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

Esmaeel Nikravan, Saeed Parsa*

Nikravan@comp.iust.ac.ir, Parsa@iust.ac.ir

School of Computer Engineering, Iran University of Science and Technology, Tehran, Iran

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 sub-domains, not covering the path. The validity of the sub-domains is checked by a constraint propagation method, against the path constraints. The proposed algorithm uses a divide and conquers technique to iteratively split the inputs domain, into subdomains and each time refutes those subdomains, inconsistent with the path constraints. The remaining shrunken sub-domains, altogether provide all possible test data covering the desired path. Obviously, the more accurate input domain the more effective test data will result. Experiments show the proposed method outperformed the other related method on a set of classical benchmark programs.

Keyword: 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, the domain of input variables should be sanitized from those values which do not cover the faulty path.

Amongst 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 the coverage capability [5] while, many studies show that RT is an effective method in detecting faults, not found by the 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 is generated without considering the specification or the structure of the PUT [7, 8]. RT is applied by an approach, 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 (PC) and then divides the domain of all the input variables within the derived PC into k equally sized subdomains. Thereafter, the validity of each of the k sub-domains, against the path constraint, 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 sub-domain to ensure if there is at least one value consistent with the PC. The difficulty is that the constraint propagation algorithm cannot detect sub-domains with no values inconsistent with the PC. PRT removes all those sub-domains, not covering the PC. However, when selecting points from within the remaining sub-domains, not every point do necessarily satisfy the path constraint. 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 bigger the number of the sub-domains, to be evaluated against the PC. On the other hand, if k is small then the resultant sub-domains 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 sub-dividing the sub-domains, including at least a value, consistent with the PC as far as the number of invalid points in the program input domain space, reduces to a reasonable number. To keep track of the remaining valid sub-domains, 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 resultant 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 and 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 the 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 path constraint, x×y<=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, …, 15} × {0, …, 15} plain in which the grey region includes all valid points, (x_i, y_i) satisfying the path constraint, x×y<=4. The greyed region contains 39 points, covering about 15% of the whole input domain space. Therefore when using uniformly distributed random test data selection scheme, the probability of getting an invalid input value, not satisfying the path constraint, will be 85%.

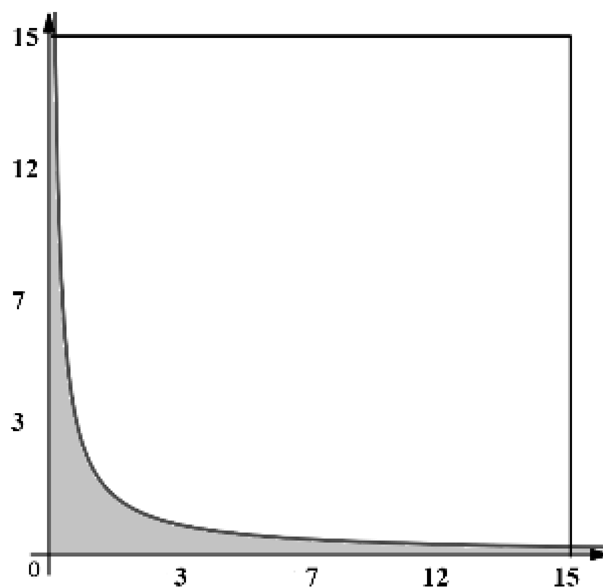


Figure 1. The domain of the predicate **x×y<=4**

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

1. Each dimension of the input space is subdivided into k equal sized sub-regions where k is a power of 2.

2. The validity of each sub-region is examined against the path constraint by a constraint propagation algorithm.

1 **3.** All those sub-regions which contain at least a point, satisfying the path constraint are

2 kept and the remaining which do not include any valid points are refuted.

