Path-oriented random testing

Arnaud Gotlieb
IRISA - INRIA
Campus Beaulieu
35042 Rennes Cedex
Arnaud.Gotlieb@irisa.fr

Matthieu Petit
IRISA - INRIA
Campus Beaulieu
35042 Rennes Cedex
Matthieu.Petit@irisa.fr

ABSTRACT

Test campaigns usually require only a restricted subset of paths in a program to be thoroughly tested. As random testing (RT) offers interesting fault-detection capacities at low cost, we face the problem of building a sequence of random test data that execute only a subset of paths in a program. We address this problem with an original technique based on backward symbolic execution and constraint propagation to generate random test data based on an *uniform* distribution. Our approach derives path conditions and computes an over-approximation of their associated subdomain to find such a uniform sequence. The challenging problem consists in building *efficiently* a path-oriented random test data generator by minimizing the number of rejects within the generated random sequence. Our first experimental results, conducted over a few academic examples, clearly show a dramatic improvement of our approach over classical random testing.

1. INTRODUCTION

Random Testing (RT) is the process of selecting test data at random according to an uniform probability distribution over the program's input domain. Uniform means that every point of a domain has the same probability to be selected. Although RT has traditionally been considered as a blind approach of program testing [14], the results of actual random testing experiments confirmed its effectiveness in revealing faults [6, 9]. Among other advantages [11], one 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 [3]. For a long time, RT has been opposed to partition testing which aims at selecting one or more test data from each subdomain of a partition of the input domain [17]. As a typical example of partition testing, path testing requires to find a test suite so that every control flow path is traversed at least once. As every feasible[1] path corresponds to a subdomain of the input domain, path testing consists in selecting at least one test datum from each subdomain.

Test campaigns usually require only a subset of paths to be thoroughly tested as (exhaustive) path testing is most of the time impossible. In fact, as soon as the program contains a loop, the number of paths is potentially unbounded, and deciding whether a path is feasible or not is an undecideable problem in the general case [16]. Usual white-box testing approaches requires only a subset of paths to be selected to cover a given structural criterion such as all statements or all decisions [20]. Moreover, it is well known that some control flow paths are irrelevant for the computation of the function implemented by the program as they will never be activated during the operational life of the program. For example, consider a program where some robustness code has been added just to avoid its usage in unsuitable conditions. Hence, this leads us naturally to the idea of performing random testing by selecting first a subset of paths, in order to benefit from the advantages of random testing in a partition testing approach.

In this paper, we introduce an original technique to generate random test data based on a *uniform* distribution for only a subset of paths. Our approach, called Path-oriented Random Testing (PRT), derives the path conditions of the selected paths by using a backward symbolic execution [13] and computes an over-approximation of their associated subdomain by using constraint propagation [10, 12]. The challenging problem consists in building *efficiently* a uniform random test data generator by minimizing the number of rejects within the generated random sequence [15]. A reject is produced whenever the randomly generated test datum does not satisfy the path conditions. Our approach addresses this problem by using subtle constraint propagation and constraint refutation combined with random test data generation. We implemented our PRT approach by using the clp(fd) constraint library of SICStus Prolog [2] and get some first experimental results showing that PRT outperforms traditional RT. In addition, we provide an algorithm that can detect some non-feasible paths in the set of selected paths. This also shows a qualitative improvement over traditional RT.

Outline of the paper. Section 2 gives an overview of our approach on a motivating example. Section 3 recalls some background on symbolic execution while Section 4 details our constraint propagation approach to build an uniform test data generator for a subset of paths. An algorithm to perform PRT method is given and analysed. Section 5 reports on the experimental results we obtained with our implementation and Section 6 presents further work.

