INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Probabilistic Choice Operators as Constraint Combinators: Application to the Statistical Structural Testing Problem

Matthieu Petit and Arnaud Gotlieb

N° 6223

Thème SYM

Rapport de recherche

ISSN 0249-6399 ISRN INRIA/RR--6223--FR+ENG

Probabilistic Choice Operators as Constraint Combinators: Application to the Statistical Structural Testing Problem

Matthieu Petit* and Arnaud Gotlieb

Thème SYM — Systèmes symboliques
Projet Lande

Rapport de recherche n° 6223 — — 19 pages

Abstract: Probabilistic Concurrent Constraint Programming (PCCP) extends concurrent constraint languages by providing probabilistic choice operators. These operators have proved to be useful in implementing randomized algorithms as well as stochastic processes. In this report, we present an instance of PCCP where probabilistic choice operators are modelled with constraint combinators. This modeling allow us to deal with a new kind of problems where the probabilistic choice is only partially known. An implementation under the form of a library of SICStus Prolog is in progress. The major practical application of our work is the statistical structural testing problem in software testing.

Key-words: Probabilistic Concurrent Constraint Programming, Probabilistic Choices Combinator, Statistical Structural Testing

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

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 Rennes Cedex (France)
Téléphone : +33 2 99 84 71 00 — Télécopie : +33 2 99 84 71 71

Des Opérateurs de Choix Probabilistes comme des Combinateurs de Contraintes : Application au Problème du Test Statistique Structurel

Résumé : La Programmation Concurrente par Contraintes Probabilistes (PCCP) étend la Programmation Concurrente par Contraintes par des opérateurs de choix probabilistes. Ces opérateurs ont prouvé leur utilit dans l'implantation d'algorithmes "randomisés" ainsi que dans la modélisation de processus stochastiques. Dans ce rapport, nous présentons une instance de PCCP dans laquelle les opérateurs de choix probabilististes sont modélisés comme un combinateurs de contraintes. Cette modélisation permet de traiter de nouveaux problèmes pour lesquels le choix probabiliste n'est que partiellement connu. Une implantation sous la forme d'une librairie de SICStus Prolog est présentée. La majeure application de notre travail est le test statistique structurelle.

Mots-clés : Programmation Concurrente par Contraintes Probabilistes, Combinateur de choix probabilistes, Test Statistique Structurel

1 Introduction

Probabilistic Concurrent Constraint Programming (PCCP) was concurrently introduced by Di Pierro and Wiklicky [5] and Gupta, Jagadeesan and Saraswat [10] to model randomized algorithms [6] and stochastic processes [9] in a declarative way. In both cases, a probabilistic choice operator was added to Concurrent Constraint Programming (CCP) [17] to introduce probabilistic behaviours in concurrent constraint processes. A probabilistic choice between two CCP processes can be thought of as flipping a coin : head the first process is triggered, tail the second process is triggered.

In CCP, concurrent processes interact via a common constraint store, which is a conjunction of constraints on the possible values of variables. The computational model acts by accumulating constraints into the constraint store and by checking whether the store entails constraints. In [5], the classical non–deterministic choice operator [18] of CCP is extended with probabilities to form a probabilistic choice operator whereas [10] introduces an operator that constrains an internal random variable. In [5], random choices operate over processes whereas in [10] random choices operate over values of variables. It is worth noticing that for both operators, random draws must be done over an instantiated probability distribution. According to our knowledge, only one implementation of PCCP has been proposed [1]. It takes the form of a meta-interpreter in Prolog and implements the probabilistic non-deterministic choice operator of [5].

In our work, we propose to extend the probabilistic choice operator introduced in [10] by defining it as a constraint combinator. Combinators are meta-constraints that express complex relations between variables and constraints. The main contribution of our approach resides in the possibility of expressing probabilistic choice relations without requiring all the parameters to be instantiated. In particular, we allow expressing random drawings over an unknown probability distribution. By this, we increase the declarativity of the probabilistic choice operator and open the door to the modelling of a new kind of problems. As a very basic example, consider the simulation of a dice drawing. When the dice is unbiased (its probability distribution is then fully instantiated), the following PCCP process simulates the dice drawing:

$$choose(X, [1, 2, 3, 4, 5, 6] - [1, 1, 1, 1, 1, 1], tell(X = Dice)).$$

The distribution probability is represented by a list of weights ($[1, 1, 1, 1, 1, 1]$) that yields to a $\frac{1}{6}$ probability of drawing each of the six possible dice face. However, the simulation of dice drawing becomes non-feasible whenever the bias of the dice is unknown or just known via some constraints (for example, knowing that the 6-face of the dice is two times more overloaded than the 1-face). As previously said, PCCP requires indeed the probability distribution of its probabilistic operator to be fully instantiated. By defining the probabilistic choice operator as a constraint combinator, such a problem can be simulated via the following request:

$$tell(W_6 = 2 * W_1) \mid\mid choose(X, [1, 2, 3, 4, 5, 6] - [W_1, W_2, W_3, W_4, W_5, W_6], tell(X = Dice))$$

This basic example is representative of a more realistic class of problems where the unknown is actually the probability distribution. We met these problems whenever working on statistical structural testing, which is a well-established software testing technique [20]. Statistical structural testing aims at finding a distribution probablity over the input domain of a program that maximizes the coverage of some structural criteria, such as all_statements or all_paths. Up to now, there is no satisfactory automated technique for biasing the choice of test data that meet this objective.

