A Uniform Random Test Data Generator for Path Testing

Arnaud Gotlieb, Matthieu Petit

A Uniform Random Test Data Generator for Path Testing [1]

Arnaud Gotlieb and Matthieu Petit

INRIA Rennes - Bretagne Atlantique Research Centre
35042 Rennes Cedex, France
{Arnaud.Gotlieb,Matthieu.Petit}@irisa.fr
Tel: +33 (0) 2 99 84 75 76 – Fax: +33 (0) 2 99 84 71 71

Abstract

Path-oriented Random Testing (PRT) aims at generating a uniformly spread out sequence of random test data that execute a single control flow path within a program. The main challenge of PRT lies in its ability to build efficiently such a test suite in order to minimize the number of rejects (test data that execute another control flow path). We address this problem with an original divide-and-conquer approach based on constraint reasoning over finite domains, a well-recognized Constraint Programming technique. Our approach first derives path conditions by using backward symbolic execution and computes a tight over-approximation of their associated subdomain by using constraint propagation and constraint refutation. Second, a uniform random test data generator is extracted from this approximated subdomain. We implemented this approach and got experimental results that show the practical benefits of PRT based on constraint reasoning. On average, we got a two-order magnitude CPU time improvement over standard Random Testing on a set of paths extracted from classical benchmark programs.

Key words: Random testing, Path Testing, Constraint reasoning

1 Introduction

Path testing is one of the most popular white-box testing techniques. It was introduced more than thirty years ago by Howden [18] and has continuously been developed since then. It consists in selecting some paths within a program, finding input test data that activate these paths and checking the results of the executed paths against an oracle. Associated to each selected path, there is a subdomain of

[1] This paper is an extended version of Ref.[15]

the input domain that is considered as covered when one of its points is selected and submitted to the program. The property saying that each point of the subdomain is interchangeable has been called "reliability" by Goodenough and Gerhart [13] while Hamlet and Taylor called it "homogeneity" in the context of Partition testing [16]. The main principle that underlies path testing says that testing each selected path with a single point from a homogeneous subdomain suffices to get confidence in the program path correctness. Though this is a reasonable assumption, Hamlet and Taylor also explained that when this strategy fails, "it is technically because a subdomain lacked homogeneity" and suggested that a uniform distribution across each subdomain would be more appropriate as we cannot evaluate homogeneity a priori. In fact, test data that cause the same path to be executed do not have the same failure-revealing capability and sampling over the associated subdomain would increase the probability to select a failure-causing input. In [14], we introduced *"Path-oriented Random Testing (PRT)"* as a new technique to perform Random Testing at the path level. The idea behind PRT is to apply the nice principle of uniform selection, to the selection of test data that all activate the same path. The advantages of such an approach are the following: it increases the chance of generating a failure-causing input for a given path by giving the same probability to each input from the path subdomain to be selected; it introduces randomness and then objectivity in the test data generation process of path testing; it allows the random testing process to focus on specific paths of the program that are more likely to contain faults. However, there is also a main drawback behind PRT. It requires building a uniform random test data generator for a given path which is a hard problem. As the tester usually ignores the exact subdomain associated to a given path, it cannot easily define a random generator for this subdomain; one is resorted to generate test data from the entire input domain. Test data that execute the selected path are then kept while test data that execute another path are simply rejected. Thus, the challenging problem in PRT consists in building efficiently a *"uniform random test data generator" (URTG)* by minimizing the number of rejects within the generated random sequence.

In this paper, we address this problem by using constraint reasoning over finite domains [17]. We propose an original divide-and-conquer approach that exploits constraint propagation and constraint refutation over finite domains to build an over-approximation of the input subdomain corresponding to a given path. By reasoning on the constraints of path condition (i.e. symbolic constraints on input variables that correspond to a given path), we remove parts of the input domain that are inconsistent with these constraints. The over-approximation should be as tight as possible in order to minimize the rejects during the test data generation. The shape of the over-approximation should also have the property of permitting easily to build an URTG. Though our divide-and-conquer algorithm is based on complex constraint manipulation, we show that the overhead introduced by constraint propagation and refutation can be justified by the gain it offers. In addition, our approach is able to detect some non-feasible paths that cannot be identified with other Random Testing approaches such as adaptive RT [5] or feedback-directed RT [21]. PRT based

```
ush foo(ush x, ush y) {
1.  if (x =< 100  &&   y =< 100) {
2.      if (y > x + 50)
3.          ...
4.      if (x * y < 60)
5.          ...
```

Figure 1. Program foo

on constraint reasoning was implemented using the `clp(fd)` finite domains constraint solver of SICStus Prolog and was evaluated on several C programs. These experiments show that PRT based on constraint reasoning outperforms Random Testing for the uniform activation of a single path. In particular, we got a two-order magnitude CPU time improvement in favor of PRT on the longest path (including 18 function calls) of a C implementation of the Traffic Collision Avoidance System.

Outline of the paper. In section 2 we give an overview of PRT based on constraint reasoning on a simple but illuminating example. In section 3 we present some background on symbolic execution and random testing while explanations on how tuning usual Constraint Programming techniques to improve PRT are given in section 4. We present the divide-and-conquer algorithm to perform PRT in section 5 and section 6 contains the experimental results obtained with our implementation. In section 6, we also discuss related work. Finally, we conclude and draw some perspectives in section 7.