2. A MOTIVATING EXAMPLE

We start by giving an overview of the PRT approach over traditional RT on a motivating example. Consider the C program of Fig.1 and the problem of building an uniform test data generator for path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4{\rightarrow}5$. By looking at the decisions of the program, we

[1] A path is feasible iff it can be activated by some test data

```
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 (ush stands for unsigned short integers)

can see that x and y must range in $0..100$ to satisfy our objective but other decisions cannot be tackled so easily. By using a uniform random test data generator that independently pick up a value x_i in $0..100$ and a value y_i in $0..100$ and rejects the pairs (x_i, y_i) that do not satisfy the constraints $y_i > x_i + 50 \wedge x_i * y_i < 60$, we get a uniform random generator that solves our problem. However, this traditional RT approach is highly expensive as it requires rejecting a lot of randomly generated pairs. In fact, by manually analyzing the program, we can determine that the average probability of rejecting a possible pair is not far from $\frac{99}{100}$ with this RT approach. Indeed, executing path $1{\rightarrow}2{\rightarrow}3{\rightarrow}4{\rightarrow}5$ is an event which has a very low probability as only 58 input points over 101^2 satisfy the two decisions. In contrast, our PRT approach exploits subtle constraint propagation and constraint refutation to minimize this probability and then it reduces the length of the generated test suite. By using constraint propagation over finite domains on this example, we get immediately that any solution pair (x, y) must range over the rectangle $D_1 = (x \in 0..1, y \in 51..100)$ which a correct over-approximation of the 58 solutions. Building a random test data generator for D_1 is an easy task as we can select x, y independently. This would not have been true if D_1 had the shape of a triangle, for example. However, by combining domain bisection and constraint refutation, we can get an even better approximation: $D_2 = (x = 0, y \in 51..101) \cup (x = 1, y \in 51..67)$ where the subdomain $(x = 1, y \in 68..100)$ has been refuted while just a single spurious pair was added ($x = 0, y = 101$). Note also that one can still easily build a random test data generator for D_2 by selecting y independently from x as D_2 can be divided in an union of rectangles of the same area. In fact, we design our PRT method while keeping this latter constraint in mind. Finally, we can determine that the average probability of rejecting a possible pair which is just around $\frac{14}{100}$ (58 input points over the 68 of D_2 satisfy the two decisions). By using the PRT approach, the size of the randomly generated test suite is dramatically decreased while the overhead introduced by constraint propagation and constraint refutation remains tractable [10].

3. BACKWARD SYMBOLIC EXECUTION

In this section, we explain how to derive path conditions associated to a subset of selected paths in the program under test. This process is based on backward symbolic execution that was first formalized by Clarke and Richardson in [13]. This technique is based on the selection of paths of the control flow graph and the computation of symbolic states.

3.1 Control flow graph

The control flow graph of a program P 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. A path of P is a finite sequence of edge-connected nodes of the control flow graph which starts on e. As an example, consider the program `power.c` given in Fig.2 along with its CFG. This program computes x^y. Note that

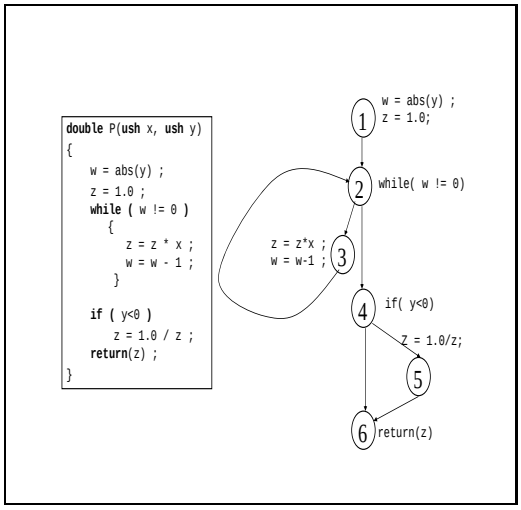

Figure 2: Control flow graph of program `power.c`

this program contains a *non-feasible path* ($1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6$), as in many imperative programs [19].

3.2 Symbolic states

Symbolic execution works by computing symbolic states for a given path. A *symbolic state* for path $e{\rightarrow}n_1{\rightarrow}\ldots{\rightarrow}n_k$ in 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 $PC(e{\rightarrow}n_1{\rightarrow}\ldots) = c_1 \wedge \ldots \wedge c_n$ is a set of constraints, called *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 a new symbolic expression, 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 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 [1]).

Solving the path conditions yields either to demonstrate that a given path is non-feasible or to find a test datum on which the selected path is executed. In the previous example, it is trivial to see that the (non-linear) path conditions $Y < 0 \wedge abs(Y) = 0$ have no solution, showing that the path $1{\rightarrow}2{\rightarrow}4{\rightarrow}5{\rightarrow}6$ is non-feasible. Unfortunately, automatically solving a set of non-linear constraints over unbounded integers is a classical undecideable problem. Hence, we have to perform some kind of approximation if one wants to automate this process.

3.3 Forward/backward analysis

Symbolic states are computed by induction on their path by a forward or a backward analysis [13]. Each statement of each node of the path is symbolically evaluated 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 $1{\rightarrow}2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6$.