This report introduces probabilistic choice operators modelled as constraint combinators, also known as global constraints. We implemented these combinators under the form of a SICStus Prolog library called PCC(FD) built over `clp(fd)` (a.k.a. finite domains constraints). A preliminary version of this library is available [15]. In this report, we focus on the practical applications of this library to the statistical structural testing problem rather than focussing on the implementation details. By giving some insights on the filtering capacities of these combinators, we explain how they can be used to efficiently solve practical problems where the unknown is actually the probability distribution.

The report is organized as follows : Section 2 briefly recalls the syntax and semantics of PCCP; Section 3 describes the operational semantics of the main probabilistic choice constraint combinator while Section 4 presents the implementation of several probabilistic constraint combinators in SICStus Prolog. Section 5 details the application of this library to the statistical structural testing problem. Finally, Section 6 indicates several perspectives to this work.

2 Probabilistic Concurrent Constraint Programming

Before presenting PCCP, we start by recalling some syntax and semantics elements of the classical concurrent constraint programming paradigm (CCP) introduced by Saraswat [17].

2.1 Concurrent Constraint Programming

In CCP, processes are executed concurrently and can interact with each other through a common constraint store. A CCP language is parameterized by a constraint system [18], which is composed of a set of primitive constraints and an entailment relation. The syntax of a CCP language can be given using the following grammar :

$$Process ::= \quad tell(C)\,|\,if\ C\ then\ Process\,|\,new\,X\,in\,Process\,|\,Process \| Process.$$

where $tell(C)$ adds the constraint C to the constraint store, $if\ C\ then\ Process$ asks whether C is entailed by the current constraint store and adds the constraints of $Process$ if C is entailed, *new X in Process* adds the constraints of *Process* to the store while hiding the variable X from other processes, and finally, $\|$ represents the parallel composition that can be interpreted as a logical conjunction in a Logic Programming environment. Well known examples of CCP languages include cc(FD)[11], AKL[12] or Oz/Mozart[19] just to name a few.

2.2 Probabilistic Choice Operator

In [10], Gupta et al. proposed to add a probabilistic choice operator to CCP. The operator $choose(X, Law_X, Process)$ injects a random variable X along with a probabilistic law Law_X into a concurrent process $Process$. Law_X contains a list of possible values for X, $[v_1, \ldots, v_n]$ along with a list of non negative weights $[w_1, \ldots, w_n]$ associated to each v_i. In this operator, X cannot be constrained outside the concurrent process, i.e. X is local to $Process$. Operationally, $choose(X, [v_1, \ldots, v_n] - [w_1, \ldots, w_n], Process)$ executes $Process_{X \leftarrow v_i}$ with a probability p_i where $Process_{X \leftarrow v}$ denotes the concurrent process P where X has been substituted by v and p_i denotes the probability of the event $X = v_i$ which is computed by the following formula:

$$p_i = \frac{w_i}{\sum_{j=1}^{n} w_j}.$$

2.3 Operational semantics

Running processes of PCCP can be formally described by using a probabilistic transitions system $(\Gamma, \longmapsto_p)$ where Γ denotes the set of states of the transition system, also called configurations. A configuration is a pair $\langle Process, \sigma \rangle$ where $Process$ denotes all the remaining processes to be executed while σ is the constraint store. As usual, the transition relation $\longmapsto_p$ is defined with the help of axiomatic rules that are available in [9]. Let σ_0 be an initial constraint store, then the set of terminal configurations $tc(Process, \sigma_0)$ in the operational semantics of PCCP are defined as:

$$tc(Process, \sigma_0) = \{\sigma \mid \langle Process, \sigma_0 \rangle \longmapsto_{\bar{p}}^{*} \sigma \not\longmapsto_p\}.$$

where $\bar{p}$ denotes the probability to reach σ and $\longmapsto_{\bar{p}}^{*}$ denotes the transitive closure of the transition relation. In PCCP, only consistent constraint stores (consistent terminal configurations $ctc(Process, \sigma_0)$) are considered for further computations:

$$ctc(Process, \sigma_0) = \{\sigma \mid \langle Process, \sigma_0 \rangle \longmapsto_{\bar{p}}^{*} \sigma \not\longmapsto_p \text{ and } consistent(\sigma)\}$$

Just to make things more concrete, we illustrate the processing of a PCCP request on a basic example extracted from [9]:

Example 1.

$$\begin{aligned} P = \quad & choose(X, [0,1] - [1,1], tell(X = Z)) \parallel \\ & choose(Y, [0,1] - [1,1], if\ Z = 1\ then\ tell(Y = 1)). \end{aligned}$$

Roughly speaking, three possible terminal configurations can be obtained: Z is constrained to 0 with the probability $\frac{1}{2}$ (event $X = 0$), Z is constrained to 1 with the probability $\frac{1}{4}$ (event $X = 1 \wedge Y = 1$) and $false$ is obtained with the probability $\frac{1}{4}$ (event $X = 1 \wedge Y = 0$). By eliminating the third possible answer, we get $ctc(P, true) = \{Z = 0, Z = 1\}$.