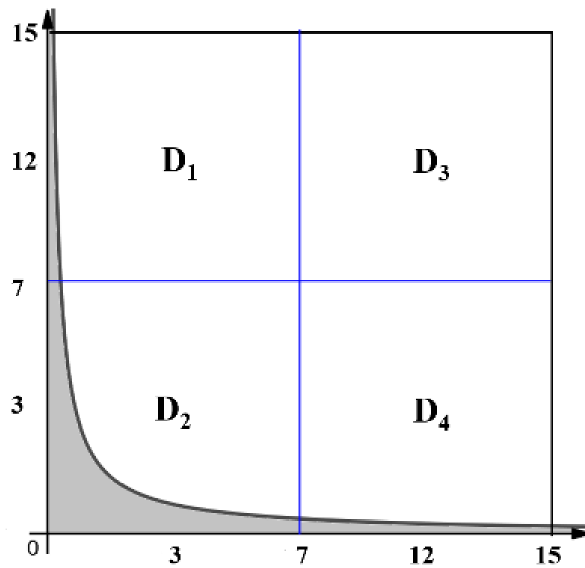
3 **4.** Repeat to step 1 as far as the number of iteration exceeds a threshold value.

4 For instance, as shown in Figure 2.a, with dividing each dimension of the initial input

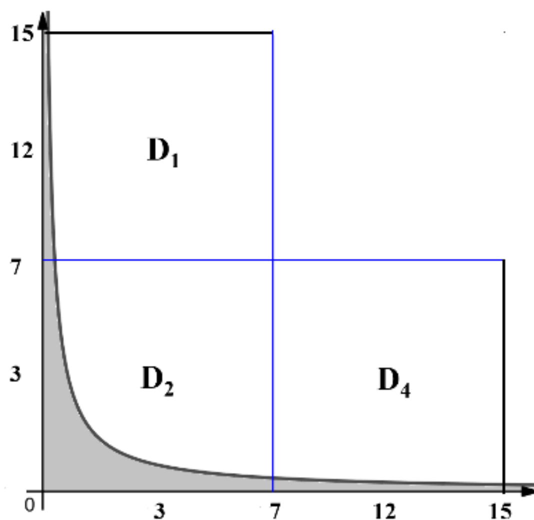
5 domain space, in Figure 1, into two distinctive subdomains 0…7 and 8…15, we get the

6 following 4 subdomains: D_1 = (x ∈ 0 ... 7, y ∈ 8 ... 15), D_2 = (x ∈ 0 ... 7, y ∈ 0 ... 7), D_3 =

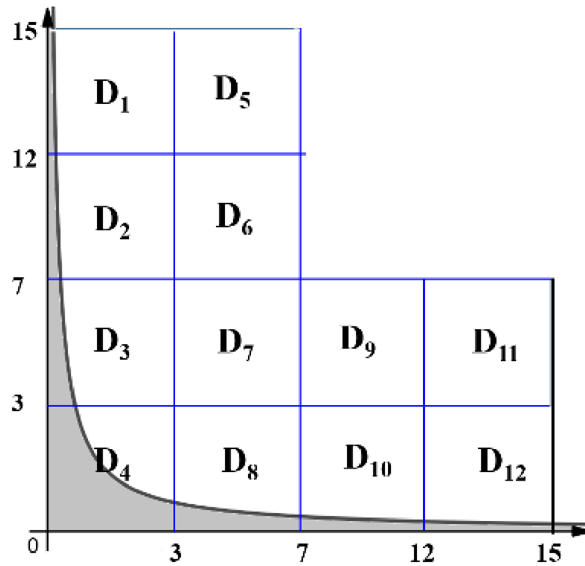
7 (x ∈ 8 ... 15, y ∈ 8 ... 15), D_4 = (x ∈ 8 ... 15, y ∈ 0 ... 7).



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

As it is observed in Figure 2.a, the sub-region D_3 has no intersection with the greyed region. Therefore, as shown in Figure 2.b, D_3 could be completely eliminated from the input domain space. Removing D_3 from the input domain space, the number of invalid points will be reduced to 153 points and the probability of getting invalid points, not satisfying the path constraint, will be decreased from 85% to 79.7%. The remaining sub-regions, D_1, D_2, and D_4, despite having many invalid points are considered as valid since, they include a few greyed points. In order to further refine the sub-regions, as shown in Figure 2.c, this time, each dimension of the input domain space is sub-divided into 4 sub-domains.