$(4{\rightarrow}6, \{(x, X), (y, Y)\}, Y \geq 0)$
$(2{\rightarrow}4{\rightarrow}6, \{(x, X), (y, Y)\}, w = 0 \wedge Y \geq 0$
$(2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6, \{(x, X), (y, Y)\}, w \neq 0 \wedge w - 1 = 0 \wedge Y \geq 0)$
$(1{\rightarrow}2{\rightarrow}3{\rightarrow}2{\rightarrow}4{\rightarrow}6, \{(x, X), (y, Y)\}, abs(Y) \neq 0 \wedge abs(Y) - 1 = 0 \wedge Y \geq 0)$

3.4 Symbolic execution for a set of paths

Path conditions for a set of paths can be derived just by considering the disjunction of conditions computed for each path. Let $\{p_1, .., p_n\}$ be a set of paths, then $PC(p_1) \vee PC(p_2) \vee \ldots \vee PC(p_n)$ is the path conditions associated to $\{p_1, .., p_n\}$. Note that these path conditions are in Normal Disjunctive Form (OR of AND). If the input domain is the Cartesian product of integer variable domains, then symbolic manipulations can be performed on these path conditions to simplify the resolution process. In this paper, we will not go into more details on this process as it is outside the scope of the paper.

4. PATH-ORIENTED RANDOM TESTING BASED ON CONSTRAINT PROPAGATION

In this section, the constraint propagation process is first explained and then its usage to build a domain on which uniform path-oriented random test data generation can easily be performed is described. In the rest of the paper, we consider that each input variable belongs to a type-dependant finite integer domain and there is only a finite number of input variables. These two hypotheses help us to focus on the technical problems of path-oriented random testing. Extensions are discussed in conclusion. This contrasts with other RT approaches where the input domain is considered to be formed of bits sequence, whatever be the types of variable [11]. We also suppose that path conditions for a subset of paths have been computed by backward symbolic execution. Hence, these constraints are given under Normal Disjunctive Form. We do not make any other assumptions as our approach is generic for any finite domains constraint system. We discuss of pseudo-random number generators just to make clear what uniformity means on computers.

4.1 Constraint propagation

The goal of constraint propagation is to shrink the finite variation domain of each variable in order to get a sound over-approximation of the solutions of a set of constraints. This process however does not guarantee the existence of solutions as it can only eliminate inconsistent values.

During constraint propagation, constraints from the path conditions are incrementally introduced into a propagation queue. An iterative algorithm manages each constraint one by one into this queue by filtering the domains of variables of their inconsistent values. Filtering algorithms consider usually only the bounds of the domains for efficiency reasons. When the domain of a variable is pruned then the algorithm reintroduces in the queue all the constraints where this variable appears (awaked constraints) to propagate this information. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fixpoint). When selected in the propagation queue, each constraint is added into a *constraint–store* which memorizes so 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. Using constraint propagation in software testing is not a new idea [4, 8], but, according to our knowledge, it has not yet been used in random testing.

As shown in the example of Fig.1, constraint propagation permits to get $D_1 = (x \in 0..1, y \in 51..100)$ as an over-approximation of the solution set of: $X \in 0..100, Y \in 0..100, Y > X + 50 \wedge X * Y < 60$.

Constraint propagation over finite domain variables computes only hypercubes: 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, in our PRT approach, we will ignore such removals as only ranges are suitable for our purpose. So, we will consider that a finite ordered domain $v_1, v_2, \ldots, v_{n-1}, v_n$ is always represented and approximated by the range $v_1..v_n$. The interesting point is that constraint propagation over ranges is really very efficient. In fact this process is just linear on the number of constraints.

4.2 Random test data generation

A random test data generator is said to be 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 (such as usual computers) and only pseudo-random numbers generators can be employed. 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. So, generating a n^{th} number is an event not totally independent of the previous generation and the probability of a given number to be selected is not equivalent for each number. By this, we lost strict uniformity. However, these generators appear as being very good in practice (not far from uniformity). Then, they are suitable 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 [7].