3 Probabilistic choice as a constraint combinator

In PCCP, the probabilistic choice operator requires the couple $Domain - Distribution$ to be fully instantiated. In this section, we relax this requirement by introducing the probabilistic choice operator *choose* as a constraint combinator. As previously claimed, this allows us to reason on probabilistic choices that are only partially known. In this view, the probability distribution associated to X is just constrained by a set of constraints on its possible values. This allows to establish a relation between the probabilistic choice and the set of its possible probability distribution. In the rest of the report, we instantiate PCCP to the case of finite domains constraints although this is not a strong requirement of our approach. In fact, *FD* is the adequate domain for our software testing application, described in Sec.5.

3.1 Probabilistic laws over finite domains

We start by giving the definition of shape of a probabilistic law over *FD*. The form is as follows:

Definition 1. *Let* $V_1, \ldots, V_n$ *and* $W_1, \ldots, W_n$ *be* $2n$ *finite domain variables, then* $[V_1, \ldots, V_n] - [W_1, \ldots, W_n]$ *denotes a probabilistic law over finite domains for the random variable* X *iff the probability of the event* $X = V_i$ *is equal to* $\frac{W_i}{\sum_{j=1}^{n} W_j}$.

3.2 Uncertainty on the probabilistic law

The simulation (random drawing) of values for the random variable X is always possible when Law_X is fully determined, i.e. when all the finite domain variables of its probabilistic law are instantiated. Uncertainty on the probability distribution only appears whenever the variables of the probabilistic law Law_X can take several possible values. This uncertainty is characterized by the following definition:

Definition 2. *Let* X *a random variable,* $Law_X = [V_1, \ldots, V_n] - [W_1, \ldots, W_n]$ *its probability law. We defined the set of the possible probability distributions to* X*, called* $\mathcal{SL}_X$*, as follows:*

$$\mathcal{SL}_X = \big\{[w_1, \ldots, w_n] \big| w_1 \in dom(W_1), \ldots, w_n \in dom(W_n)\big\}$$

where $dom(W)$ *is the domain of a variable* W*.*

is an over approximation of the possible probabilistic laws for X. The uncertainty on the probability law is caused by the uncertainty on the probability distribution. Indeed, the simulation of values for X is always possible when the probability distribution is fully valuated.

Example 2. Consider the example of dice drawing given in the introduction:

$$tell(W_6 = 2 \cdot W_1) \,||\, choose(X, [1, 2, 3, 4, 5, 6] - [W_1, W_2, W_3, W_4, W_5, W_6], tell(X = Dice))$$

Suppose that $dom(W_1) = 1..2$, $dom(W_2) = 2..2$, $dom(W_3) = 2..2$, $dom(W_4) = 2..2$, $dom(W_5) = 2..2$ and $dom(W_6) = 2..4$, then the uncertainty on the unknown bias of the dice is given by the following set:

$$\mathcal{SL}_X = \{ \begin{array}{l} [1,2,2,2,2,2], [1,2,2,2,2,3], [1,2,2,2,2,4], \\ [2,2,2,2,2,2], [2,2,2,2,2,3], [2,2,2,2,2,4]\}. \end{array}$$

It is worth noticing that only two probabilistic laws over six satisfy the constraint $W_6 = 2{\cdot}W_1$. Hence, $\mathcal{SL}_X$ is indeed an over approximation of the possible probabilistic laws for X.

When considered over finite domains, the combinator *choose* $(X, [V_1, \ldots, V_n] - [W_1, \ldots, W_n], Process)$ builds a relation between the simulation of values for X and the set of finite domain variables of its probabilistic law. Technically, X is not a finite domain variable (albeit it can only take a finite set of possible values), it is a random variable, meaning that the choice over its possible values is done at random. However, using the definition of $\mathcal{SL}_X$, the relation $choose(X, [V_1, \ldots, V_n] - [W_1, \ldots, W_n], Process)$ allows us to reason and filter over the possible random values of X. In the next section, we give an operational description of this process.