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

3. Background

In this section, at 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 detail of the constraint propagation is discussed (Section 3.3). Finally, the PRT method is detailed (Section 3.4).

3.1. Improvements in random testing

Random testing (RT) is a simple and low-cost software testing technique [3, 4]. RT randomly selects test cases from the whole input domain space. The test data selection process is continued as far 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. To this aim, search-based test data generation techniques have been developed to improve the efficiency of RT [15]. 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, genetic algorithms [1, 15, 16, 17, 18]. Each of these search-based algorithms is 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 it could find 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 which 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 the others. Thus, if previously executed test cases are not a failure causing value, new test cases should be chosen far 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 [19-25] try to decrease computational overhead while maintaining the performance. For example, authors in [25] proposed a new ART approach with the aim of decreasing the distance calculation computational overhead, while distributing test cases evenly. In another work [19], authors propose an innovative divide-and-conquer approach to improve the efficiency of ART algorithms while maintaining their performance in effectiveness. They make use of the intuition of breaking up a large problem into smaller sub-problems and specify 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 would be required to execute the faulty path several times with different test data before the fault could 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 many 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 [11, 12, 13]. In this method, each execution path is associated with a path constraint (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 which 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, 14]. This technique can easily be used to assessing the validity of a subdomain D against a path constraint PC. To this aim PC is ANDed with the constraint which determines the boundary of D 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

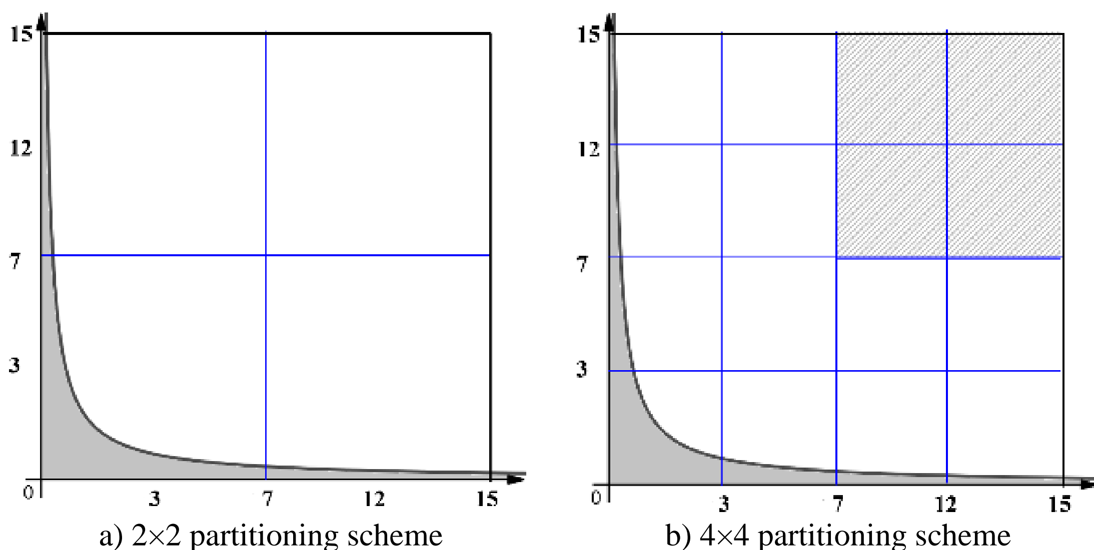
introduces the given constraints into a propagation queue. Then, an iterative algorithm manages each constraint one by one into this queue by filtering the domains of variables, within PC, of their inconsistent values. For instance, consider again the subdomain D3 in Figure 2.a. as can be seen from the figure, this subdomain is restricted by the constraint $8 \leq x \leq 15$ and $8 \leq y \leq 15$. To examine its validity, satisfying the constraint $x \times y <= 4$, 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 the subdomain D3 is an invalid subdomain to satisfy $x \times y <= 4$.