4.3 Random Testing over an hypercube

An hypercube is the n-dimensional extension of the 3-dimensional cube. Performing random testing based on an uniform distribution over an hypercube domain is trivial as any of its points can be randomly selected by selecting its coordinates independently. Let us assume a two-dimensional input space (x, y), then RT can be implemented just by selecting x at random and then y at random, without paying attention on the value obtained for x. These two events are independents. This property is true for some shapes but not for all. Consider for example a triangle domain, such as the one shown in figure 3. Suppose this triangle domain is limited by the three lines $y = 0, x = 14, y = x$. Let x_1, x_2 be two randomly selected integers such as $x_2 > x_1$ and y_1, y_2 two randomly selected integers such as y_1 belongs to $1..x_1$ and y_2 belongs to $1..x_2$, then the probability of the event (x_1, y_1) *is selected* is equal to $1/x_1$ whereas the probability of the event (x_2, y_2) *is selected* is equal to

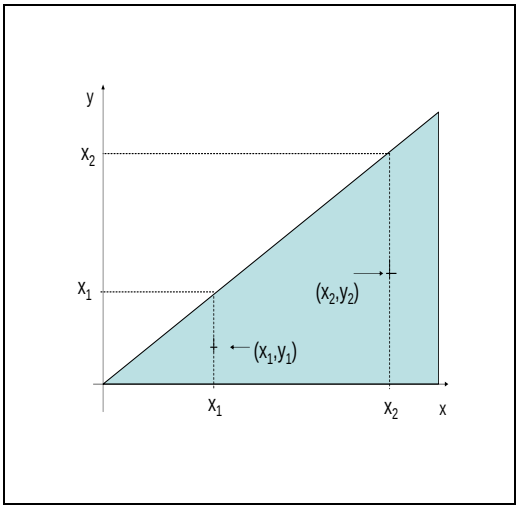

Figure 3: The triangle domain

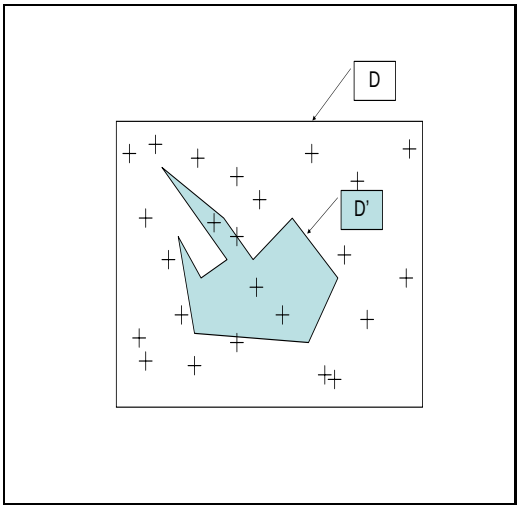

Figure 4: The first invariance principle

$1/x_2$. As $x_1 \neq x_2$, we have lost uniformity. In fact, these two events are not independents over the triangle domain whereas they always are over an hypercube domain.

4.4 Two invariance principles of RT

Our PRT approach makes use of two well-known fundamental invariance principles of uniform random number generators. The first principle just states that any uniform random generator for a given domain D can also be employed as an uniform random generator for any of the subdomains of D. More formally:

PROPERTY 1 (FIRST INVARIANCE PRINCIPLE). *Let S be a sequence of uniformly distributed tuples of values for a domain D, then for any subset D' of D, it is possible to extract from S a sequence S' of uniformly distributed tuples for D' by rejecting the tuples of S that do not belong to D'. The remaining sequence S' is still uniformly distributed over D'.*

This first invariance principle is illustrated in Fig.4. To obtain a uniform sequence of values for D', it suffices to examine each of the tuples of the sequence for D and to reject those tuples that do not belong to D'. Of course, the smaller D' is w.r.t. D, the larger the uniform sequence for D must be. By looking back at the triangle example, the first invariance principle just says that we get a random test data generator for the triangle domain by taking a uniform test data generator for the encompassing rectangle and rejecting the pairs that do not belong to the triangle.

The second principle is more subtle as it states that a random test data generator can be built in a hierarchical manner:

PROPERTY 2 (SECOND INVARIANCE PRINCIPLE). *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 an uniform random sequence for D can be built by building first a uniform random sequence over $D_1, \ldots, D_k$, and then picking up a single tuple in each D_i at random. The resulting sequence of tuples is still uniformly distributed over D.*

The important point here is that all the domains D_i have the same number of tuples. Fig. 5 illustrates the second invariance principle over the triangle domain. To build an uniform sequence over the triangle domain, we can first break its encompassing rectangle into $D1, \ldots, D15$ equivalent subdomains then build a random sequence of these subdomains and finally select a point in each subdomain.