3.3 Operational description of *choose*

The constraint combinator $choose(X, [V_1, \ldots, V_n] - [W_1, \ldots, W_n], Process)$ is managed by the solver into the finite domain constraint propagation mechanism. The solver exploits the relation between X and $\mathcal{SL}_X$ to prune the domain of X in order to accelerate the convergence toward a fix point and reason even in the absence of certainty over the probabilistic choice. First of all, the combinator *choose* just succeeds whenever X is valuated. In this case, the constraints of *Process* are set up into the (dynamic) constraint system. When X cannot be instantiated during the constraint propagation step, then a filtering algorithm is launched to prune the possible values of X, i.e. to remove any finite domain variable V_i of the domain of the possible values for X that can be randomly drawn When no more pruning can be performed, the constraint combinator falls into an asleep state. It is awoken whenever the domain of at least one variable of the probabilistic law is modified.

The pruning algorithm launched during constraint propagation exploits the fact that X can be drawn very early at random in $[V_1, \ldots, V_n]$ according to the probability distribution defined by $[W_1, \ldots, W_n]$. Although the pruning algorithm plays a prevalent role in the acceleration of convergence, it is not given here as we preferred to focus on the applications of the constraint combinator *choose* rather than focusing on technical details. However, all the details of the filtering process can be found elsewhere [16]. Note just that the complexity of the filtering algorithm is only linear w.r.t. the length of the domain of the random variable X [16].

4 The PCC(FD) library

In this section, a library of three probabilistic choice operators defined as new combinators of the constraint system is presented. The implementation of these combinators is based on the clp(FD) library of SICStus prolog [2], by making use of its global constraint definition interface. Before switching on the description of combinators, let us just explain the predicate `ptc(Goal,Var_List,Result)` which empirically computes the set of terminal configurations in PCC(FD).

4.1 Empirical computation of the set of terminal configurations

Given a Prolog goal `Goal` along with a list of variables `Var_L ist`, the `ptc` predicates launches a given number (5000) of `Goal` runs, records the resulting constraint store projection (i.e. projection of domains on `Var_List`) after the constraint propagation step, and computes the occurrence rate of each constraint store projection. By using this predicate, one can study the probabilistic behaviour of our constraint combinators in PCC(FD).

4.2 Probabilistic combinators in PCC(FD)

The three combinators distinguish themselves on the three possible notations of the probabilistic choice we gave above. In all the three cases, the probabilistic choice *Domain* − *Distribution* takes the form of Prolog terms that can be either closed or disclosed. A closed term is associated to a fully instantiated probabilistic choice whereas a disclosed term represents uncertainty on the probabilistic choice. The three combinators are:

- `choose`, where *Domain* is a list of values and *Distribution* is a list of finite domain variables that represent weights;
- `choose_range`, where *Domain* is a range represented with two distinct FD variables Min and Max, and *Distribution* is a difference list of finite domain variables that represent weights;
- `choose_decision`, where *Domain* is the boolean domain $\{0, 1\}$ and *Distribution* is a couple of distinct finite domain variables that represent weights.

4.3 The `choose` combinator

The syntax of the combinator `choose` is as follows:

```
choose(X,[V1,..Vn]-[W1,..,Wn],Goal,Options)
```

where X is a random variable and `V1,..,Vn,W1,..,Wn` are finite domain variables. Recall that X cannot be constrained elsewhere in the Prolog program as it is local to the combinator. `Options` is used to parameterize the filtering capacities of the combinator into the constraint propagation mechanism. Three options are currently available:

- `no_filtering` can be employed to switch off the pruning capacities of the filtering algorithm. This option is useful for experiments ;
- `inconsistency_check` can be used to check the consistency of all the possible values V of `X` with respect to `Goal`. This option allows one to parameterize the filtering algorithm by trying to refute `Goal` $\wedge X = V$. This option is useful to improve the pruning capacities of the combinator but requires a lot of memory space. Its usage is then confined to some specific situations (for example, when `Goal` is made of a single constraint) ;
- `lvar(L)` can be used to enrich the list of variables on which the combinator is awaked. This option is useful to parameterize the awaking conditions of the combinator.

By default, none of these options are activated.

Let us illustrate the behaviour of the choose combinator on the example 2 of the biased dice given above.

Example 3. `dice(Dice) :-`

```
  W1 in 1..2,W2 in 2..2,W3 in 2..2,W4 in 2..2,W5 in 2..2,W6 in 2..4,
  2*W1#=W6,
  choose(X,[1,2,3,4,5,6]-[W1,W2,W3,W4,W5,W6],[X=Dice],[]).

? - ptc(dice(Dice),[Dice],Result).

Result=[(Dice=1,0.07735),(Dice in 1..2,0.09065),
        (Dice=2,0.06285),(Dice in 2..3,0.10175),
        (Dice=3,0.05135),(Dice in 3..4,0.11795),
        (Dice=4,0.03860),(Dice in 4..5,0.12775),
        (Dice=5,0.02575),(Dice in 5..6,0.1401),
        (Dice=6,0.16590)]
```

The results show the different constraint store projections on X obtained after the constraint propagation step. For example, `(Dice=1,0.07735)` means that `Dice` is equal to 1 with a probability `0.07735`. For some cases, the dice has been instantiated although the probability distribution was not fully known. On the contrary, some other constraint store projections show domains that contain two values. In this case, the filtering process failed to instantiate X but dramatically pruned the domain of possible values for X. On this example, our implementation permits valuating `X` with a probability of 0.4018 which is an good performance as these cases correspond to obtaining the dice face without knowing the bias of the dice.