2 Motivating example

Consider the C program of Fig.1 and the problem of building a URTG for path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4{\rightarrow}5$. By looking at the decisions of the program, we can see that x and y must range in $0..100$ [2]. But, the other decisions cannot be tackled so easily. By using a URTG that independently picks up pairs (x_i, y_i) in $0..100 \times 0..100$ and rejects the pairs (x_j, y_j) that do not satisfy the constraints $y_j > x_j + 50 \wedge x_j * y_j < 60$ (rejection method [9]), we get a URTG that solves the problem. However, this approach is highly expensive as it will reject a lot of randomly generated pairs. In fact, by manually analyzing the program, we can see that the average probability of rejecting a pair is not far from $\frac{99}{100}$ with this approach. Indeed, activating the path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4{\rightarrow}5$ has a very low probability as only 58 input points over 10201 satisfy the constraints. In contrast, by using constraint propagation and constraint refutation, we can minimize this probability and reduce the length of the generated test suite. By using constraint propagation over finite domains, we get immediately that any solution pair (x, y) must range over the rectangle $D_1 = (x \in 0..1, y \in 51..100)$

[2] Let's suppose that **ush** stands for unsigned short integers

which is a correct [3] and tight over-approximation of the solutions of the problem. Building a random test data generator for D_1 is easy as we can still select x, y independently. This would not have been true if D_1 had the shape of a triangle, for example. Technically, one says that D_1 is an *hypercuboid*. In addition, by combining domain bisection and constraint refutation, we can get an even tighter over-approximation. D_1 can be fairly divided into 4 subdomains: $(x = 0, y \in 51..75), (x = 1, y \in 51..75), (x = 0, y \in 76..100), (x = 1, y \in 76..100)$. This division is fair as each subdomain has exactly the same number of two-dimensional points. Thanks to constraint refutation, the fourth subdomain can be safely removed from the domain for which we want to build a URTG. Indeed, constraint propagation shows easily that there is no solution of the path conditions in this subdomain. As $D_2 = (x = 0, y \in 51..75) \cup (x = 0, y \in 76..100) \cup (x = 1, y \in 51..75)$ is the union of subdomains of same areas, we can still easily build a URTG. for D_2 by selecting y independently from x. In fact, we design our method by keeping this latter constraint in mind. Finally, by using this method, the average probability of rejecting a possible pair in D_2 is just around $\frac{22}{100}$ (58 input points over the 75 of D_2 satisfy both decisions).

3 Background

In this section, we recall how to derive the path conditions associated to a control flow path by using symbolic execution (Sec. 3.1) and the basic principles of Random Testing (Sec. 3.2).

3.1 Symbolic execution

3.1.1 Control Flow Graph

The Control Flow Graph (CFG) of a program is a connected oriented graph composed of a set of vertices, a set of edges and two distinguished nodes, e the unique entry node, and s the unique exit node. Each node represents a basic block and each edge represents a possible branching between two basic blocks. Programs with multiple exits can easily been tackled by adding an additional exit node. A path is a finite sequence of edge-connected nodes of the CFG which starts on e. As an example, the CFG of the C program `power` is given in Fig.2. This program computes x^y. Note that this program contains a *non-feasible path*: $(1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6)$.

[3] No solution is lost

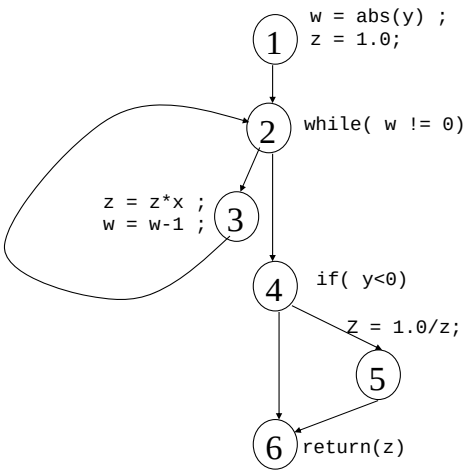

Figure 2. CFG of program power

3.1.2 Symbolic states

Symbolic execution works by computing symbolic states for a selected path. A *symbolic state* for path $e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k$ in program P is a triple $(e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k, \{(v,\phi_v)\}_{v\in Var(P)}, PC)$ where ϕ_v is a symbolic expression associated to the variable v and $PC = c_1 \wedge \ldots \wedge c_n$ is a set of constraints associated to path $e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k$, called the *path conditions*. $Var(P)$ denotes the set of variables in P. A symbolic expression is either a symbolic value (possibly **undef**) or a well parenthesized expression composed over symbolic values. In fact, when computing new symbolic expressions, each internal variable reference is replaced by its previously computed symbolic expression. In the program of Fig.2, the symbolic state of path $1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6$ can easily be obtained by inductively computing the following sequence of symbolic states:
$(1, \{(x,X),(y,Y),(w,\mathbf{undef}),(z,\mathbf{undef})\}, true)$
$(1{\rightarrow}2, \{(x,X),(y,Y),(w,abs(Y)),(z,1.0)\}, true)$
$(1{\rightarrow}2{\rightarrow}4, \{(x,X),(y,Y),(w,abs(Y)),(z,1.0)\}, abs(Y)=0)$
$(1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6, \{(x,X),(y,Y),(w,abs(Y)),(z,1.0)\}, Y<0 \wedge abs(Y)=0)$
where X (resp. Y) is the symbolic value of the input variable x (resp. y). Note that symbolic expressions and path conditions hold only over symbolic input values (except in the presence of floating-point computations [3]). Solving the path conditions yields either to show that the corresponding path is non-feasible or to find a test datum on which the path is executed. In the above example, the path conditions $Y < 0 \wedge abs(Y) = 0$ have no solution, meaning that the path $1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6$ is non-feasible.