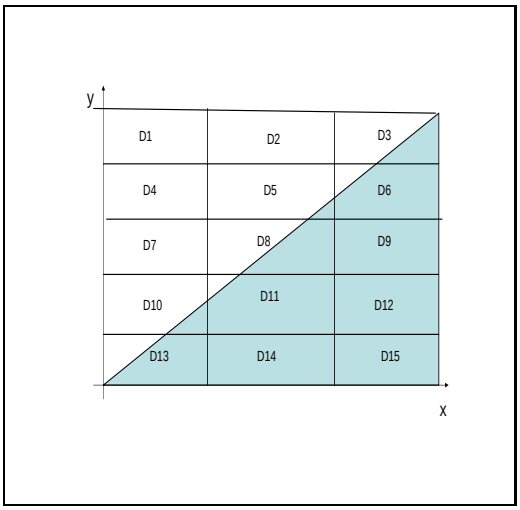

Figure 5: The second invariance principle

By combining the two principles, we can optimize the process of building a random tuples generator for the triangle domain. It suffices to remove the subdomains that do not intersect the domain D'. For example, in the Fig. 5, we can first remove the subdomains $D1, D2, D4, D7$ and then build a random tuples generator by first building a random sequence over $D_3, D_5, D_6, D_8, \ldots, D_{15}$. The algorithm we implemented is based on these same ideas. The key point of our PRT approach is that it employs constraint refutation to remove subdomains as we do not have a geometrical view of the solution set of the constraint system. We just have the symbolic

expressions of the path conditions.

4.5 Constraint refutation to test domain intersection

Constraint refutation is the process of temporarily adding a constraint to a set of constraints and testing whether there is no solution by constraint propagation. When there is no solution, then the added constraint is shown to be contradictory with the rest of the constraint system. When there could be a solution, then we cannot deduce anything as constraint propagation is incomplete in general. This process can be implemented very efficiently as constraint propagation over ranges is linear w.r.t. the number of constraints.

Consider again the triangle example and the constraint set $Y \geq 0, X \leq 14, X \geq Y$ that correspond to the triangle domain. Constraint propagation over these constraints give $D = (X \in 0..14, Y \in 0..14)$. Then adding the constraint corresponding to $D1 = (X \in 0..4, Y \in 12..14)$ and performing constraint propagation leads to exhibit a contradiction. As a result, $D1$ can be removed from the list of subdomains. By repeating this process until all the subdomains of D be tested, we get that $D1, D2, D4, D7$ can be removed, as expected.

It is worth noticing that constraint refutation is an efficient but incomplete process. As said previously, constraint propagation over finite domains does not guarantee satisfiability. Consider the following constraint system over finite domains: $X \in 1..2, Y \in 1..2, Z \in 1..2, X \neq Y, Y \neq Z, Z \neq X$. On this example constraint propagation does not perform any pruning on the domains, although the constraint system is clearly unsatisfiable. Hopefully, these situations are not so frequent in practice and inconsistent subdomains can often be refuted.

Using constraint refutation has another advantage as it is useful to detect some non-feasible paths. In fact, non-feasible paths correspond to unsatisfiable constraint systems. Hence, in the PRT approach, if all the subdomains of the partition are shown to be inconsistent, then that means the corresponding path is non-feasible. This contrasts with traditional RT which can never detect non-feasible paths. Note however that the PRT approach can miss to detect some non-feasible paths due to the incompleteness of constraint propagation.

4.6 Dividing the hypercube into equivalent subdomains

When a path is feasible, constraint propagation always results in an hypercube that is a correct over-approximation of the solution set of the path conditions. The PRT approach builds a random test data generator for the exact path-conditions solution domain within this hypercube. Special attention must be paid to the way this hypercube is broken into subdomains in order to preserve the uniformity of the generator. As the hypercube is only made of integer tuples, we need to introduce some tricks to preserve uniformity. Let k be a given parameter, called the division parameter, our PRT algorithm is based on the division of each domain variable 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. If the input domain is of dimension n, then this trick yields to partition the (augmented) hypercube into k^n subdomains.

For the triangle domain of Fig.3, 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 15, therefore we propose to enlarge the domain of x and the domain of y with a value each. Finally, we get the 16 following subdomains: $D_1 = (x \in 0..3, y \in 0..3)$, $D_2 = (x \in 4..7, y \in 0..3)$,..,$D_{16} = (x \in 12..15, y \in 12..15)$.

