Uniform Path Selection of Feasible Paths as a Stochastic Constraint Problem

Matthieu Petit* Arnaud Gotlieb
IRISA / INRIA
Campus Beaulieu
35042 Rennes cedex, FRANCE
{Matthieu.Petit,Arnaud.Gotlieb}@irisa.fr

Abstract

Automatic structural test data generation is a real challenge of Software Testing. Statistical structural testing has been proposed to address this problem. This testing method aims at building an input probability distribution to maximize the coverage of some structural criteria. Under the all_paths testing objective, statistical structural testing aims at selecting each feasible path of the program with the same probability.

In this paper, we propose to model a uniform path selector of feasible paths as a stochastic constraint program. Stochastic constraint programming is an interesting framework which combines stochastic decision problem and constraint solving. This paper reports on the translation of uniform feasible path selection problem into a stochastic constraint problem. An implementation which uses the library PCC(FD) of SICStus Prolog designed for this problem is detailed. First experimentations, conducted over a few academic examples, show the interest of our approach.

1 Introduction

Structural software testing aims at increasing our confidence in the correctness of a given (imperative) program. The major difficulties of Program Testing reside in a selection of a test suite which is a representative sampling of program behaviours. When the test suite is randomly generated according to a given probability distribution such as the probability to cover each part of the program is maximized, one speaks of statistical structural testing (SST) [14].

The SST problem lies in the difficulty of finding a probability distribution over the input domain that maximizes the probability to cover each element of the testing criterion. The structural testing criteria are based on the coverage of the control flow or data flow graph model of the program under test [17]. As an example, consider the all_paths criterion, the goal of statistical structural testing is to find a probability distribution such as each path has the same probability to be executed. In this case, giving the same probability to each path leads to maximize the probability of covering each path but for other criteria this is not so simple.

A key problem of the statistical structural testing is to select a sequence of paths such as the probability to cover each element of the testing criterion is maximized. This problem was originally studied by Thévenod-Fosse andWaeselynck [14] and more recently, Gouraud, Denis, Gaudel and Marre proposed new automated solutions based on combinatorial structures [7]. However, both approaches reason on a static model of the program under test and does not consider information on infeasible paths. Indeed, path selector is built on static information on the structure of the program under test. As basic example, consider the **foo** program given in Fig. 1. The SST problem for the all_paths criterion can

```
int foo(int x, int y) {
1. if (x * y < 100)
2.     ...
3.     else ...
4. if (x * y = 100)
5.     ...
6.     else ... }
```

Figure 1. Program foo

be modelled as a dice draw between the four paths of the **foo** program: 1-2-4-5, 1-2-4-6, 1-3-4-5 and 1-3-4-6. One dice draw corresponds to a path selection. When reasoning statically on a model like its control flow graph, the STT problem can be solved with a fair 4-face dice.

$$choose(X, [1,2,3,4] - [1,1,1,1], X = Dice)$$

The probabilistic choice operator $choose$ can modelled this dice draw. This probabilistic choice operator introduces a

*This work is part of the GENETTA project granted by the Brittany region

stochastic variable X where $[1, 2, 3, 4]$ represents its domain and $[1, 1, 1, 1]$ its uniform probability distribution. The path $1 - 2 - 4 - 5$ is selected when X is equals to 1, the path $1 - 2 - 4 - 6$ is selected when X is equals to 2, and so on. However, in this program, about one quarter of the selected paths is an infeasible path.

In this paper, we propose a method to address the problem of selecting feasible paths of the program with the same probability. A uniform path selector is extracted from a dynamic model of the program under test. Our approach is based on the translation of the problem of a uniform path selection of feasible paths (UPSFP) into a stochastic constraint problem. Stochastic constraint programming is a convenient framework to deal with decision making problem under uncertainty. UPSFP problem can be modelled as a biased dice drawing with an unknown bias and bias is constrained dynamically during the constraint solving. In the case of **Foo** program, the unknown bias can be modelled by a list of weight variables $[W_1, W_2, W_3, W_4]$ that will be constraint.

$$choose(X, [1, 2, 3, 4] - [W_1, W_2, W_3, W_4], X = Dice)$$

Outline of the paper. The paper is organized as follows : section 2 briefly describes the background on the stochastic constraint programming required to understand the rest of the paper. Section 3 presents the translation of uniform path selection of feasible path into a stochastic constraint problem, while section 4 describes the implementation of the translation in SICStus Prolog. Finally, Section 5 indicates several perspectives to this work.