4.4 The `choose_range` combinator

The `choose_range` combinator implements a probabilistic choice operator for a range of values. The range is given by `[Xmin,Xmax]`, where `Xmin` and `Xmax` are two finite domain variable. `Xmin` denotes the lower bound of X wheras `Xmax` denotes its upper bound. The probability distribution takes the form of a list as the exact length of the probability distribution is unknown. The probability distribution itself evolves according to the domain of `Xmin` and `Xmax`. Information on `Xmin` and `Xmax` is exploited to constrain the shape of the list.

`choose_range(X,[Xmin,Xmax]-[W1,..,WN],Goal,Options)`

where `N` is equal to `max(Xmax)-min(Xmin)+1`. `no_filtering, inconsistency_check and lvar(L)` options are available.
`choose_range(X,[Xmin,Xmax]-[W1,..,Wn|Q]-Q,Goal,Options)` can be rewritten as `choose(X,[min(Xmin),..,max(Xmax)]-[W1,..,WN],[Goal],Options)`.

We illustrate the `choose_range` combinator on an implementation of the (weak) Miller-Rabin primality test. This well_known primality test is based on a randomized algorithm that can be implemented in PCC(FD). The core of the test is built on the contrapositive form of the Fermat's little theorem, saying that if n is a prime number and $x \in \mathbb{N}^*$ such as $\gcd(x, n) = 1$, then

$$x^{n-1} \equiv 1 \bmod n. \tag{1}$$

If there exists $x \in \mathbb{N}^*$ such as $gcd(x, n) = 1$ and $x^{n-1} \not\equiv 1 \bmod n$, then n is a composite number (i.e. not a prime number). Trivial algebraic manipulations show that x can be chosen in $\{2, \ldots, n-1\}$ For great values of n, it is unreasonable to verify that $\forall x \in \{2, \ldots, n-1\}$ the equation (1) is satisfied (a NP_hard problem), hence values for x are chosen according to a uniform probability distribution. This randomized algorithm is easily modelled (for any number `N`) by making use of the `choose_range` combinator, as shown in Fig. 1. In

Figure 1 An implementation of the (weak) Miller-Rabin primality test in PCC(FD)

```
primal_testing(N,K) :-
    Xmax #= N-1,
    itere(N,Xmax,K).

itere(_N,_Xmax,0).
itere(N,Xmax,K) :-
    choose_range(X,[2,Xmax]-uniform,[fermat_test(X,N)],Options),
    K1 is K-1,
    itere(N,Xmax,K1).
```

the model, `fermat_test(X,N)` is true iff the equation (1) is satisfied. `itere/3` set up `K` non-primality tests and `uniform` is used to represent a uniform probability distribution.

This declarative view of the primality test allows us to create a new relation for primality testing: we constrain `N` to be a prime number with a certain probability. The goal fails whenever `N` is shown to be a composite number and then the predicate `primal_testing` allows us to generate (likely) prime numbers:

```
?- N in 2..335544431,primal_testing(N,20),labeling([],[N]).

N = 2 ?;
N = 3 ?;
N = 5 ?;
N = 7 ?;
N = 11 ?;
...
yes
```

This toy-example shows that randomized algorithm can be employed to define new relations which are easily implemented within PCC(FD). However, for generating prime numbers, current limitations of the clp(fd) library of SICStus Prolog (namely, the 25-bits limit of the bounds of finite domains) forbid the approach to scale to more interesting cryptographic applications where one looks for 200-digits prime numbers.

4.5 The `choose_decision` combinator

The `choose_decision` combinator implements a probabilistic boolean choice between two processes. This probabilistic boolean choice is represented as a list `[W1,W2]` of two finite domain variables. The term `neg(Constraint)` denotes the negation of `Constraint`. The probabilistic choice arises between `Constraint,Goal1` and `neg(Constraint),Goal2`:

`choose_decision(Constraint,[W1,W2],Goal1,Goal2,Options)`

`no_filtering`, `inconsistency_check` and `lvar(L)` options are available. Note that `choose_decision(Constraint,[W1,W2],Goal1,Goal2,Options)` can be rewritten as `choose(X,[0,1]-[W1,W2],[ask(X=0,[Constraint,Goal1]),ask(X=1,[neg(Constraint),Goal2],Options)])`.

This combinator has been introduced to simulate the behaviour of the conditional and loop statements in imperative programming. It is mainly devoted to the statistical structural testing problem we will discuss later in the next section.