3.1.3 Forward/backward analysis

Symbolic states are computed by induction on their path by a forward or a backward analysis [19]. Each statement of each node of the path is symbolically eval-

uated using an evaluation function which computes the symbolic states. Forward analysis follows the statements of the selected path in the same direction as that of actual program execution, whereas backward analysis uses the reverse direction. Backward analysis is usually preferred when one only wants to compute the path conditions, as it saves memory space. Indeed, backward analysis does not require the symbolic expressions to be stored when computing the path conditions. The idea is just to replace local references by symbolic expressions within the path conditions. We illustrate this point on the backward symbolic execution of path $1{\rightarrow}2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6$.
$(4{\rightarrow}6, \{(x,X),(y,Y)\}, Y \geq 0)$
$(2{\rightarrow}4{\rightarrow}6, \{(x,X),(y,Y)\}, w = 0 \wedge Y \geq 0$
$(2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6, \{(x,X),(y,Y)\}, w \neq 0 \wedge w - 1 = 0 \wedge Y \geq 0)$
$(1{\rightarrow}2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6, \{(x,X),(y,Y)\}, abs(Y) \neq 0 \wedge abs(Y) - 1 = 0 \wedge Y \geq 0)$

3.2 Random Testing

Random Testing (RT) is the process of selecting test data at random according to a uniform probability distribution over the program's input domain. Although RT has traditionally been considered as a blind approach of program testing [20], the results of actual random testing experiments confirmed its effectiveness in revealing faults [8,16]. We believe that a key advantage of RT over other techniques is that it selects objectively the test data by ignoring the specification or the structure of the program under test. When the input domain of a program is the Cartesian product of some finite numeric domains, building a Uniform Random Test Data Generator (URTG) is trivial but when the input domain is formed of data structures or infinite domains, the task is more complex [2]. For the sake of simplicity, we shall confine ourselves to a simple input domain made of the Cartesian product of bounded intervals of integers. Extensions will be considered in the conclusion of the paper. In this section, we recall the principle of URTG (Sec.3.2.1) and explain why performing Random Testing over a hypercuboid is a simple task (Sec.3.2.2). We end this section by giving two invariance properties of RT on which our approach is based (Sec.3.2.3).

3.2.1 Uniform Random Test data Generation (URTG)

An RTG is Uniform when each point of the input domain of a program has the same probability to be chosen. However, it is well known that uniformity can only be approximated on deterministic machines [9]. Most of the time, pseudo-random numbers generators make use of linear congruent rules such as $x_n = (a_1 x_{n-1} + a_2 x_{n-2} + ...)\ mod\ m$ to generate numbers. Thus, generating a n^{th} number is not independent of previous generations. Nevertheless, these pseudo-random number

generators behave well in practice (not far from uniformity) and so they suffice for our purpose. The design of such generators is outside the scope of this paper and a complete and recent survey of this topic can be found in [9].

3.2.2 Random Testing over a hypercuboid

The input domain of the program under test is formed by the Cartesian product of bounded intervals of integers. Technically, such an input domain is called a hypercuboid, which is the n-dimensional extension of the 3-dimensional cuboid. Performing random testing based on a uniform distribution over a hypercuboid domain is simple as any of its points can be randomly chosen by selecting its coordinates independently. Let us assume a two-dimensional input space (x, y), then RT can be implemented by selecting x at random and then y at random, without paying attention on the value obtained for x.

3.2.3 Two invariance properties of RT

Our PRT approach makes use of two fundamental invariance properties of uniform generators. The first property states that a uniform random generator for a given domain D can also serve as a uniform generator for any of the subdomains of D. More formally:

Property 1 (First invariance property) *Let S be a sequence of uniformly distributed tuples of values for a domain D, then for any subset D' of D, it is always possible to extract from S a sequence S' of uniformly distributed tuples for D'.*

Proof: *Let $S = \{x_1, .., x_N\}$ be a set of N points uniformly distributed over D. Then, the probability to draw x_i from D is the same for each i and if $S' = \{x_{t_1}, .., x_{t_M}\}$ is the set of M points that belong to both S and D', then S' is also uniformly distributed over D'*

Extracting such a sequence from S can be done simply by rejecting the tuples that do not belong to D'. The remaining sequence S' is still uniformly spread out over D' as D' is a subset of D. Of course, the smaller D' w.r.t. D, the larger the uniform sequence for D must be.

The second property states that a URTG can be built in a hierarchical manner:

Property 2 (Second invariance property) *Let D be a domain of N tuples, let K be a divisor of N and $D_1, \ldots, D_K$ be a partition of D such that each D_i possesses the same number of tuples, then a uniform random sequence for D can be built by generating first a uniform random sequence over $D_1, \ldots, D_K$, and then picking up a single tuple in each D_i, at random.*

Proof: *Each tuple of D has the probability $1/N$ to be drawn. The probability to draw D_i from $D_1, \ldots, D_K$ is $1/K$ and as $D_1, \ldots, D_K$ is an equi-partition of D, then each D_i possesses N/K tuples. Hence, each tuple resulting from the proposed process has the probability $1/K * 1/(N/K) = 1/N$ to be drawn.*

The important point here is that all the domains D_i have the same number of tuples. Whenever K is given and N cannot be divided by K, then it is possible to consider instead the smallest integer greater than N that can be divided by K. This remark is necessary in our context, as explained below.

4 Constraint Reasoning in PRT

Path-oriented Random Testing aims at finding a test suite that uniformly exercises a selected control flow path. We propose using constraint reasoning to build efficiently such a test suite. Constraint reasoning usually involves two interleaved processes in order to get a solution of a constraint system: constraint propagation and variable labeling. Constraint propagation prunes the variation domain of variables by eliminating inconsistent values while labeling tries to infer solutions by elaborating hypothesis and refuting subdomains. The key point of our approach is to employ constraint propagation (Sec.4.1) to find a hypercuboid that over-approximate the solution set of the path conditions, and to exploit constraint refutation (Sec.4.2) to remove spurious subdomains. We now turn on the description of these processes.

