# Constraint-Based Reasoning on Probabilistic Choice Operators

Matthieu Petit, Arnaud Gotlieb

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

# *Constraint-Based Reasoning on Probabilistic Choice Operators*

Matthieu Petit and Arnaud Gotlieb

**N° 6165**

Thème SYM


ISSN 0249-6399 ISRN INRIA/RR--6165--FR+ENG

# Constraint-Based Reasoning on Probabilistic Choice Operators

Matthieu Petit* and Arnaud Gotlieb

Thème SYM — Systèmes symboliques
Projet Lande

Rapport de recherche n° 6165 — — 21 pages

**Abstract:** Probabilistic Concurrent Constraint Programming (PCCP) extends concurrent constraint languages with a probabilistic choice operator. This operator has proved to be useful in implementing randomized algorithms as well as stochastic processes. In this report, we present a filtering algorithm dedicated to the probabilistic choice operator which permits to address new kind of applications where probabilistic choices are partially known. This filtering algorithm helps to deduce information on the possible values of the probabilistic choice without requiring its full valuation. An implementation under the form of a library of SICStus Prolog is presented.

**Key-words:** Probabilistic Concurrent Constraint Programming, Filtering Algorithm

* 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

# Une approche  contrainte pour raisonner avec un choix probabiliste partiellement connu (version tendue

**Résumé :** La Programmation Concurrente par Contraintes Probabilistes (PCCP) tend la Programmation Concurrente par Contraintes par un oprateur de choix probabiliste. Cet oprateur a prouv son utilit par l'implantation d'algorithmes "randomiss" ainsi que dans la modlisation de processus stochastiques. Dans ce rapport, nous prsentons un nouvel algorithme de filtrage ddi  l'oprateur de choix probabiliste nous permettant de modliser de nouvelles applications pour lesquelles les choix probabilistes ne sont que partiellement connus. Cet algorithme de filtrage nous permet d'obtenir de l'information sur les valeurs pouvant tre prises par le choix probabilistes tout en n'ayant qu'une connaissance partielle sur celui-ci. Une implantation prenant la forme d'une nouvelle librairie de SICStus Prolog est dcrite.

**Mots-clés :** Programmation Concurrente par Contraintes Probabilistes, Algorithme de Filtrage

# 1 Introduction

Probabilistic Concurrent Constraint Programming (PCCP) was concurrently introduced by Di Pierro and Wiklicky [3] and Gupta, Jagadeesan and Saraswat [6] to model randomized algorithms [4] and stochastic processes [5] in a declarative way. In both cases, a probabilistic choice operator was added to Concurrent Constraint Programming (CCP) [12] 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, a conjunction of constraints on the possible values of variables. The computational model acts by accumulating constraints into the constraint store and processes communicate by checking whether the store entails some constraints. In [3], the classical non–deterministic choice operator [13] of CCP is redefined as a probabilistic choice operator whereas [6] introduces an operator that constrains an internal random variable. In [3], random choices operate over processes whereas in [6] random choices operate over values of variables. It is worth noticing that for both operators, random draws shall be performed on an instantiated probability distribution. According to our knowledge, only a single implementation of PCCP was realized under the form of a meta-interpreter in Prolog [1].

In our work, we propose to extend the probabilistic choice operator introduced in [6] to deal with probability distribution only partially known. Our approach is based on the possibility of expressing probabilistic choice relations without requiring all the parameters to be instantiated and deducing information on it during the constraint solving. In particular, we allow expressing random draws over an unknown probability distribution. This increases the declarativity of the probabilistic choice operator and open the door to model new kind of problems. As a very basic example, consider the simulation of a dice draw. When the dice is unbiased (its probability distribution is then fully instantiated), the following PCCP process simulates the dice draw:

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

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 becomes non-feasible whenever the bias of the dice is unknown or just partially 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 the probability distribution of its probabilistic operator to be fully instantiated in order to simulate the draw. By defining the probabilistic choice operator as a constraint, such a problem can be simulated via the following request:

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

The major contribution of this work is to propose a filtering algorithm dedicated to the probabilistic choice operator to reason with probabilistic choices partially known. This

permits us to deal with new kind of applications where probabilistic choices are partially known. We implemented several probabilistic choice operators under the form of a SICStus Prolog library called PCC(FD) built over `clp(fd)` (a.k.a. finite domains constraints) and first experimental result is presented. A version of this library is available online [11].