5 Application to the statistical structural testing problem

In this section, we present our modeling of the statistical structural testing problem as a PCCP(FD) problem. We start by recalling some background on this software testing problem.

5.1 Background

Structural software testing aims at increasing our trust in the correctness of a given (imperative) program. Major difficulties reside in an automatic selection of a test suite which is a representativ sampling of program under test behaviours. When the test suite is randomly generated according to a given probability distribution, one speaks of statistical structural testing (SST) [20].

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 [21]. The all_paths criterion aims at covering all paths of the control flow graph. As an example, consider the all_paths criterion, the goal of statistical structural testing for all_paths 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. This problem was initially studied by Thv́enod-Fosse and Waeselynck [20] and more recently Gouraud,Denis, Gaudel and Marre proposed new automated solutions based on combinatorial structures [8].

5.2 A constraint-based approach to the statistical structural testing problem

To address the problem of deriving an input probability distribution from the program structure, we propose translating it into a PCC(FD) program and solve a given request generated from the static source code analysis. Before explaining the translation and the generation of the request, we start by giving an overview of the overall approach.

5.2.1 An overview of the approach

Our approach is based on a two-step process. Firstly the imperative program is translated into a PCC(FD) program. Each statement is inductively translated into a primitive constraint or a constraint combinator. For this translation, we followed the scheme proposed in [7] which makes use of the static single assignment form (SSA form) [3]. During the translation free finite domain variables are inserted in the decisions of the program in order to capture probabilistic information associated to the statistical structural testing problem. Secondly, a static analysis of the imperative program leads to generate a set of structural constraints on the free variables of the probabilistic distribution. Solving a PCC(FD) request that set up these constraints as well as the constraints of the translation allows us to generate a fully instantiated probability distribution that solve the problem. The most interesting point is that our approach exploits static as well as dynamic information for finding the probability distribution, while other statistical structural testing approaches don't.

5.2.2 Static Single Assignment form

The SSA form is a version of a program where each variables have a unique definition and each use of the same variable can be reached by this definition [3]. The SSA form of a basic block is obtained by a simple renaming ($i = i + 1$ leads to $i_2 = i_1 + 1$). For the

Figure 2 SSA form of control statements

if $(x < 4)$	**if** $(x < 4)$
$\quad u = 10$;	$\quad u_1 = 10$;
else	**else**
$\quad u = 2$;	$\quad u_2 = 2$;
	$u_3 = \phi(u_1, u_2)$;
$j = 1$;	$j_1 = 1$;
	/* Heading - while */
	$j_3 = \phi(j_1, j_2)$;
while $(j * u > 16)$	**while** $(j_3 * u_3 > 16)$
$\quad j = j + 1$;	$\quad j_2 = j_3 + 1$;

control structures, SSA form introduces special assignments, called ϕ-functions, to merge several definitions of the same variable. For example, the SSA form of the `if_then_else` is illustrated in the top of Fig. 2. The ϕ-function of the statement $u_3 = \phi(u_1, u_2)$ returns one of its arguments : if the flow comes from the *then*- part then the ϕ-function returns u_1, otherwise u_2.
For other control structures such as loops, the ϕ-functions are introduced in a special heading, as exemplify in Fig. 2. If the flow comes from the statement j_1 =..., then the ϕ-function returns j_1. On the contrary, if the flow comes from the body of the loop (j_2 =...) then, the ϕ-function returns j_2. The SSA form allows the statements to be interpreted as constraints or combinators [7].

5.2.3 Translation of the imperative program

Assignment statement. The statement $x := expr$ is translated into `X#=E` where `E` is the syntactic translation of *expr*.

Compound statement. The statement $Stmt_1; Stmt_2$ is translated into a conjunction (,) of two goals where `Goal1` (resp. `Goal2`) is the translation of $Stmt_1$ (resp. $Stmt_2$).

Conditional statement. The statement **if** b **then** $Stmt_1$ **else** $Stmt_2$ is translated into `choose_decision(C,[W1,W2],Goal1,Goal2,Options)` where `C` is the syntactic translation of b, `Goal1` (resp. `Goal2`) is the translation of $Stmt_1$ (resp. $Stmt_2$). `W1` and `W2` are free variables that are associated to the probability distribution. They will be constrained later.

Loop statement. The statement **while** b **do** $Stmt$ is treated as a conditional statement with a recursive call. In our translation, the statement **while** b **do** *Stmt* is considered as the statement **if** b **then** (*Stmt*;**while** b **do** *Stmt*). In other words, the loop statement obeys to a lazy unfolding in our current framework. Note however that we plan to perform a better translation by making use of a global combinator to simulate the behaviour of the while statement. But this is outside of the scope of the report.