4.1 Constraint propagation

The process. Constraint propagation introduces constraints from the path conditions into a propagation queue. Then, an iterative algorithm manages each constraint one by one into this queue by filtering the domains of variables of their inconsistent values. When the variation domain of variables is large, filtering algorithms consider usually only the bounds of the domains for efficiency reasons: a domain $D = \{v_1, v_2, \ldots, v_{n-1}, v_n\}$ is approximated by the range $v_1..v_n$. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints where this variable appears, in order to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed. When selected in the propagation queue, each constraint is added into a constraint–store which memorizes all the considered constraints. The constraint–store is contradictory if the domain of at least one variable becomes empty. In this case the corresponding path is shown to be non-feasible.

Efficiency and completeness. When considering only the bounds of domains, constraint propagation is really very efficient as it runs in $O(m)$ where m denotes the number of constraints [17]. But, it is worth noticing that constraint propagation alone does not guarantee satisfiability. In fact, constraint propagation just tries to prune the variation domain and it does not test for satisfiability. For example, consider the following constraint system over finite domains: $x \in 1..100, y \in 1..100, z \in 1..100, x = y * z, x < z * y$. Here, constraint propagation does not perform any pruning on the domains, although the constraint system is clearly unsatisfiable. Hopefully, these situations are infrequent in practice and inconsistent subdomains can often be discarded. Note that computing the exact solution set of integer constraints over bounded domains is NP-hard [17].

Hypercuboids. Constraint propagation over finite domain variables computes *hypercuboids*: each variable of an n-dimensional space belongs to a range $Min..Max$ of values. Sometimes values can be removed from ranges such as in the presence of disequality constraints (e.g. $X \neq a$) but we will ignore such removals as our ultimate goal is to build a URTG and not to solve the constraints. In the example of Fig.1, constraint propagation permits to get the hypercuboid $D_1 = (x \in 0..1, y \in 51..100)$ where D_1 is an over-approximation of the solution set of the path conditions $x \in 0..100, y \in 0..100, y > x + 50 \wedge x * y < 60$.

4.2 Constraint refutation

Constraint refutation is the process of temporarily adding a constraint to a set of constraints and testing whether the resulting constraint system has no solution by using constraint propagation. If the resulting constraint system is unsatisfiable, the added constraint is shown to be contradictory with the rest of the constraints and then it is refuted. When constraint propagation does not yield to a contradiction, then nothing can be deduced as constraint propagation is not complete in general. Based on constraint addition/removal and propagation, this process is very efficient and it can be exploited in PRT to test domain intersection: let D be a subdomain defined by a set of constraints and C be a constraint, checking whether $D \cap C = \emptyset$ is true can be done by adding constraint C to D and test whether C is refuted or not. An example of such a refutation was given in the motivating example of the paper.

5 PRT based on constraint reasoning

In this section, we detail our divide-and-conquer algorithm to perform PRT based on constraint reasoning. Firstly, we detail how to fairly divide the hypercuboid resulting from constraint propagation (Sec. 5.1) and secondly, we explain how to

exploit constraint refutation to prune the subdomain associated to the path conditions (Sec.5.2). Finally, we show how our algorithm can exploit these processes to build an efficient URTG for PRT (Sec.5.3).

5.1 Dividing the hypercuboid

Applying constraint propagation on the path conditions results in a hypercuboid that is a correct approximation of the solution set of the path conditions. Using this approximation to define a URTG for PRT is possible but not optimal. We propose a new way of refining this hypercuboid in smaller subdomains. It is worth noticing that special attention must be paid to the way this hypercuboid is broken into subdomains in order to preserve the uniformity of the generator. Let k be a given parameter, called the *division parameter*, our method is based on the division of each variable domain into k subdomains of equal area. When the size of a domain variable cannot be divided by k, then we enlarge its domain until its size can be divided by k. By iterating this process over all the n input variables, we get a fair partition of the (augmented) hypercuboid, in k^n subdomains.

Consider the constraint set $\{y \geq 0, x \leq 14, x > y\}$ that corresponds to the triangle domain shown on the left in Fig.3. We will use this example in the rest of the paper to present our approach. Constraint propagation over these constraints gives

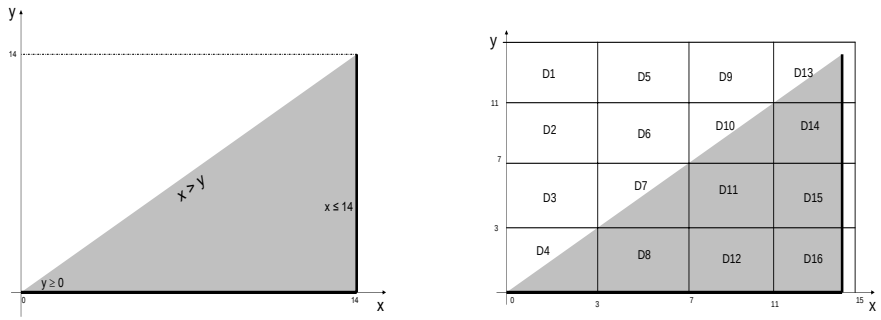

Figure 3. The triangle domain example

$D = (x \in 0..14, y \in 0..14)$. Consider a division parameter equal to 4. Then we have to divide the rectangle domain $x \in 0..14, y \in 0..14$ into $4^2 = 16$ subdomains of equal area. But, 4 does not divide [4] 15, therefore we enlarge the domain of x and the domain of y with a single value each. As a result, we get the 16 following subdomains: $D_1 = (x \in 0..3, y \in 12..15)$, $D_2 = (x \in 0..3, y \in 8..11)$,..,$D_{16} = (x \in 12..15, y \in 0..3)$ that form a partition of the (augmented) hypercuboid $D' = (x \in 0..15, y \in 0..15)$, as shown on the right in Fig.3.