The report is organized as follows : Section 2 briefly recalls the syntax and semantics of PCCP; Section 3 describes the operator with probabilistic choice partially known. While Section 4 presents the filtering algorithm associated to the operator, section 5 details the implementation of several probabilistic constraint choice operators in SICStus Prolog. 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 [12].

## 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 [13], which is a set of primitive constraints and an entailment relation. The syntax of a CCP language is given by the following grammar:

$$\begin{array}{ll} Program ::= & Declaration.Process \\ Declaration ::= & \varepsilon \mid Declaration.Declaration \mid \mathrm{p}(X) : -Process \\ Process ::= & tell(C) \mid if\ C\ then\ Process \mid new\ X\ in\ Process \\ & \mid Process \| Process \mid p(X) \end{array}$$

A PCCP program is composed of procedure declarations and a process. A process is build on the classical operators: $tell(C)$ 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, $\|$ represents the parallel composition that can be interpreted as a logical conjunction in a Logic Programming environment, and finally, $p(X)$ represents a procedure call. Well known examples of CCP languages include cc(FD)[7], AKL[9] and Oz/Mozart[14] just to name a few.

## 2.2 Probabilistic Choice Operator

In [6], 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 $v_i$. The scope of $X$ is limited 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 $pr_i$ where $Process_{X \leftarrow v_i}$ denotes the concurrent process $Process$ where $X$ has been substituted by $v$ and $pr_i$ denotes the probability of the event $X = v_i$ which is computed by the following formulae:

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

### 2.3 Operational semantics

Running processes of PCCP can be formally described by using a probabilistic transitions system $(\Gamma, \mapsto)$ where $\Gamma$ 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 $\sigma$ is the constraint store. As usual, the transition relation $\mapsto$ is defined with the help of axiomatic rules that are available in [5]. Let $\sigma_0$ be an initial constraint store, then the set of terminal configurations $tc(Process, \sigma_0)$ in the operational semantics of PCCP are defined as:

$$tc(Process, \sigma_0) = \{\sigma \mid \langle Process, \sigma_0 \rangle \mapsto^* \sigma \not\mapsto\}.$$

where $\mapsto^*$ 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:

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

Just to make things more concrete, we illustrate the processing of a PCCP request on a basic example extracted from [5]:

**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 PCCP in a constraint solver over finite domains

Before switching on the description of the probabilistic choice partially known, we presents in this section the introduction of PCCP in a constraint solver over finite domains. As the Concurrent Constraint framework is generally considered as an extension CLP scheme [8], we decides to translate a PCCP process into a CLP($\mathcal{X}$) goal, more precisely we restricts

our approach to finite domain constraints. We decide to restrict our work to finite domains because of the probabilistic choice operates over a finite domain, noted $val(X)$. We also assume that $[v_1, \ldots, v_n]$ is ordered set.

We supposed that previously the procedure declarations have been translated into predicate declarations. Then, the translation is obtained by a translation of each PCCP operator into CLP($\mathcal{FD}$) goals or constraint. The translation is given by FIG.1.

**Figure 1** Translation of PCCP process into a CLP($\mathcal{FD}$) goal

| | |
|---|---|
| $(tell)$ | $tell(C) \rightarrow C$ |
| $(if\ then)$ | $\dfrac{Process \rightarrow Goal}{if\ C\ then\ Process \rightarrow ask(C, Goal)}$ |
| $(new\ in)$ | $\dfrac{Process \rightarrow Goal}{new\ X\ in\ Process \rightarrow Goal}$ |
| $(\|\|)$ | $\dfrac{Process_1 \rightarrow Goal_1 \text{ and } Process_2 \rightarrow Goal_2}{Process_1 \| \| Process_2 \rightarrow Goal_1, Goal_2}$ |
| $(choose)$ | $\dfrac{Process \rightarrow Goal}{choose(X, LawX, Process) \rightarrow choose(X, LawX, Goal)}$ |
| $(proc)$ | $p(x) \rightarrow \mathrm{p}(x)$ |

The meaning of $\rightarrow$ is " is translated into". The operator $tell(C)$ is translated into the constraint $C$. The implication operator $ask$ introduced by P. Van Hentenryck et al. in [7] is used to model the operator of synchronisation $if\ then$. We do not modelled the introduction of a local variable $X$ in a $Process$. It supposes that $X$ is only used in $Process$. The parallel composition operator $||$ is considered as a logical conjunction represented by ",". The *choose* operator is translated into a new constraint of CLP($\mathcal{FD}$) solver. This new constraint is more detailed in the section 4. A procedure call is considered as a predicate call.