2 Background

In this section, we introduce the stochastic constraint programming paradigm used to model the UPSFP problem. Using constraint programming to address the problem of structural testing [2, 5] or random testing [6] is not a new idea but, according to our knowledge, Stochastic Constraint Programming has not yet been used to address the UPSFP problem.

2.1 Stochastic Constraint Programming

Stochastic Constraint Programming was introduced by Walsh [15] to model combinatorial decision problems involving uncertainty and probability: resource management, network traffic analysis, energy trading. A stochastic constraint programs contain both decision variables, that can be set, and stochastic variables, which follow a discrete probability distribution.

A stochastic constraint program can be described in the formalism of the constraint satisfaction problem. A stochastic constraint satisfaction problem consists of 6-tuple $\langle V, S, D, P, C, \theta \rangle$ where V is a set of decision variables, S is a set stochastic variables, D is a function mapping each element of V and each element of S to a domain of potential values. A decision variable in V is instantiated to a value from its domain, P is a function mapping each element of S to a probability distribution for its associated domain. C is a set of constraints. Given random values for the stochastic variables, the probability that all constraints are satisfied equals or exceeds a threshold θ. Solutions of stochastic constraint satisfaction problem are obtained using an extended version of the algorithm used when doing constraint solving: backtracking or forward checking algorithm [15, 1].

In [9], Gupta et al. introduced a probabilistic choice operator in the framework of Concurrent Constraint Programming, named $choose$. The probabilistic choice operator allows introducing a stochastic variable into a constraint program. This probabilistic extension of the Concurrent Constraint programming has proved to be useful to model stochastic processes [8]. In [13], we proposed to extend the declarativity of the probabilistic choice operator $choose$ to reason with probabilistic choice only partially known. In this case, $choose$ operator is considered as a probabilistic constraint combinator. We tunes a probabilistic choice combinator, named $choose\_decision$, to address the UPSFP problem.

2.2 $choose\_decision$ combinator

$choose\_decision$ probabilistic constraint combinator has been introduced to simulate the behaviour of conditional statements into a constraint programming [11].

Probabilistic choice combinator models a boolean probabilistic choice between two constraints. This boolean probabilistic choice is represented as a list of two weights variables $[W_1, W_2]$.

$$choose\_decision(Constraint, [W_1, W_2], Ctrs_1, Ctrs_2)$$

The probabilistic choice arises between $Constraint \wedge Crts_1$ and $\neg Constraint \wedge Ctrs_2$. Note that $choose\_decision$ has been defined as a probabilistic combinator. The combinator allows dealing with problem where W_1 and W_2 are only partially known.

Operationally, when W_1 or W_2 is not valuated, the algorithm associated to the constraint solving of $choose\_decision$ tries to prove that $Constraint \wedge Crts_1$ or $\neg Constraint \wedge Ctrs_2$ is unsatisfiable. As unsatisfiability checking of a constraint system is an expensive process, only a partial unsatisfiability checking is associated to the probabilistic constraint combinator.

Our usage of this probabilistic constraint combinator to

address the UPSFP problem is described into the next section.

3 Stochastic constraint model of imperative programs

In this section, the translation of UPSFP problem into a stochastic constraint problem is presented. This translation is done on a generic fragment of an imperative language presented in the section 3.1. The stochastic constraint model is obtained in two stages: 1) executions of the imperative program are modelled as a probabilistic execution tree, and 2) stochastic constraints are generated from this tree.

3.1 Language While

For sake of clarity, we decide to describe our approach on a generic imperative language named While and composed of: *assignment statement*, *conditional statement*, *loop statement*, *compound statement* and *skip statement*. We discuss possible extension of this language in the last section of the paper. It assumes that programs have only a single return point *exit*.

Note that the coverage of all execution paths is generally an intractable testing criterion, due to the presence of loops. As done in [16, 4], we limit the number of iterations in a *loop statement* to a fixed number k. Hence in the following, *loop statements* are considered as k imbricated *conditional statements*.

3.2 Probabilistic execution tree

Uniform path selection of feasible paths is extracted from information on the structure of the program P under test. The probabilistic execution tree is a probabilistic extension of the execution tree proposed in [4] where branches of the execution tree are labelled by the probability to transit from a node to one of its successors.