In practice, we recommend to select a small division parameter such as $k = 2, 3$ or 4 as the gain will be maximal[2] while the overhead will remain reasonable. In addition, the number of spurious tuples introduced by the process will remain neglictible.

4.7 An algorithm to perform path-oriented random testing

By making use of the two invariance principles described above we can then set up an algorithm able to perform path-oriented random testing (PRT). The algorithm takes as inputs a set of variables along with their finite variation domain, a constraint set corresponding to the path conditions (under Disjunctive Normal Form), the division parameter k, an integer N that represent 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 void whenever the corresponding paths are all non-feasible.

Algorithm 1: The Path-oriented Random Testing Algorithm

Input : $(Dom(x_1), \ldots, Dom(x_n))$, $(x_1, \ldots, x_n)$, C, k, N
Output : $t_1, \ldots, t_N$

```
T := ∅ ;
(D_1, ..., D_{k^n}) := Divide({Dom(X_1), ..., Dom(X_n)},k);
forall D_i ∈ (D_1, ..., D_{k^n}) do
    if D_i is inconsistent w.r.t. C 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 ≥ 1 then
    while N > 0 do
        Pick up D at random from D'_1, ..., D'_p;
        Pick up t = (y_1, ..., y_n) at random from D;
        if C is satisfied by t then
            add t to T ;
            N := N − 1;
        end
    end
end
return T;
```

The PRT algorithm makes use of the `Divide` function which partitions the hypercube in k^n subdomains as described above. The PRT algorithm is only a semi-correct procedure, meaning that it is not guaranteed to terminate, but when it terminates, it provides the correct result. Indeed, in the second loop, N is decreased iff t satisfies C, which can happen only if C is satisfiable. In other words, if C is unsatisfiable and if this has not been detected by the constraint propagation step ($p \geq 1$), then the PRT algorithm will surely not terminate. Note that similar problems arise whenever classical random testing with rejects is employed as nothing prevent an unsatisfiable goal C to be selected and in this case all the test cases will be rejected. In practice, a concurrent time out procedure is

[2]The smaller the value of k is, the larger the subdomains are

necessary to force termination. This process is not detailed here but it is mandatory on actual implementations.

The PRT algorithm produces a sequence of N random test data that is uniformly distributed over the solution set of C. In our experiments, we checked the uniformity hypothesis with the classical χ^2 test.

5. EXPERIMENTAL RESULTS

5.1 Our PRT and RT implementations

To evaluate Path-oriented Random Testing (PRT), we compared its results with respect to classical Random Testing (RT). We built two programs that both take path conditions and a set of domains as input parameters and provide a random test suite as a result. These two programs were implemented in less than one hundred lines of SICStus Prolog code. The random number generator is provided by a library based on a Richard A. O'Keefe's Prolog implementation of the AS 183 algorithm from Wichmann and Hill [18]. In addition, our PRT implementation exploits the constraint library clp(fd) which provides constraint propagation over finite domains.

5.2 Programs to be tested

We evaluated PRT w.r.t. RT on three programs: the `foo` program given in Fig.1, the `power` program given in Fig.2, and the well-known `trityp` program of the Software Testing literature. The program `trityp`, initially proposed by Myers [14] and fully studied by DeMillo and Offut [5], takes three non-negative integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene, isocele, equilateral or illegal. Although it implements a very simple specification, this program is difficult to handle for test data generators as it contains several imbricate conditional structures and a lot of non-feasible paths (43 over a total of 57 paths in the version of [5]). Moreover, it is usually considered as representative of the more general class of decisional programs (programs without iterative computations) that is mainly employed in the development of realtime embedded software.

All the results given in Tab.**??** were computed on a 2.0Ghz Pentium M personal computer with 1Go of RAM.