[4] There are 15 values in each variable domain

5.2 Pruning the hypercuboid

As said previously, constraint refutation can be used to test efficiently domain intersection. Thus, we eliminate parts of the hypercuboid that are inconsistent with the path conditions. For the triangle domain, we can safely eliminate D_1, D_2, D_3, D_5, D_6 and D_9 by using constraint refutation. For example, $D_1 = (x \in 0..3, y \in 12..15)$ does not intersect the triangle domain, as $x > y$ does not hold in D_1.

As all the subdomains have the same area, we can still build an uniform test data generator for the resulting subdomain $D' = D_4 \cup D_7 \cup D_8 \cup D_{10} \cup ... \cup D_{16}$. On this example, we eliminated 6 subdomains over 16. By using the second invariance property, we get an easy way to draw uniformly test data. It suffices to draw at random a subdomain in D' and then to draw at random a value in this subdomain. This process is explained below. Thanks to the invariance properties, uniformity is preserved. Note that building a URTG from subdomains of distinct areas is also possible by sampling $D_1, .., D_k$ with probability proportional to the sizes of each D_i, but using uniform partitions is simpler.

Another advantage of constraint refutation is that it can detect non-feasible paths. Recall that non-feasible paths correspond to unsatisfiable constraint systems. Hence, when all the subdomains of the partition are shown to be inconsistent, then it means that the corresponding path is non-feasible. This contrasts with RT approaches such as Adaptive RT [5] or Feedback-directed RT [21] which cannot detect non-feasible paths. Note however that our approach can fail to detect some non-feasible paths due to the incompleteness of constraint propagation.

5.3 A divide-and-conquer algorithm

We present an algorithm that performs PRT based on constraint reasoning. The algorithm takes as inputs a set of variables along with their variation domain, PC a constraint set corresponding to the path conditions of the selected path, k the division parameter, and N the length of the expected random sequence. The algorithm returns a list of N uniformly distributed random tuples that all satisfy the path conditions. The list is empty when the corresponding path is detected as being non-feasible.

Firstly, the algorithm partitions the hypercuboid resulting from constraint propagation in k^n subdomains of equal area (`Divide` function). Then, each subdomain D_i in the partition is checked for unsatisfiability. This results in a list of subdomains $D'_1, \ldots, D'_p$ where $p \leq k^n$. Secondly, a URTG is built from this list by picking up first a subdomain and then picking up a tuple inside this subdomain. If the selected tuple does not satisfy the path conditions then it is simply rejected. This process is repeated until a sequence of N test data is generated. This 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. Indeed, in the second loop, N is decreased iff t satisfies PC, which can happen only if PC is satisfiable. In other words, if PC is unsatisfiable and if this has not been detected by constraint propagation ($p \geq 1$), then the algorithm will not terminate. Note that similar problems arise with random testing or path testing as nothing prevents a 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 on actual implementations. Note that any testing tools that execute programs should be equipped by such a time-out mechanism as nothing prevents a tested program to activate an endless path.

Algorithm 1: Path-oriented Random Testing

Input : $(x_1, .., x_n)$, PC, k, N
Output: $t_1, .., t_N$ or $\emptyset$ (non-feasible path)

$T := \emptyset$;
$(D_1, .., D_{k^n}) :=$ `Divide`$(\{x_1, .., x_n\}, k)$;
forall $D_i \in (D_1, \ldots, D_{k^n})$ **do**
  if D_i *is inconsistent w.r.t.* PC **then**
    remove D_i from $(D_1, .., D_{k^n})$;
  end
end
Let $D'_1, .., D'_p$ be the remaining list of domains;
if $p \geq 1$ **then**
  while $N > 0$ **do**
    Pick up uniformly D at random from $D'_1, .., D'_p$;
    Pick up uniformly t at random from D;
    if PC *is satisfied by* t **then**
      add t to T ;
      $N := N - 1$;
    end
  end
end
return T;

Our algorithm generates a sequence of uniformly spread out test data that activate a selected path of the program.

6 Experimental results

6.1 Our PRT and RT implementations

We implemented Path-oriented Random Testing (PRT) with constraint reasoning and compared it with Random Testing (RT). Both implementations take path conditions and domains as input parameters and provide a uniform random test suite as a result. To be fair, both implementations (PRT and RT) make use of the same random number generator (AS 183 algorithm from Wichmann and Hill [23]) and the same path condition evaluation scheme, under the form of Prolog constraints. The PRT implementation additionally exploits the SICStus Prolog library `clp(fd)` which offers constraint propagation and labeling heuristics. Both implementations (RT and PRT) and all our experiments are available online [5] . PRT also comes with an additional parameter k which is the division parameter defined in Sec. 5.1. When $k = 1$, the input domain is not divided and constraint refutation is applied only once on the entire domain. When $k > 1$, the constraint refutation part of our divide-and-conquer algorithm is applied on various subdomains of the input domain and permits sometimes to prune the size of the input domain.

6.2 Programs to be tested

We evaluated PRT w.r.t. RT on several programs: the `foo` program given in Fig.1, the `power` program given in Fig.2, the `trityp` program that is part of the Software Testing folklore and two real-world programs coming from the Civil and Military Aerospace domain. `Tcas` is extracted from the Traffic alert and Collision Avoidance System (TCAS) which is a computerized avionics device designed to reduce the danger of mid-air collisions between aircrafts. From the Software-artifact Infrastructure Repository (Do *et al.* 2005), it is possible to download a C component, called `tcas.c`, of a preliminary version of TCAS. This freely and publicly available component is (modestly) made up of 173 lines of C code. Finally, `ardeta` is a C program belonging to a large application designed to connect electronic equipment for military aircrafts on a test bench airplane. This program is made of 1305 lines of code. Both source codes contain nested conditionals, logical operators, bit-level operators, type definitions, macros and function calls but no floating-point variables, loops, pointers or dynamically allocated structures.