3.4. Path oriented random testing

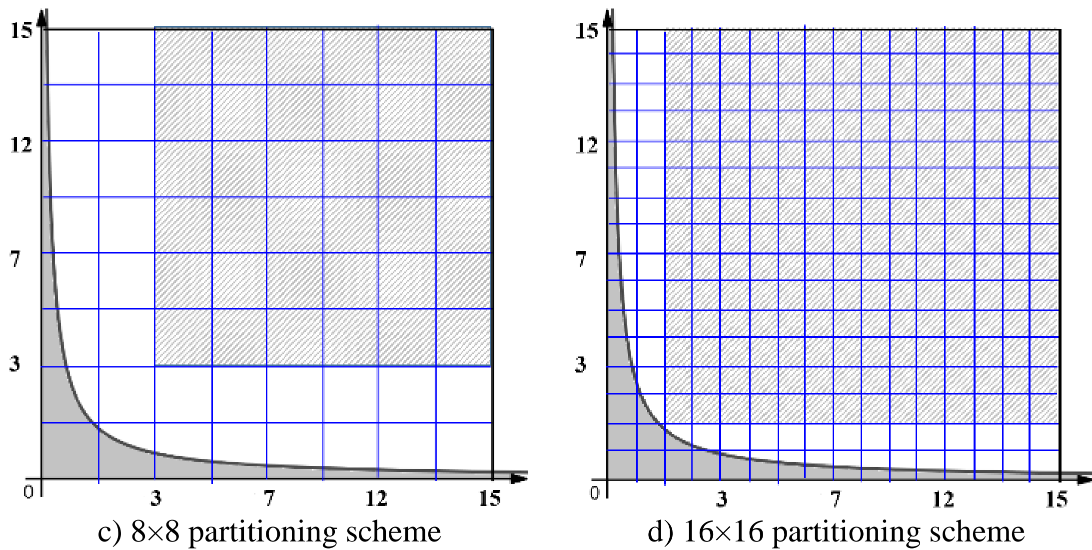
PRT [9, 14] works like random testing and generate test data randomly to execute a given path according to a uniform probability distribution over the program's input domain. This method at first derives the path constraints corresponding to a selected path using backward symbolic execution then separates each variable domain into k equal sub-domains. 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 sub-domains. Then the separated sub-domains check, one by one, for satisfiability using constraint propagation. The subdomains which could not satisfy the path constraints would be refuted. Obviously removing any portion of the invalid data, from the input domain, will decrease the number of rejects test data which deviate to execute the desired path. As a result a, uniform random sequence for the input domain can be built by generating first 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 is mentioned in section 3.4, with considering n input variables for a PUT the PRT method divides the input domain into k^n subdomains and checks their satisfiability against 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 CPU usage will be. This can be achieved through an iterative approach which successively partitions the input domain from a coarse scheme to a finer. Doing this speeds up the suggested algorithm, via 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 assess in the next iterations. For instance, consider again the motivating example and let 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, it would be decreased to 105. This is shown in Figure 3.



a) 2×2 partitioning scheme b) 4×4 partitioning scheme



c) 8×8 partitioning scheme d) 16×16 partitioning scheme

Figure 3. The valid subdomains in the various partitioning scheme

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 which need to be checked for satisfiability, in each partitioning scheme, is shown in table 1:

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

Partition scheme	#subdomains 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 then when 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. With 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. So in total 1+4+12+28+60=105 subdomains would be checked which is a remarkable result compared to the PRT method.

5.1. Overview of the IP-PRT algorithm

In the IP-PRT the input domain is divided into a regular grid of equally-sized cells. The grid cells are categorized into valid and invalid cells, according to their relative subdomains validity. The related subdomains of each valid cells are checked for satisfiability over the PC using constraint propagation. If adding a subdomain boundary to 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, then all the invalid cells are mapped into the new partition. This process can be repeated until the testing resources are exhausted or the number of iteration exceeds a threshold value.