5.2.4 PCC(FD) request generation

By a static analysis of the program under test, some structural constraints on the expected probability distribution can be generated. These constraints are based on the selected structural criterion. More precisely, counting the number of simple paths in the control flow graph of the program allows us to set up some constraints on the probability distribution. For example, when analysing a conditional, by computing the number of simple control flow paths starting from each branches, say n_1 and n_2, we can set up the following constraint $n_1 \cdot W_1 = n_2 \cdot W_2$ if W_1, W_2 are the variables associated to the unknown probability distribution. This process is fully explained in [14] and it illustrated below.

5.3 Examples

We illustrate our approach on two (academic) examples to show its expressiveness.

5.3.1 First example

The first program is given in Fig.3. Some statements are not shown just to allow the reader to focus only on the relevant part of the program. To make things clearer, we assume that

Figure 3 Program foo

int foo(**int** x, **int** y) {
1. **if** ($x =< 100$ && $y =< 100$)
{
2. **if** ($y > x + 50$)
3. ...
4. **if** ($x * y < 100$)
5. ...
6. }

x, y are constrained to take their values in $[0, 1000]$. By using our constraint-based approach, the foo program is automatically translated into the following PCCP(FD) program:

```
foo(X,Y, [W1,W2,W3,W4,W5,W6]) :-
 X in 0..1000,Y in 0..1000,
```

```
choose_decision(X#=<100#/\Y#=<100,[W1,W2],
  [choose_decision(Y#>X+50,[W3,W4],[],Options),
   choose_decision(Y*X#<100,[W5,W6],[],Options)],Options).
```

By a structural analysis, the following relations can be extracted:

```
N1*W1#=N2*W2,N3*W3#=N4*W4,N5*W5#=N6*W6.
```

Given the criterion all_paths, statistical structural testing aims at generating test data such that each path of the program has the same probability to be activated. The paths of the foo program are : $1-2-3-4-5-6$, $1-2-3-5-6$, $1-2-4-5-6$, $1-2-4-6$ and $1-6$. As four paths activate the branch $1-2$ and only one activate the branch $1-6$, the constraint `4*W1#=W2` is generated. As two paths activate the branch $2-3$ and $2-4$, then the constraint `2*W3#=2*W4` is generated the same way. Finally, the following request is obtained:

```
?- foo(X,Y,[W1,W2,W3,W4,W5,W6]),4*W1#=W2,2*W3#=2*W4,2*W5#=2*W6.
```

This request corresponds to the constraint propagation step. When one wants to get test data, it suffices to launch a randomized labelling process on X, Y.

By computing the all terminal configurations of the PCCP(FD) request, we get:

```
? - ptc([foo(X,Y,[W1,W2,W3,W4,W5,W6]),4*W1#=W2,2*W3#=2*W4,2*W5#=2*W6)],
                                                          [X,Y],Result).

Result =
 [([X in 0..1,  Y in 51..100],0.213), % path 1-2-3-4-5-6
 ([X in 1..49,  Y in 52..100],0.196), % path 1-2-3-4-6
 ([X in 0..100, Y in 0..100], 0.200), % path 1-3-5-6
 ([X in 1..100, Y in 1..100], 0.195), % path 1-3-4-6
 ([X in 0..1000,Y in 0..1000],0.196)] % path 1-6
```

The distinct constraint store projections correspond to each path of the program[1]. The results of `ptc` are not far from the $\frac{1}{5}$-probability theoretic expected result for each path.

5.3.2 Second example

Secondly, we applied our constraint-based approach of the statistical structural testing problem to a well-established program of the software testing literature. The program trityp, initially proposed by Myers [13] and fully studied by DeMillo and Offut [4], 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.

[1] In this example, all the paths are feasible

Figure 4 Program trityp

```
int trityp(int i, int j,int k)
if ( (i == 0) or (j == 0) or (k == 0))
trityp = 4 ;
else
    trityp = 0 ;
    if ( i == j)
        trityp = trityp + 1 ;
    if ( i == k)
        trityp = trityp + 2 ;
    if ( j == k )
        trityp = trityp + 3 ;
    if (trityp == 0)
        if ( (i+j <= k) or (j+k <= i) or (i+k <= j))
            trityp = 4 ;
        else
            trityp = 1 ;
    else
        if (trityp > 3)
            trityp = 3 ;
            else
        if ( (trityp == 1) and (i+j > k) )
            trityp = 2 ;
        else
            if ( (trityp == 2) and (i+k > j) )
                trityp = 2 ;
            else
                    if ( (trityp == 3) and (j+k > i))
                        trityp = 2 ;
                    else
                        trityp = 4 ;
return(trityp) ;
```

Although it implements a very simple specification, this program is difficult to handle for test data generators as it contains several nested conditonals structures and a lot of non-feasible paths (43 over a total of 57 paths in the published 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. The program trityp is given in Fig.4.