All the experimental results were computed on a 2.4GHz Intel Core Duo with 2GB of RAM.

[5] `www.irisa.fr/lande/gotlieb/resources/PRT`

6.3 *Experiments on the* `foo` *program*

Fig.4 reports on the results obtained for the path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4{\rightarrow}5$ in the `foo` program by regularly increasing the desired length of the random test suite. Fig.4 shows the number of test data generated with the PRT approach with four distinct values of the division parameter and traditional RT. For example, the first column shows that the number of rejects of the RT method is $9392 - 50 = 9342$ test data while it only evaluates to $88 - 50 = 38$ with PRT when $k = 1$, 15 with PRT when $k = 2$, and so on.

Requested	50	100	150	200	250	300	350	400	450	500
RT	9392	17800	26206	30859	42852	51184	61034	69690	77274	82669
PRT ($k = 1$)	88	180	280	351	432	495	589	718	821	840
PRT ($k = 2$)	65	132	187	263	311	387	461	534	586	644
PRT ($k = 3$)	54	119	186	254	294	352	412	460	531	576
PRT ($k = 4$)	53	114	159	216	280	325	381	457	502	556

Figure 4. Length of the test suite generated for `foo`

In PRT with $k = 2$, a single subdomain over 4 is shown to be unsatisfiable, whereas 5 subdomains over 9 with $k = 3$ and 11 subdomains over 16 are shown to be unsatisfiable with $k = 4$. When the requested length of the test suite is less than 500, the CPU time required to get a uniform random test suite (including unsatisfiability detection) is always less than 1sec. Next experience will study CPU time on longer test suites. The results of Fig.4 show that the probability of rejecting test data (those that do not satisfy path condition) decreases whenever the division parameter increases. For example, PRT with $k = 1$ requires 840 test data for producing 500 test data that cover the selected path while PRT with $k = 4$ only requires 556 test data for the same task.

Fig.5 shows the CPU time required to generate longer suites of random test data on the `foo` program. When the requested length is 35000, more than 10 Million

Requested	5000	10000	15000	20000	25000	30000	35000
RT	27.5s	55.5s	82.5s	111.1s	139.4s	158.3s	159.4s
PRT ($k = 1$)	0.08s	0.17s	0.34s	0.42s	0.61s	0.73s	0.98s
PRT ($k = 2$)	0.06s	0.19s	0.32s	0.53s	0.80s	1.06s	1.37s
PRT ($k = 3$)	0.06s	0.19s	0.32s	0.53s	0.77s	1.03s	1.34s
PRT ($k = 4$)	0.06s	0.17s	0.32s	0.53s	0.76s	1.03s	1.34s

Figure 5. CPU time required for generating test suite for `foo`

test cases are generated and evaluated for the RT implementation. The results show that PRT in any version is almost two order magnitude better than traditional RT on this example. One can object that traditional RT may be directly implemented in C and the satisfaction of path condition may be checked by instrumentation during

program execution. This would optimize the test data rejection process by saving the time required to keep track of contexts in our Prolog implementation, but this would have gained nothing but a constant factor on CPU time. Note however that the CPU time required by PRT with $k = 4$ becomes greater than the one required for PRT with $k = 1$ when the requested length is greater than 20000. This is due to the cost of constraint refutation on subdomains. Hence, the value of the division parameter k appears to be a good choice for balancing between the number of generated test data and the CPU time required to get a test suite for a given length.

6.4 Experiments on the `power` *program*

We selected path $(1{\rightarrow}2{\rightarrow}(3{\rightarrow}2)^{10000}{\rightarrow}4{\rightarrow}6)$ from the `power` program that iterates 10^4 times in the loop, in order to evaluate PRT when larger number of constraints are involved in the constraint propagation and refutation process. Input variables were constrained to belong to $0..50000$ and the constraint `R1#=X*R` that computes the power of X was replaced by `R1#=X*R mod 2` to avoid the computation of big integers. In this experiment, the constraint solver has to manage more than 20000 constraints.

The experimental results show that PRT with $k = 1$ generates a random sequence of 100 test data in 38.5sec of CPU time. Whenever $k = 2$, the time required is 38.8sec and 2 subdomains over 4 have been refuted. Whenever $k = 3$, the time required is 38.7sec and 6 subdomains over 9 are refuted and finally, when $k = 4$, 38.9sec are required and 12 subdomains over 16 are refuted. Hence, in all the cases, the CPU time required is similar. It is worth noticing that the constraint propagation step permitted to instantiate the second input parameter of `power` and then there was no reject at all. Hence, any randomly generated test data within the domain was accepted. The same request for the RT program never answers as the event $Y = 10000$ has a very low probability to happen. Note that each path has the same probability to be activated in `power` as each value of Y in $0..50000$ yields to activate a distinct path. This experience shows that PRT can scale up when numerous constraints are involved.

6.5 Experiments on the `trityp` *program*

For the `trityp` program, we manually extracted a list of 7 paths with their associated path condition, that covers all the decisions of the program. In this process, we did not pay attention to the feasibility of these paths, as many other structural testing tools. We confined the domain of input variables to be in $0..100$ and compared PRT and RT while generating random test suites of increasing lengths. The experimental results are given in Fig.6.