5.2. Grid Coordinates Used in IP-PRT

In IP-PRT, the whole input domain is iteratively divided into equally sized subdomains as far as the size of the grid cells reaches a certain threshold value, specified by the tester. The coordinates of the sub-domains for each variable is 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 $0 \le x, y < M$ 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, then the

boundaries, of the grid cell, GridCells(i, j) could be computed by applying the following relation:

$$(i-1) \times C + 1 \leq x \leq i \times C \textit{ and } (j-1) \times C + 1 \leq y \leq j \times C$$

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

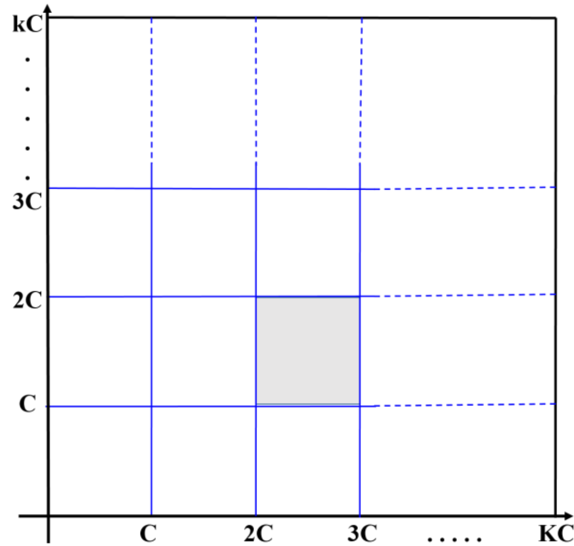


Figure 4. The coordinate of the cell (3,2)

5.3. 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 path constraints of the selected path, N the length of the expected random sequence and a threshold which determines the number of iterations. The threshold is taken to control the depth of the iterations.

To perform IP-PRT, at 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 partitions the input domain all over again. The proposed algorithm elaborates for a 2-dimensional square input domain with a size of M×M. Extension to input domains of higher dimensions is straightforward. A 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 sub-domains in each iteration of partitioning scheme. For example, k = 2 indicates a 2×2 partitioning grid.