Figure 5 The PCCP(FD) program generated for trityp

```
trityp(I,J,K,[W1,W2,W3,W4,W5,W6,W7,W8,W9,W10,W11,W12,W13,W14,W15,W16,
                                                       W17,W18,W19,W20]):-
  choose_decision((I#=0)#\/(J#=0)#\/(K#=0),[W1,W2],[T0#=4,T20#=T0],Goal2,Options),
  Goal2 = [choose_decision(I#=J,[W3,W4],[T1#=T0+1, T2#=T1],[T2#=T0],Options),
         choose_decision(I#=K,[W5,W6],[T3#=T2+2, T4#=T3],[T4#=T2],Options),
         choose_decision(J#=K,[W7,W8],[T5#=T4+3, T6#=T5],[T6#=T4],Options),
         choose_decision(T6#=0,[W9,W10],Goal3,Goal4,Options)],
  Goal3 = [choose_decision((I+J#=<K)#\/(J+K#=<I)#\/(I+K#=<J),[W11,W12],
                              [T7#=4, T9#=T7],[T8#=1,T9#=T8],Options),T19#=T9],
  Goal4 = [choose_decision(T6#>3,[W13,W14],
                                    [T10#=3,T18#=T10],Goal5,Options),T19#=18],
  Goal5 = [choose_decision((T6#=1)#\/(I+J#>K),[W15,W16],
                                   [T11#=2,T18#=T11],Goal6,Options),T18#=T17],
  Goal6 = [choose_decision((T6#=2)#\/(I+K#>J),[W17,W18],
                                   [T12#=2,T17#=T12],Goal7,Options), T17#=T16],
  Goal7 = [choose_decision((T6#=3)#\/(J+K#>I),[W19,W20],
                        [T13#=2,T15#=T13],[T13#=4,T15#=T14],Options), T16#=15].
```

Fig. 5 shows the translation of program trityp into a PCCP(FD) program.

By a static analysis and by choosing the all_paths criterion, we get the following constraints for the weights associated to each probabilistic choice.

```
56*W1#=W2,W3#=W4,W5#=W6,W7#=W8,5*W9#=2*W10,W11#=W12,
4*W13#=W14,3*W15#=W16,2*W17#=W18,W19#=W20
```

In fact, 56 paths activate the branch $1-3$ whereas only a single path activates the branch $1-2$. Hence, the constraint `56*W1#=W2` is generated. These analyses are repeated until all the variables were constrained. Note however that in trityp program, this analysis does not allow a uniform activation of (feasible) paths of the program. The uniform coverage is biased by the existence of non-feasible paths. There are indeed only 13 feasible paths which activate the branch $1-3$. Therefore, the path $1-2-29$ is more activated than other paths in the program.

The `choose_decision` combinator we introduced allows us to deal with non-feasible paths. Non-feasible paths cannot be all statically detected but constraint reasoning as employed in our approach permits to early detect some of them. Non-feasible paths correspond to inconsistent constraint stores. Moreover, this information is available during the computation of the unknown probability distribution which is a great advantage. However, dealing efficiently with loops (without systematic unfolding) remains a challenge for our approach to scale up. The first experimental results we got are encouraging but the class of programs being addressed is too small to deduce anything from them. Nevertheless, the constraint-based approach we introduce here for the statistical structural testing problem is very promising and requires to be fully studied.

6 Future work

This report has introduced an extension of PCCP where the probabilistic choice operator is considered as a constraint combinator over finite domains. An implementation under the form of a SICStus Prolog library has been built over clp(fd) and it has been used to address some academic applications and one real-world application, namely the statistical structural problem. We are actually working on increasing the deductive capacity of the combinators by exploiting partial information on the probabilistic choice more efficiently. Soon, we will propose an interface to allow the user to declare there own probabilistic choice combinators by providing facilities to make random draws when only partial information on probability distribution is available. Finally, we will study how to extend these combinators to deal with uncertainty in stochastic processes, such as random walks or Markov processes.

References

[1] N. Angelopoulos, Di Pierro A., and Wiklicky H. Implementing randomised algorithms in constraint logic programming. In *Joint International Conference and Symposium on Logic Programming*, Manchester, UK, 1998.

[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] R. Cytron, J. Ferrante, B.K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Language and Systems*, 13(4):451–490, Oct. 1991.

[4] 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.

[5] A. Di Pierro and H. Wiklicky. On probabilistic ccp. In *APPIA-GULP-PRODE*, pages 225–234, Grado, Italy, 1997.

[6] A. Di Pierro and H. Wiklicky. Implementing randomised algorithms in constraint logic programming. *Proceedings of the ERCIM/Compulog Workshop on Constraints*, 2000.

[7] 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.

[8] 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.

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

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

[11] P. Van Hentenryck, V. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). Technical Report CS-93-02, Brown University, 1993.

[12] S. Janson and S. Haridi. Programming paradigms of the Andorra kernel language. In *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991.

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

[14] 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.

[15] 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.

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

[17] V.A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.

[18] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proceedings of Symposium on Principles of Programming Languages*, pages 333–352, Orlando, Florida, 1991.

[19] G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[20] 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.

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

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399