Although the results show that PRT outperforms RT, they are not as good as we expected. Firstly, it is well known that RT cannot easily cover the all decisions criterion on the `trityp` program as several events have very low probability to happen. For example, generating a sequence of three equal tuples (equilateral triangle) is a rare event. Of course, similar drawbacks exist with the PRT approach. A randomly chosen value is not propagated throughout the constraint network as this would bias the uniformity of the generator. Secondly, we expected PRT to detect non-feasible

Requested	10000	20000	30000	40000	50000	60000	70000	80000	90000	100000
RT	11.8s	23.6s	35.3s	47.1s	58.9s	70.7s	82.5s	94.5s	105.7s	117.9s
PRT ($k = 1$)	4.3s	8.8s	13.7s	18.8s	24.2s	30.0s	35.9s	42.3s	48.7s	55.9s
PRT ($k = 2$)	4.3s	8.9s	13.8s	18.9s	24.4s	30.1s	36.1s	42.6s	49.2s	55.9s
PRT ($k = 3$)	4.3s	8.9s	13.9s	18.9s	24.4s	30.4s	36.3s	42.5s	49.3s	55.9s
PRT ($k = 4$)	4.3s	9.0s	14.1s	19.0s	24.8s	30.7s	36.9s	43.1s	49.8s	56.7s

Figure 6. CPU time required for generating random test suite on program `trityp`

paths among the paths selected to cover all decisions. But finding inconsistent subdomains requires the division parameter k to be instantiated to 13. In this case, 469 subdomains are shown inconsistent over a total of 2197. Note that among the 7 paths, 4 are non-feasible. As the value of the division parameter $k = 13$ depends on the problem, we decided to avoid taking advantage of this knowledge and then we confined our experiments to small values of k. In theory, selecting greater values for k would yield to increase the deductions as many additional subdomains will be tested for satisfiability and possibly discarded. But the time required to check satisfiability will also increase accordingly. In practice, selecting small values for k (e.g. $kin1..4$) permits to maximize the gain by eliminating large subdomains while keeping an acceptable overhead.

6.6 Experiments on the `tcas` *program*

For the `tcas` program, we selected the longest path of the function `alt_sep_test`. This path contains 18 function calls and several complex logical decisions. The input space of the function `alt_sep_test` is made of 12 global 32-bits unsigned integer variables. We arbitrarily restricted each input variable to belong to $0..1000$ in order to avoid undesirable effects at the bounds of domains in both the RT and PRT implementations. Hence, the input domain is of cardinality 1001^{12}. The results we got for this program are given in Fig.7.

Our results on the `tcas` example merely show a two-order magnitude improvement of PRT with $k = 1$ over RT. This is explained well by the fact that activating the longest path of the program is difficult as it corresponds to a small subdomain of the input space. Constraint propagation permits to prune drastically the search space on this example. By analyzing the results, we found that 28672 subdomains

Requested	10	20	30	40	50	60	70	80	90	100
RT										
CPU time	58.6s	103.4s	191.4s	275.6s	298.7s	282.8s	482.9s	424.2s	525.6s	541.4s
Test data	185160	328874	609571	866125	949171	925341	1578769	1388161	1719640	1772755
PRT ($k = 1$)										
CPU time	0.7s	0.7s	1.0s	1.4s	1.4s	1.8s	2.3s	2.4s	2.7s	3.4s
Test data	154	179	225	320	343	483	601	624	721	864
PRT ($k = 2$)										
CPU time	92.0s	93.0s	93.6s	96.1s	93.9s	90.5s	93.2s	92.7s	93.1s	92.9s
Test data	30	88	133	221	236	278	329	377	523	447

Figure 7. CPU time required for generating random test suite on program `tcas`

over 65536 were eliminated when $k = 2$. So, using the constraint refutation process on this example is useful and means that a tighter over-approximation can be automatically found. However, the CPU time required to prune the refuted subdomains, even if it stays constant when the requested length of the test suite increases, penalizes PRT with $k = 2$.

Requested	10	20	30	40	50	60	70	80	90	100
RT										
CPU time	60.3s	105.6s	139.9s	185.9s	206.3s	328.5s	331.0s	372.6s	480.7s	491.1s
Test data	138509	242536	320570	425687	472805	744892	749479	841704	1093311	1114409
PRT ($k = 1$)										
CPU time	0.0s	0.0s	0.0s	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s	0.1s
Test data	27	81	161	180	203	237	239	355	470	410
PRT ($k = 2$)										
CPU time	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.0s	0.1s	0.1s
Test data	17	46	74	61	105	102	145	173	181	206

Figure 8. CPU time required for generating random test suite on program `ardeta`

The results obtained on program `ardeta` shown in Fig.8 are similar to those obtained for `tcas`, except that the time required to generate the requested test suites are similar when $k = 1$ and $k = 2$. In the second case, 2048 subdomains over 4096 are found to be inconsistent which corresponds to half of the entire input domain and the time required to find inconsistencies is small w.r.t. the CPU time required to generate the test suite. These results indicate that both constraint propagation and constraint refutation are useful and efficient in PRT on moderated-sized benchmarks. However, other experiments on larger benchmarks would be required to confirm these results.

6.7 Related work

PRT is a technique that improves path testing by building a URTG that activates a single control flow path. We are not aware of any other technique that addresses

the same problem in the context of software testing. However, in the context of hardware verification [7], the research work of Gogate and Dechter in [11] also aims at sampling the solutions of a constraint system uniformly at random. Their algorithm belongs to the class of Monte-Carlo algorithms that samples from the output of a generalized belief propagation algorithm which is a variation of what we called the *rejection method* in this paper. Nevertheless, their approach is dedicated to Constraint Satisfaction Problems where constraints are defined with tuples (e.g. if x and y belong to $1..2$, constraint $x \neq y$ is defined in extension as the tuples $\{(1,2),(2,1)\}$) and boolean satisfiability problems [12]. It seems uneasy to adapt these techniques to constraint systems extracted from path conditions as variables hold over large domains (e.g. 32-bits integer variables) and constraints are defined with formula instead of tuples. In addition, unlike the algorithm of Gogate and Dechter, our divide-and-conquer is non-intrusive, meaning that the constraint solver is used as a black-box.