Algorithm : IP-PRT

```
Input: x, y, PC, N, threshold
Output: t1, . . ., tN or ϕ (non-feasible 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 which
            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 Di,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 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. Firstly, the algorithm partitions the hypercuboid into a 2×2 grid and

then, each relative subdomain in the grid cells are checked for non-satisfiability. In the next iteration the division parameter k is changed to 2×k and the previous partitioning scheme is discarded and 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 to be small enough (the number of iteration exceeds the threshold value).

Secondly, a uniform random test data generator is built from the valid cells by picking up first a subdomain and then picking up a tuple inside this subdomain. If the selected tuple does not satisfy the path constraint 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. In the implemented prototype to solve the constraints the open source constraint solver, Choco is exploited [33].

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

In PRT and IP-PRT implementation after partitioning the input domain and refuting the invalid subdomains. At first, a subdomain is picked up from the remainder subdomains and then a tuple is selected inside this subdomains. Selecting the subdomains and tuples within the subdomains are random with uniform distribution probability. Furthermore, the PRT and IP-PRT have been evaluated with several distinct values of k. Considering that the achieved subdomains in both methods are identical so they have the same

1 situation to generate test data. Hence as expected, the number of generated test data in
2 both methods is nearly the same value. According to this in the experimental results, only
3 the number of test data which are generated by the IP-PRT method are shown.
4 To apply the methods over the benchmark at first a list containing of all the execution
5 path, for each benchmark, is created then, in a loop, the paths were chosen one by one
6 and given to the methods. Each method performs 20 independent runs over the given
7 path. For the benchmark which has a loop(s), we have selected a path such that each loop
8 iterates between 4 to 20 times. As a result, the average values of the 20 runs, over the all
9 paths, are considered.
10 All experiments were run on 64-bits, 2.10 GHz Intel® CoreTM i7 computer running
11 Microsoft Windows 7 with 8 GB memory.

12 **5.1. Programs to be tested**

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

22 **5.2. Experiments on the motivating example**

23 Table 2 reports on the obtained results for the predicate $x \times y <= 4$ of the motivating
24 example with 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 0.35ms and the number of rejects of the PRT and IP-PRT methods when k = 2 is 43 with CPU time 0.01ms and so on.

Table 2.The required CPU time and length of the test data generated for predicate **x×y<=4**

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

5.3. Experiments on the program *trityp* , *Middle,* and *remainder*

For the *trityp*, *Middle* and *remainder* programs, at first we extracted a list of the paths with their associated path condition that covers all the decisions of the program. For the program *remainder,* which has a loop, each path is selected in such a way that the loop iterates between 4 to 20 times. The domain of input variables are confined in the range [0…63] and compared the methods while generating random test data of increasing lengths from 10000 to 80000. The experimental results are given in Table 3, Table 4, and

1 Table 5 for the programs trityp, middle, and remainder, respectively. the PRT and IP-PRT
2 methods are compared with the three distinct values of the division parameter k. the
3 results demonstrate that the proposed method outperformed RT to minimize the amount
4 of the rejected test data with less CPU usage. Furthermore, the results show that the
5 proposed method improves the PRT method in terms of the CPU time consumption.

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

Requested		10000	20000	30000	40000	50000	60000	70000	80000
RT	Test data	293372	591298	874846	1163378	1451149	1745932	2039450	2329345
	Cpu time	35.4s	67.2s	89.9s	127.3s	151.7s	182.9s	216.9s	243.2s
k=4	Test data	144927	289512	432782	582120	731003	869305	1014498	1159410
	Cpu time (PRT)	2.9s	6.6s	13.2s	18.2s	23.1s	29.2s	32.8s	42.8s
	Cpu time (IP-PRT)	2.8s	5.3s	9.4s	12.5s	15.8s	19.9s	25.2s	34.1s
k=8	Test data	78125	156195	234405	312530	390604	468850	546775	625010
	Cpu time (PRT)	10.73s	25.2s	48.6s	67.9s	84.9s	109.1s	128.6s	158.8s