5.3 Experiments on the `foo` program

Tab.1 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 length of the expected random test data sequence. Tab.1 shows the number of test data generated with the PRT approach with three distinct values of the division parameter and traditional RT. In the first column, we asked PRT and RT to generate a sequence of 50 test data and get that RT generated 9942 test data (rejecting $9942 - 50$ test data) whereas PRT generated only 60 test data with a division parameter k equal to 2, 57 test data with $k = 3$ and 55 test data with $k = 4$. Note that we repeated three times the experiments and took the best results in all the cases (RT and PRT) to avoid the factor of bad luck that can be observed on a single experiment. Note also that we confirmed the uniformity of the resulting random sequence by using the χ^2 test with a parameter of 0.95. In PRT with $k = 2$, 1 subdomain over 4 was shown to be unsatisfiable whereas in PRT with $k = 3$, 5 subdomains over 9 and in PRT with $k = 4$ 11 subdomains over 16 were shown to be unsatisfiable. In this experiment, the CPU time required to get the random sequence (including unsatisfiability detection) is always less than 1sec so it is not indicated. The next experience will discuss this CPU time in more details. The experimental results shown in Tab.1 confirmed our manual analysis on the foo program about the probability of rejecting test data that do not satisfy the path conditions. In all the cases, PRT remains linear on the length of the generated sequence and so is clearly competitive with traditional RT.

Expected	50	100	150	200	250	300	350	4
RT	9942	17085	25751	35246	42682	51611	60429	72
PRT ($k = 2$)	60	133	206	253	321	376	465	5
PRT ($k = 3$)	57	113	179	229	288	351	411	4
PRT ($k = 4$)	55	110	166	226	280	330	397	4

Table 1: Length of the generated test suite on program `foo`

Tab.2 shows the CPU time required to generate long sequences of random test data on the `foo` program. With an Prolog-interpreted RT program, the CPU time required for random sequences of length greater than 10000 could not be computed as our implementation runs out of memory. Therefore, the RT program was compiled to optimize memory consumption while PRT remained interpreted[3]. Anyway, the results show that PRT (in interpreted mode) is almost two order of magnitude better than traditional RT (in compiled mode). The PRT approach outperforms traditional RT, at least on this example. We can object that traditional RT could have been implemented in C to optimize reject checking but this approach would have gained nothing but a constant factor.

Expected	5000	10000	15000	20000	25000	30000	35000
Compiled RT	27.7s	56.8s	83.2s	109.6s	140.3s	160.3s	–
PRT ($k = 2$)	0.3s	0.5s	0.6s	0.7s	0.9s	1.0s	1.1s
PRT ($k = 3$)	0.3s	0.4s	0.5s	0.6s	0.7s	0.9s	1.0s
PRT ($k = 4$)	0.3s	0.4s	0.5s	0.6s	0.7s	0.9s	1.0s

Table 2: CPU time required for generating random test suite on program `foo`

5.4 Experiments on the `power` program

We selected a single long path within the `power` program and computed its path conditions by using backward symbolic execution. The selected path $(1{\rightarrow}2{\rightarrow}(3{\rightarrow}2)^{1000}{\rightarrow}4{\rightarrow}6)$ iterates 1000 times on the loop of the program in order to build a large constraint system. In fact, it contains more than 1000 constraints. With this experiment, we would like to study how the PRT approach behaves when numerous constraints are involved in the computations. Note that the consistency of any of the computed subdomains were checked w.r.t. these 1000 constraints. Input variables were constrained to belong to 0..50000.

The experimental results show that PRT with $k = 2$ generates a random sequence of 10^4 test data in less than 1sec of CPU time. Here, 2 subdomains over 4 were removed but more importantly, the constraint propagation step yields to instantiate the second input parameter of the `power` program and then there was not a single reject. Indeed, all the randomly chosen test data were kept to sensitize the selected path. The same request for a compiled version of the RT program never answers as the event $Y = 1000$ has a very low probability to happen (we stopped the process after 1hour of computation).

[3]It is generally accepted that there is one order of magnitude between compiled and interpreted code

5.5 Experiments on the `trityp` program

For the `trityp` program, we manualy built first a list of control flow paths that covers the all decisions criterion. It contained 7 paths. In this process, we do not pay any attention to the path feasibility as in many structural testing tools. As a result, the list may contain some non-feasible paths. We restrained the domain of input variables to be in $0..100$ and compared PRT and RT when generating random sequences of increasing lengths. The experimental results are given in Tab.3. Note that in both cases (RT and PRT) nothing guarantees the all decision criterion to be covered as the list may contain non-feasible paths. It is well known that RT cannot easily cover the `trityp` program as there is at least one event which has a very low probability: generating an equilateral triangle. Of course, a similar drawback is expected for the PRT approach as it is mainly an RT approach. A randomly chosen value is never propagated on the constraint network as this would bias the random test data generator! The PRT approach with $k = 2$ does not eliminate any subdomain but as it makes use of constraint propagation to eliminate some of the non-feasible paths, the complete process outperforms again traditional RT. In fact, among the 7 paths, 4 are shown to be non-feasible. For the PRT method, to start finding inconsistent subdomains, the division parameter k must be instantiated to 13. In this case, 469 subdomains are inconsistents over a total of 2197. As this value depends on the problem, we have not used it in our experiments, so the results are just presented with $k = 2$. This experiment shows that PRT is suitable not only for a single path but also when a subset of paths is given as input. Of course, in this case, constraint propagation is less efficient as disjunctions are handled in a lazy manner in the constraint solver. In fact, it waits until one of the disjuncts is entailed by the rest of constraints. However, even in this case, PRT outperforms traditional RT as shown by Tab.3.