Note that the idea of exploiting constraint reasoning in Random Testing is not new. Chan et al. proposed in [4] several implementations of the Center of Gravity constraint as a way to improve Adaptive RT [5]. However, unlike other Random Testing approaches, PRT exploits constraint propagation and refutation to get a uniform sequence of test data that activate a selected path. Thanks to its usage of constraint reasoning, PRT is able to show in some cases that the path conditions have no solution and that the corresponding path is non-feasible. This is outside the scope of advanced RT techniques such as adaptive RT [5] or feedback-directed RT [21].

There exist tools that perform automatic test data generation for path testing. In PathCrawler [24], Williams et al. propose a randomized algorithm that generates test suites to cover the k-paths [6] criterion by using symbolic execution and constraint propagation over finite domains. Godefroid, Klarlund and Sen independently followed a similar approach in the tools DART (Directed Automated Random Testing) [10] and CUTE [22]. They got very good experimental results on C programs extracted from real-world applications. Recently, the tool JPF-SE [1] was proposed in the context of software model checking to generate test data. This tool exploits various decision procedures to find test data for activating certain paths of Java programs. However, all these approaches generate a single test data for each considered path and their goal is to get the complete coverage of all the feasible paths of a program up to a certain limit. In [6], Collavizza and Rueher explored the capabilities of finite domain constraint solvers for testing Java programs. They showed that these solvers could be very efficient to generate a single test datum that satisfies the path conditions.

We believe that PRT could be used to complement these approaches by generating a uniform sequence of test data to activate each path that is selected. This would certainly improve the fault revealing capabilities of these techniques as each path

[6] Paths that iterate at most k times each loop of the program

would be more thoroughly exercised.

7 Conclusion

This paper introduced constraint reasoning in Path-oriented Random Testing, through the usage of constraint propagation and refutation over finite domains. We proposed a simple divide-and-conquer algorithm that permits to build efficiently a uniform sequence of test data exercising a selected path in the program under test. Although our approach was evaluated on a few benchmark programs only, we showed that Path-oriented Random Testing outperforms traditional RT on realistic examples. As discussed in the paper, we believe that Path-oriented Random Testing could be advantageously exploited in other path-oriented test data generation techniques to improve their fault-revealing capabilities. However, our approach is also currently limited to integer variables and dealing with programs that manipulate pointers and floating-point variables is indispensable to scale the approach up to realistic languages. This is challenging as it requires not only to solve constraints on these features but also to build uniform random test data generator for complex data structures, such as simple lists, circular lists, double-linked lists and so on.

Acknowledgments

Many thanks to Nicky Williams and the anonymous referees who provided us with helpful comments on an earlier draft of this paper.

References

[1] S. Anand, C. S. Pasareanu, and W. Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *Int. Conf. on Tools and Algo. for the Construction and Analysis of Systems (TACAS'07)*, April 2007.

[2] J. H. Andrews, S. Haldar, Y. Lei, and F. C. Hang. Tool support for randomized unit testing. In *1st ACM Int. Workshop on Random Testing (RT'06)*, Portland, Maine, July 2006.

[3] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.

[4] F. Chan, K. Chan, T. Chen, and S. M. Yiu. Adaptive random testing with cg constraint. In *Proc. of the 28th Int. Computer Software and Applications Conf. (COMPSAC'04)*, 2004.

[5] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *ASIAN'04*, pages 320–329, 2004.

[6] H. Collavizza and M. Rueher. Exploration of the capabilities of constraint programming for software verification. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06)*, pages 182–196, 2006.

[7] R. Dechter, K. Kask, E. Bin, and R. Emek. Generating random solutions for constraint satisfaction problems. In *Eighteenth national conference on Artificial intelligence*, pages 15–21, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.

[8] J. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, Jul. 1984.

[9] P. Ecuyer. *Random Number Generation*, chapter Chapter 2 of Handbook of Computational Statistics, pages 35–70. Springer-Verlag, 2004.

[10] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proc. of PLDI'05*, pages 213–223, 2005.

[11] V. Gogate and R. Dechter. A new algorithm for sampling csp solutions uniformly at random. In *Principles and Practice of Constraint Programming (CP'06)*, volume 4204 of *LNCS*, pages 711–715, 2006.

[12] V. Gogate and R. Dechter. Studies in solution sampling. In *Proc. of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI'08)*, pages 271–276, July 2008.

[13] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, 1(2):156–173, June 1975.

[14] A. Gotlieb and M. Petit. Path-oriented random testing. In *1st ACM Int. Workshop on Random Testing (RT'06)*, Portland, Maine, July 2006.

[15] A. Gotlieb and M. Petit. Constraint reasonning in path-oriented random testing. In *32nd Annual IEEE International Computer Software and Applications Conference (COMPSAC'08)*, Turku, Finland, Jul. 2008. Short paper, 4 pages.

[16] D. Hamlet and R. Taylor. Partition Testing Does Not Inspire Confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, Dec. 1990.

[17] P. Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37:139–164, 1998.

[18] W. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–214, September 1976.

[19] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications – Chapter 9 : L. Clarke, D. Richardson*. Prentice-Hall, 1981.

[20] G. J. Myers. *The Art of Software Testing*. John Wiley, New York, 1979.

[21] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. of the Int. Conf. on Software Engineering (ICSE'07)*, Minneapolis, 2007.

[22] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. of ESEC/FSE-13*, pages 263–272. ACM Press, 2005.

[23] B. A. Wichmann and I. D. Hill. Algorithm as 183: An efficient and portable pseudo-random number generator. *Applied Statistics*, 31:188–190, 1982.

[24] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *Proc. Dependable Computing - EDCC'05*, 2005.