0%	6.8%
	k = 8	31.1%	20.9%	18.4%	16.9%	18.8%	23.3%	20.4%	17.6%
	k = 16	38.5%	30.5%	20.3%	27.7%	29.8%	32.1%	30.4%	26%

expected result, but it is not guaranteed to terminate. Note that similar problems arise with random testing or path testing as nothing prevents an unsatisfiable goal PC from being selected. In that case, all the test cases will be rejected. In practice, a time-out mechanism is necessary to enforce termination. This mechanism is not detailed here but it is mandatory for actual implementations. Note that any testing tools that execute programs should be equipped with such a time-out mechanism as nothing prevents a tested program from activating an endless path.

It is worth mentioning that the input domain of a program is restricted to the Cartesian product of the bounded intervals of the program input variable boundaries. Thus, the Cartesian product of the bounded intervals of n variables can be represented as an n-dimensional hypercube, which is the n-dimensional extension of the 2-dimensional cuboid. Therefore, our proposed method can handle large domain spaces containing many input variables by iteratively partitioning the hypercube to subhypercubes.

6. Threats to validity

In this section, the potential threats to the validity of our studies, including external and internal validity, are explained. We use Java language to implement our tool for test data generation. Threats to internal validity concern possible errors in our implementations that could affect our results. In this regard, we carefully checked most of our results for decreasing these threats considerably. The main threat to external validity is that our experiments are restricted to only small or medium-sized programs. More experiments on larger programs may further strengthen the external validity of our findings. Further investigations of other programs in different programming languages would also help generalize the obtained results.

7. Concluding remarks

In this paper, we propose a path-oriented automatic random testing method through the use of constraint propagation. In the proposed method a simple divide-and-conquer algorithm is introduced that permits us to efficiently build a uniform sequence of test data exercising a selected path. After obtaining the constraint set along with a selected path by the symbolic execution techniques, the reduced input subdomain can be computed by dividing the initial input domain iteratively and refuting such subdomains that cannot satisfy the constraint. As a result, the random testing effectiveness can be remarkably enhanced by the generated test data from the reduced input domain. We showed that the proposed method outperforms traditional RT and PRT.

References

[1] Swain S, Mohapatra DP. Genetic algorithm-based approach for adequate test data generation. In: Mohapatra D, Patnaik S (editors). Intelligent Computing, Networking, and Informatics. New Delhi, India: Springer, 2014, pp. 453-462.

[2] Sahin O, Akay B. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. Applied Soft Computing 2016; 49 (1): 1202-1214. doi: 10.1016/j.asoc.2016.09.045

[3] Hamlet D. When only random testing will do. In: Proceedings of the First International Workshop on Random Testing; Portland, ME, USA; 2006. pp. 1-9.

[4] Liu H, Chen TY. Randomized quasi-random testing. IEEE Transactions on Computers 2016; 65 (6): 1896-1909. doi: 10.1109/TC.2015.2455981

[5] Groce A, Holzmann G, Joshi R. Randomized differential testing as a prelude to formal verification. In: 29th International Conference on Software Engineering; Minneapolis, MN, USA; 2007. pp. 621-631.

[6] Arcuri A, Iqbal MZ, Briand L. Formal analysis of the effectiveness and predictability of random testing. In: Proceedings of the 19th International Symposium on Software Testing and Analysis; Torento, Italy; 2010. pp. 219-230.

[7] Chen TY, Kuo FC, Merkel RG, Tse TH. Adaptive random testing: the art of test case diversity. Journal of Systems and Software 2010; 83 (1): 60-66.

[8] Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation; Chicago, IL, USA; 2005. pp. 213-223.

[9] Gotlieb A, Petit M. A uniform random test data generator for path testing. Journal of Systems and Software 2010; 83 (12): 2618-2626. doi: 10.1016/j.jss.2010.08.021

[10] Dinges P, Agha G. Targeted test input generation using symbolic-concrete backward execution. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, Västerås, Sweden; 2014. pp. 31-36.

[11] McMinn P. Search-based software test data generation: a survey. Software Testing, Verification and Reliability 2004; 14 (2): 105-156. doi: 10.1002/stvr.294

[12] Harman M, McMinn P. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis; Washington, DC, USA; 2007. pp. 73-83.

[13] Şahin O, Akay B. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. Applied Soft Computing 2016; 49 (1): 1202-1214. doi: 10.1016/j.asoc.2016.09.045

[14] Feyzi F, Parsa S. Bayes-TDG: Effective test data generation using Bayesian belief network: towards failure-detection effectiveness and maximum coverage. IET Software 2018; 12: 225-235. doi: 10.1049/iet-sen.2017.0112

[15] Chow C, Chen TY, Tse TH. The art of divide and conquer: an innovative approach to improving the efficiency of adaptive random testing. In: Proceedings of 13th International Conference on Quality Software; Nanjing, China; 2013. pp. 268-275.

[16] Chen TY, Huang DH, Zhou ZQ. On adaptive random testing through iterative partitioning. Journal of Information Science and Engineering 2011; 27 (4): 1449-1472.

[17] Chan KP, Chen TY, Towey D. Normalized restricted random testing. In: International Conference on Reliable Software Technologies; Berlin, Germany; 2003. pp. 368-381.

[18] Chen TY, Kuo FC, Merkel RG, Ng SP. Mirror adaptive random testing. Information and Software Technology 2004; 6 (15): 1001-1010. doi: 10.1016/j.infsof.2004.07.004

[19] Chen TY, Merkel R, Wong PK, Eddy G. Adaptive random testing through dynamic partitioning. In: Proceedings of Fourth International Conference on Quality Software; Braunschweig, Germany; 2004. pp. 79-86.

[20] Nikravan E, Feyzi F, Parsa S. Enhancing path-oriented test data generation using adaptive random testing techniques. In: Proceedings of 2nd International Conference on Knowledge-Based Engineering and Innovation; Tehran, Iran; 2015. pp. 1-14.

[21] Sabor KK, Thiel S. Adaptive random testing by static partitioning. In: Proceedings of the 10th International Workshop on Automation of Software Test; Florence, Italy; 2015. pp. 28-32.

[22] Baldoni R, Coppa E, D'Elia DC, Demetrescu C, Finocchi I. A survey of symbolic execution techniques. ACM Computing Surveys 2018; 51 (3): 50. doi: 10.1145/3182657

[23] Braione P, Denaro G, Mattavelli A, Pezzè M. Combining symbolic execution and search-based testing for programs with complex heap inputs. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis; Santa Barbara, CA, USA; 2017. pp. 90-101

[24] Qiu R, Păsăreanu CS, Khurshid S. Certified symbolic execution. In: International Symposium on Automated Technology for Verification and Analysis; Chiba, Japan; 2016. pp. 495-511.

[25] Gotlieb A, Petit M. Path-oriented random testing. In: Proceedings of the 1st International Workshop on Random Testing; Portland, ME, USA; 2006. pp. 28-35.

[26] Offutt AJ, Jin Z, Pan J. The dynamic domain reduction procedure for test data generation. Software Practice and Experience 1999; 29 (2): 167-193.