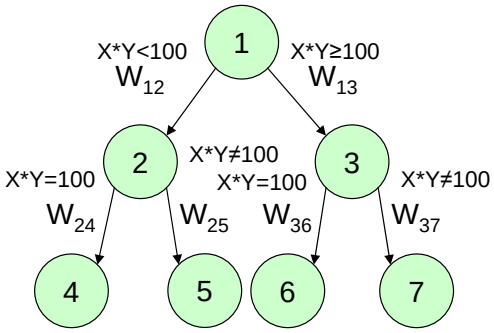
Let C be the set of *conditional statements*, A be the set *assignments statements* and P be the program under test. $Execs(P) := (C|A)^*(exit)$ represents the set of the possible execution of a program P. An execution tree of P is represented by a couple $\langle N, B\rangle$ where each statement of $Execs(P)$ is considered as an element of N, named statement node. The set B is a couple of statement nodes where the node associated to an assignment statement node has only one successor and to a conditional statement node have two successors. Root node denotes the input of the program. Leaves node of the execution tree are labelled by the *exit statement*. A path execution of a program P is defined as a sequence $N_1N_2 \dots N_n$ such as $\forall 1 \leq i \leq n, N_i \in N$, $\forall 1 \leq i \leq n-1, (N_i, N_{i+1}) \in B$ and N_n is an exit node.

Execution tree has been extended to model the UPSFP problem. We decided then to label each branch of the execution tree by the probability to transit from a node to one of its successors. These transition probabilities are represented by unknown weights. Our purpose is to constrain these unknown weights to model the testing objective.

Definition 1 (Probabilistic execution tree). Let P be a program under test and $\langle N, B\rangle$ be its execution tree. A probabilistic execution tree is a triplet $\langle N, B, W\rangle$ where W_{ij} an element of W associated to each element of $(N_i, N_j) \in B$. W_{ij} represents the probability to transit from the node N_i to the node N_j.

In the following, we denotes by W_{exit} a subset of the weights associated to an exit node.

Example. *The probabilistic execution tree of the* **Foo** *program is given by the Fig. 2.*



The set of the weight variables associated to exit node statements is:

$$W_{Exit} = \{W_{24}, W_{25}, W_{36}, W_{37}\}$$

Figure 2. Probabilistic execution of the foo program

3.3 Stochastic constraint program generation

The stochastic constraint program generation is obtained by a deep first search in the probabilistic execution tree. For sake of clarity, we present the constraint generation in two steps.

- translation of each statement node of the probabilistic execution tree into stochastic constraint;
- generation of constraints on the weight variable to model our testing objective.

In practice, stochastic constraints program generation is done in a single step.

3.3.1 Translation of the program into a stochastic constraint program

A logical variable is associated to each variable of the imperative program. A constraint domain is defined for a logical variable from the variable type.

Assignment node Statement x:= expr is translated into `X#=E` where `E` is the syntactic translation of $expr$. Branch associated to the node output is translated into a logical conjunction with generated constraints for the successor.

Conditional node A conditional node is translated into a `choose_decision(C,[W1,W2],Ctr1,Ctr2)` where `C` is the syntactic translation of the boolean expression associated to the conditional node, `[W1,W2]` represents the probabilistic choice between the two successors and `Ctr1` (resp. `Ctr2`) is the translation of the tree associated to the first (resp. second) successor.

Exit node Statement x:= expr associated to the node is translated into `X#=E` where `E` is the syntactic translation of $expr$.

From the constraint generation, path conditions of an execution path can be extracted. Indeed, path condition associated to an execution is a conjunction of constraints generated by the assignment node and conditionnal node. Suppose that $Path$ is an execution path of a program P, $pathcond(Path)$ denotes $Path$ path conditions.

Propriety 1. *Let P be a program. Then, for all $Path$ path execution of the probabilistic execution tree associated to P*

$pathcond(Path)$ is satisfiable $\Leftrightarrow$ $Path$ is a feasible path.

3.3.2 Uniform path selection of feasible paths

Our approach aims at constraining weight variables given a testing objective. Under the testing objective: give the same probability to each feasible paths to be activated, the weight variables are defined as the number of the feasible paths which activate the branch associated to the weight variable. Indeed, a conditional statement introduces a flow transfer between two possible parts of the program. The number of the feasible paths which activate the conditional statement node is divided by two. The probability to transit to one of the successor of the conditional statement node is then proportional to the number of the feasible paths which activate the two different branches. Then, the definition of the weight variable allows to give a greater probability to branches which are activated by a lot of feasible paths than branches which are activated by few feasible paths.

Obviously, given a branch of the execution tree, the number of feasible paths which activates this branch is unknown. Weight variables are constrained as follows.