	Cpu time (IP-PRT)	10.3s	18.1s	32.3s	41.6s	56.9s	71.4s	95.5s	116.8s
k=16	Test data	39835	79581	119621	159303	199265	239143	278784	318715
	Cpu time (PRT)	17.6s	41.3s	80.1s	111.9s	139.2s	180.4s	218.4s	258.9s
	Cpu time (IP-PRT)	9.6s	22.7s	45.8s	62.7s	87.4s	112.5s	151.2s	194.3s

6

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

Requested		10000	20000	30000	40000	50000	60000	70000	80000
RT	Test data	31706	62831	94086	125067	157154	188811	220063	252188
	Cpu time	4.2s	8.0s	11.6s	15.7s	19.3s	23.9s	28.1s	33.7s
k=4	Test data	21525	43071	64486	86052	107577	129002	150638	172103
	Cpu time (PRT)	1.2s	2.1s	4.1s	4.9s	5.6s	6.3s	7.2s	7.9s
	Cpu time (IP-PRT)	1.0s	2.1s	3.7s	4.5s	5.2s	6.1s	6.9s	7.1s
k=8	Test data	14275	28450	42715	56988	71225	85507	99795	114009
	Cpu time (PRT)	2.3s	4.5s	6.1s	7.9s	9.3s	10.1s	11.9s	12.8s
	Cpu time (IP-PRT)	1.6s	2.8s	4.8s	6.7s	7.9s	8.8s	10.1s	10.9s
k=16	Test data	12417	24784	37301	49618	62130	74392	86819	99286
	Cpu time (PRT)	3.5s	6.9s	11.1s	13.2s	15.1s	16.8s	17.9s	19.1s
	Cpu time (IP-PRT)	2.3s	4.1s	8.1s	10.8s	11.9s	12.8s	14.2s	15.6s

7

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

Requested		10000	20000	30000	40000	50000	60000	70000	80000
RT	Test data	3135816	6293698	9385870	12555939	15868593	18817339	22064590	25108365
	Cpu time	27.3s	58.9s	75.8s	98.0s	114.8s	154.6s	181.3s	213.7s
k=4	Test data	135107	261817	409109	491200	612908	721522	895463	972101
	Cpu time (PRT)	2.1s	5.9s	11.4s	13.1s	15.2s	17.0s	18.8s	20.3s
	Cpu time (IP-PRT)	2.1s	5.7s	10.9s	12.8s	14.3s	15.9s	17.1s	18.9s
k=8	Test data	65340	110234	189320	278901	312009	391023	442981	569012
	Cpu time (PRT)	4.5s	8.1s	11.9s	15.3s	18.0s	22.3s	24.5s	26.1s
	Cpu time (IP-PRT)	3.1s	6.4s	9.7s	12.7s	14.6s	17.1s	19.5s	21.5s
k=16	Test data	34781	71240	92194	105672	128923	192903	231982	290823
	Cpu time (PRT)	8.3s	15.7s	21.1s	29.6s	37.2s	53.8s	61.0s	69.9s
	Cpu time (IP-PRT)	5.1s	10.9s	16.8s	21.4s	26.1s	36.5s	42.4s	51.7s

1 The obtained results on the three programs, trityp, middle and remainder, have been
2 summarized and presented in Table 6. In this table, the percent of improvement, in respect
3 to CPU usage, which have been achieved by IP-PRT, compared to PRT, is represented.
4 The results reveal that IP-PRT outperforms PRT significantly in all cases. In particular, it
5 is found that higher improvement is achieved when the division parameter, k, is increased. For
6 instance, in the case of trityp program, by using IP-PRT, up to 15.1% of improvement
7 may be obtained when the value of k is 4 and the requested number of test data to be
8 generated is 20000. It is worth mentioning that, the amount of improvement increases to
9 about 45%, when the value of k is increased to 16. A similar observation could be made
10 in other cases. The main reason behind this improvement is that IR-PRT significantly
11 reduces the number of invocations of the constraint propagation algorithm.

Table 6. the summarized results

Programs		Requested							
		10000	20000	30000	40000	50000	60000	70000	80000
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%

5.4. Discussion

In this section, we provide some discussion on the negative effects of eliminating input subdomains and the ability of IP-PRT in dealing with PUTs having large input domain spaces. In fact, determining the validity of a subdomain in respect to a PC is dependent on the constraint propagation algorithm. So, similar to other related works, like [9, 14, 26], a valid (or invalid) sub-domain 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 the corner cases in the proposed method. Please note that other related works, like PRT, employing constraint propagation algorithm, also suffer from this issue that may increase the number of invalid generated test data. Similar to PRT, our proposed algorithm is semi-correct, meaning that when it terminates, it is guaranteed to provide the correct 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 to be selected and, in this 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 to activate 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. So, 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 two-dimensional cuboid. Therefore, our proposed method can handle large domain spaces containing many input variables by iteratively partitioning the hypercube to sub-hypercubes.

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 with 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 for 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 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 to build efficiently 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 which cannot satisfy the constraint. As a result, the random testing effectiveness can thus be remarkably