6. FURTHER WORK

In this paper, we introduced a new approach that combines both the advantage of partition testing and random testing. This approach, called Path-oriented Random Testing (PRT), implements RT over only a restricted subset of the control flow paths of a program to be tested. The challenging problem is to preserve the uniformity of the random generator only for a subdomain of the program's input domain. We have shown that PRT outperforms traditional RT by minimizing the number of rejects needed by any RT approach of this problem. Moreover, by exploiting subtle constraint propagation and refutation, the PRT approach also permits to detect some of the non-feasible paths, something which is out of the scope of traditional RT. Our further work will focus on techniques to improve the scope of current PRT methods. In particular, dealing with pointers and dynamic structures in PRT appears to be very challenging as we do not possess yet any constraint solvers on these constructions. Moreover, consider pointers as inputs of programs leads to consider unbounded input domains which is challenging for RT methods. More generally, for a random test data generator, maintaining uniformity on a subdomain defined by a set of constraints is a difficult and not yet solved problem. A lot of work remains to be done to understand what are the links between this problem and other RT approaches.

7. REFERENCES

[1] B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 2006. to appear.

[2] M. Carlsson, G. Ottosson, and B. Carlson. An open–ended finite domain constraint solver. In *Proc. of Programming Languages: Implementations, Logics, and Programs*, 1997.

[3] T. Chen, T. Tse, and Y. Yu. Proportional sampling strategy: a compendium and some insights. *The Journal of Systems and Software*, 58:65–81, 2001.

[4] R. DeMillo and J. Offut. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[5] R. DeMillo and J. Offut. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–127, April 1993.

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

[7] P. Ecuyer. *Random Number Generation*, chapter draft for a chapter of the forthcoming Handbook of Computational Statistics. Springer-Verlag, 2004. J. E. Gentle, W. Haerdle, and Y. Mori, eds.

[8] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'98)*, pages 53–62, Clearwater Beach, FL, USA, March 1998.

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

[10] 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. Also in CS-93-02 Brown–University 1993.

[11] S. Mankefors, R. Torkar, and A. Boklund. New quality estimations in random testing. In *The 14th Int. Symp. on Software Reliability Engineering (ISSRE'03)*, pages 468–478. IEEE Computer Society Press, 2003.

[12] K. Marriott and P. Stuckey. *Programming with Constraints : An Introduction*. The MIT Press, 1998.

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

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

[15] M. Petit and A. Gotlieb. An ongoing work on statistical structural testing via probabilistic concurrent constraint programming. In *SIVOES-MODEVA workshop – satelite event of Int. Symp. on Software Reliability Engineering (ISSRE'04)*, Saint-Malo, France, November 2004.

[16] E. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal of Computing*, 8(4):587–598, November 1979.

[17] E. Weyuker and B. Jeng. Analyzing Partition Testing Strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, Jul. 1991.

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

[19] D. Yates and N. Malevris. Reducing The Effects Of Infeasible Paths In Branch Testing. In *Proc. of Symposium on Software Testing, Analysis, and Verification (TAV3)*, volume 14(8) of *Software Engineering Notes*, pages 48–54, Key West, FL, Dec. 1989.

[20] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–426, Dec. 1997.

Expected	10000	20000	30000	40000	50000	60000	70000	80000
RT	17.6s	34.9s	52.6s	70.0s	87.7s	105.3s	123.1s	140.4s
PRT with $k = 2$	0.8s	1.8s	1.9s	2.5s	3.2s	3.8s	4.4s	5.0s

Table 3: CPU time required for generating random test suite on program `trityp`