Assignment node Suppose that W_i (resp. W_o) is the weight variable associated to the branch which goes in (resp. out) the node. Then, a constraint `Wi #= Wo` is generated because of the number feasible path stays unchanged.

Conditional node Suppose that W_i is the weight variable associated to the branch which goes in the node and W_{o_1} and W_{o_2} the weight variables associated to the two branches which go out the node. Then, a constraint `Wi #= Wo1+Wo2` is generated because of the number of feasible path is divided by two due to the conditional statement.

Exit node Suppose that W_{exit} is the weight variable associated to the branch which goes in the exit node and $Path$ the execution path which contains this node. The constraint `We in 0..1` is generated because only a single feasible path or a infeasible path activates this branch of the three.

3.4 A uniform path selector of feasible paths

A uniform path selector of feasible paths is obtained when each weight variable is valuated. One approach consists in testing the satisfiability of each execution path of the program. By the result given by the property 1, this approach allows to detect all feasible paths of the program and then to find immediately of valuation of the weight variables. A major drawback of this approach reside in the fact that we must restrict constraint domain to be in a decidable theory and constraint solving to be obtained in polynomial time. Moreover, there is a combinatorial explosion of constraint systems to resolve. Our approach consists in defining the testing objective as an objective function to optimize. The idea is to over-approximate the number of feasible path and to refine the first valuation of the weight variables given more information on the program under test. Information is obtained as follows:

- partial unsatisfiability checking associated to $choose\_decision$ permits to detect that some execution paths are infeasible;
- test cases generation allows to detect that a path is feasible.

By definition, each weight variable $W_{e_i} \in W_{exit}$ represents that an execution path which activates the exit node N_{e_i} is feasible or infeasible. When W_{e_i} is valuated to 0, the execution path has been detected as infeasible. When W_{e_i} is valuated to 1, the execution path has been detected as feasible. When W_{e_i} in $0..1$, constraint solving has not been able to conclude that the execution path is feasible or infeasible. Then, to conserve a sound uniform path selector of feasible paths, the following objective function is maximizes.

$$\sum_{Wj \in Wexit} W_j$$

Indeed, when $\sum_{Wj \in Wexit} W_j$ is maximized, weight variables of $Wexit$ which are not valuated before the solution

search are valuated to 1. By this process, we obtain a uniform path selector on an over approximation of the feasible paths set.

3.5 Example

Consider again the **Foo** program introduced by the Fig. 1 and its probabilistic execution tree given by the Fig. 2. Here is the translation of the **Foo** program into a stochastic constraint problem.

```
foo(X,Y,[W12,W24,W25,W13,W36,W37]) :-
   X in - 2147483647..2147483648,
   Y in - 2147483647..2147483648,
   choose_decision(X*Y#<100,[W12,W13],
               [choose_decision(X*Y#=100,
                         [W24,W25],[],[])],
               [choose_decision(X*Y#=100,
                         [W36,W37],[],[])]).
    W12+W13#>=0, W12 #= W24+W25,
    W13 #= W36+W37, W24 in 0..1,
    W25 in 0..1,W36 in 0..1,
    W37 in 0..1.
```

Partial unsatisfiability checking of $choose\_decision$ allows to detect that the path $1-2-3-4$ is infeasible. Then, `W24` is valuated to 0. Constraint propagation permits to reduce the domain of each weight variables to:

```
    W12 in 0..1,W13 in 0..2,
    W24 #= 0,W25 in 0..1,
    W36 in 0..1,W37 in 0..1
```

Uniform path selector of feasible paths for the **foo** program is modelled by the following request:

```
?- foo(X,Y,[W12,W24,W25,W13,W36,W37]),
   labeling([maximize(W24+W25+W36+W37],
               [W12,W24,W25,W13,W36,W37]).
```

The predicate `labeling/2` permits to find a valuation the weights variable `[W12,W24,W25,W13,W36,W37]` such as our objective function `W24+W25+W36+W37` is optimized.

In reiterating the request 5000 times, the uniform path selector constraint the variable `X` and `Y` to verify `X*Y #<100,X*Y#\=100` with a probability 0.3522, `X*Y #>=100,X*Y#\=100` with a probability 0.3162 and `X*Y #>=100,X*Y#=100` with a probability 0.3316.

4 Implementation

In this section, we detail the implementation of our approach. This implementation is done in SICStus Prolog. We also present a first experimental validation conducted on an academic program: Trityp.