1 enhanced by the generated test data from the reduced input domain. We showed that the

2 proposed method outperforms traditional RT and PRT.

3 **References**

4 [1] Swain, S., & Mohapatra, D. P. (2014). Genetic algorithm-based approach for adequate test
5 data generation. In Intelligent Computing, Networking, and Informatics (pp. 453-462). Springer,
6 New Delhi.

7 [2] Sahin, Omur, and Bahriye Akay. Comparisons of metaheuristic algorithms and fitness
8 functions on software test data generation. APPL SOFT COMPUT 49 (2016): 1202-1214.

9 [3] Hamlet, D. (2006, July). When only random testing will do. In Proceedings of the 1st
10 international workshop on Random testing (pp. 1-9). ACM.

11 [4] Liu, Huai, and Tsong Yueh Chen. Randomized quasi-random testing. IEEE T COMPUT 65.6
12 (2016): 1896-1909.

13 [5] Groce, Alex, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude
14 to formal verification. in ICSE 2007: PROC INT CONF SOFTW , 2007

15 [6] Arcuri, A., Iqbal, M. Z., & Briand, L. (2010, July). Formal analysis of the effectiveness and
16 predictability of random testing. In Proceedings of the 19th international symposium on Software
17 testing and analysis (pp. 219-230). ACM.

18 [7] Chen, Tsong Yueh, et al. Adaptive random testing: The art of test case diversity. J SYST
19 SOFTWARE 83.1 (2010): 60-66.

20 [8] Godefroid, Patrice, Nils Klarlund, and Koushik Sen. DART: directed automated random
21 testing. ACM SIGPLAN NOTICES. Vol. 40. No. 6. ACM, 2005.

22 [9] Gotlieb, Arnaud, and Matthieu Petit. A uniform random test data generator for path testing. J
23 SYST SOFTWARE 83.12 (2010): 2618-2626.

24 [10] Dinges, P., & Agha, G. (2014, September). Targeted test input generation using symbolic-
25 concrete backward execution. In Proceedings of the 29th ACM/IEEE international conference on
26 Automated software engineering (pp. 31-36). ACM.

27 [11] Baldoni, Roberto, et al. A survey of symbolic execution techniques. arXiv preprint
28 arXiv:1610.00502 (2016).

29 [12] Braione, P., Denaro, G., Mattavelli, A., & Pezzè, M. (2017, July). Combining symbolic
30 execution and search-based testing for programs with complex heap inputs. In Proceedings of the
31 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 90-101).
32 ACM.

33 [13] Qiu, R., Păsăreanu, C. S., & Khurshid, S. (2016, October). Certified Symbolic Execution. In
34 International Symposium on Automated Technology for Verification and Analysis (pp. 495-511).
35 Springer, Cham.

36 [14] Gotlieb, A., & Petit, M. (2006, July). Path-oriented random testing. In Proceedings of the 1st
37 international workshop on Random testing (pp. 28-35). ACM.

38 [15] McMinn, Phil. Search-based software test data generation: a survey. SOFTW TEST VERIF
39 BEH 14.2 (2004): 105-156.

[16] Harman, M., & McMinn, P. (2007, July). 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 (pp. 73-83). ACM.

[17] Sahin, Omur, and Bahriye Akay. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. APPL SOFT COMPUT 49 (2016): 1202-1214.

[18] Feyzi, F., & Parsa, S. (2018). Bayes-TDG: Effective Test Data Generation Using Bayesian Belief Network: Towards Failure-Detection Effectiveness and Maximum Coverage. IET SOFTW. DOI: 10.1049/iet-sen.2017.0112.

[19] Chow, C., Chen, T. Y., & Tse, T. H. (2013, July). The art of divide and conquer: an innovative approach to improving the efficiency of adaptive random testing. In Quality Software (QSIC), 2013 13th International Conference on (pp. 268-275). IEEE.

[20] Chen, Tsong Yueh, De Hao Huang, and Zhi Quan Zhou. On adaptive random testing through iterative partitioning. J INF SCI ENG 27.4 (2011): 1449-1472.

[21] Chan, K. P., Chen, T. Y., & Towey, D. (2003, June). Normalized restricted random testing. In International Conference on Reliable Software Technologies (pp. 368-381). Springer, Berlin, Heidelberg.

[22] Chen, Tsong Yueh, et al. Mirror adaptive random testing. INFORM SOFTWARE TECH 46.15 (2004): 1001-1010.

[23] Chen, T. Y., Merkel, R., Wong, P. K., & Eddy, G. (2004, September). Adaptive random testing through dynamic partitioning. In Quality Software, 2004. QSIC 2004. Proceedings. Fourth International Conference on (pp. 79-86). IEEE.

[24] Nikravan, E., Feyzi, F., & Parsa, S. (2015, November). Enhancing path-oriented test data generation using adaptive random testing techniques. In Knowledge-Based Engineering and Innovation (KBEI), 2015 2nd International Conference on (pp. 510-513). IEEE.

[25] Sabor, K. K., & Thiel, S. (2015, May). Adaptive random testing by static partitioning. In Proceedings of the 10th International Workshop on Automation of Software Test (pp. 28-32). IEEE Press.

[26] Offutt, A. Jefferson, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation. SOFTWARE PRACT EXPER 29.2 (1999): 167-93.