4.1 Generation of a PCC(FD) request

The implementation uses two libraries of SICstus prolog: clp(FD) and PCC(FD). The library clp(FD) is a constraint solver finite domain that is used to resolve arithmetic constraints. The library PCC(FD) is a library of probabilistic constraint combinators. The implementation takes as input a program represented by its abstract syntax tree. The stochastic constraint program and the objective function is then generated from a deep first search algorithm in the tree.

The uniform path selector of feasible paths is modelled by a simple request which calls the stochastic constraint program and the labeling predicate. The labelling process is looking for a weight variables valuation such as the objective function is maximized This predicate uses a branch and bound algorithm. Note that before launched the labelling process, domains of each weight variable are recorded. Then, information on previous uniform path selector is conserved during the selection of a new feasible path. Uniform path selector of feasible paths is refined during the path sequence generation when a domain of the weight variable is pruned.

4.2 First experimental result

The program trityp, initially proposed by Myers [10] and fully studied by DeMillo and Offut [3], takes three non-negative integers as arguments that represent the relative lengths of the sides of a triangle and classifies the triangle as scalene, isosceles, equilateral or illegal.

Although it implements a very simple specification, this program is difficult to handle for test data generators as it contains several nested conditionals structures and a lot of infeasible paths (43 over a total of 57 our program version). 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 real time embedded software.

Partial unsatisfiability checking associated to the probabilistic constraint combinator $choose\_decision$ allows to detect 40 of the infeasible paths. Then, this information permits to build an efficient uniform path selector of feasible paths because of only $\frac{3}{17}$ of the generated paths are rejected. Moreover, the uniform path selector of feasible path is refined to avoid the random draw of feasible paths during test cases generation. This first experimental validation conducted on an academic example shows the practical interest of our approach.

5 Further Work

In paper, we propose to define a dynamic model to address the problem of the uniform path selection of feasible

paths. UPSFP problem has been translated into a stochastic constraint program. First experimental validations show the interest the approach.

Our further works will focus on 1) generating a statistical sequence of test cases extracted from a sequence of uniformly selecting feasible paths and 2) extending our approach to an embedded Java language: Javacard. To address the first problem, we are working on the combination of two approaches: uniform path selection of feasible paths and path oriented random testing [6]. Extending our approach for Javacard allows us to proposed a statistical method to test programs composed of method calls, exceptions which are a real challenge for the statistical structural testing.

References

[1] Thanasis Balafoutis and Kostas Stergiou. Algorithms for stochastic csps. In *12th International Conference on Principles and Practice of Constraint Programming*, pages 44–58, Nantes, France, Sept. 2006. Springer.

[2] R.A. DeMillo and J.A. Offut. Constraint-based automatic test data generation. *IEEE Trans. on Soft. Eng.*, 17(9):900–910, Sep. 1991.

[3] R.A. DeMillo and J.A. Offut. Experimental results from an automatic test case generator. *ACM Transactions on Software Engineering Methodology*, 2(2):109–175, 1993.

[4] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, USA, June 2005. ACM.

[5] 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.

[6] A. Gotlieb and M. Petit. Path-oriented random testing. In *Proceedings of the 1st International Workshop on Random Testing (RT'2006)*, pages 28–35, Portland, USA, 2006. ACM.

[7] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Sixteenth IEEE International Conference on Automated Software Engineering (ASE)*, pages 5–12. IEEE Computer Society Press, 2001.

[8] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Proceedings of Symposium on Principles of Programming Languages*, 1999.

[9] V. Gupta, R. Jagadeesan, and V.A. Saraswat. Probabilistic concurrent constraint programming. In *Proceedings of CONCUR*, pages 243–257. Springer, 1997.

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

[11] M. Petit and A. Gotlieb. An ongoing work on statistical structural testing via probabilistic concurrent constraint programming. In *Proc. of SIVOES-MODEVA workshop*, St Malo, France, November 2004.

[12] M Petit and A. Gotlieb. Library of probabilistic constraint combinators over finite domain, available at http://www.irisa.fr/lande/petit/tools.html. May 2006.

[13] M. Petit and A. Gotlieb. Constraint-based reasoning on probabilistic choice operators. Research Report 6165, INRIA, 04 2007.

[14] P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Journal of Sotware Testing, Verification and Reliability*, 1(2):5–25, July 1991.

[15] T. Walsh. Stochastic constraint programming. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 111–115, Lyon, France, 2002. IOS Press.

[16] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *5th European Dependable Computing Conference*, pages 281–292, Budapest, Hungary, April 2005. Springer.

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