This translation aims at using the CLP($\mathcal{FD}$) constraint solving engine to simulate the behaviour of a PCCP process. As find solutions of CLP($\mathcal{FD}$) problem is a decidable but NP-complete problem, a constraint propagation process based on domain and interval reasoning is used to deduce information on the problem during a $Process$ run. During constraint propagation, the constraint is fallen asleep if only a partial solution is found. The constraint solver can be seen as a scheduler. Indeed, constraints are incrementally introduced into a propagation queue. A fix point algorithm manages each constraint one by one into this queue by filtering the domains of finite domain variables of their inconsistent values. Filtering algorithms will consider only the bounds of the domains to eliminate inconsistent values. When the domain of a variable is pruned then the algorithm reintroduces in the queue the

asleep constraints. The algorithm iterates until the queue becomes empty, which corresponds to a state where no more pruning can be performed (a fix point). The interval propagation process reaches a fix point because only a finite number of values can be removed from the domains. This fix point is a conservative approximation (intervals) of the possible values for the variables.

Note that the mechanism of backtracking is avoided when a process failure takes place. Indeed, a process failure is considered as an programming or modelling error in the PCCP. More precisely, it is supposed that when a choice is made, it is the programmer's responsibility to ensure that it is the good one. Thus PCCP languages provide "don't care nondeterminism" if more than one choice is enabled, we don't care which of the corresponding rules are used since will be correct.

# 4 An operator with a probabilistic choice partially known

In PCCP, the probabilistic choice operator requires the pair *Domain Distribution* to be fully instantiated. In this section, we relax this requirement by introducing the probabilistic choice operator *choose* with a probability distribution only partially known. This partial knowledge is represented by constraints on variables of the probability distribution.

## 4.1 Partial knowledge on a probabilistic law

The simulation (random draw) of values for the random variable $X$ is always possible when $Law_X$ is fully determined. A partial knowledge of $Law_X$ is represented by a list a finite domain variables for the distribution probability. Then, partial knowledge on the probability distribution only appears whenever the variables of the probability distribution can take several possible values. This partial knowledge is characterized by the following definition:

**Definition 1.** *Let $X$ be a random variable, $Law_X = [v_1, \ldots, v_n] - [W_1, \ldots, W_n]$ its probability law where $W_1, \ldots, W_n$ are finite domain variables. The set of the possible probability distributions associated to $X$ is:*

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

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

**Example 2.** Consider again the example of dice

$$\begin{array}{l} tell(W_6 = 2 * W_1) \ || \\ choose(X, [1,2,3,4,5,6] - [W_1, W_2, W_3, W_4, W_5, W_6], tell(X = Dice)) \end{array}$$

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 partial knowledge on the unknown bias

of the dice is given by the following set:

$$\begin{aligned}\mathcal{SL}_X = \{ \quad & [1,2,3,4,5,6]-[1,2,2,2,2,2], [1,2,3,4,5,6]-[1,2,2,2,2,3],\\ & [1,2,3,4,5,6]-[1,2,2,2,2,4], [1,2,3,4,5,6]-[2,2,2,2,2,2],\\ & [1,2,3,4,5,6]-[2,2,2,2,2,3], [1,2,3,4,5,6]-[2,2,2,2,2,4]\}.\end{aligned}$$

The operator $choose(X, [v_1, \ldots, v_n] - [W_1, \ldots, W_n], Process)$ builds a relation between the simulation of values for $X$ and the set of finite domain variables of its probabilistic law that can be exploited to filter the domain of random variable $X$ during the constraint propagation.

### 4.2 Constraint propagation on *choose*

The constraint *choose* succeeds whenever $X$ is valuated and $Process$ succeeds. When the probabilistic choice is only partially known, the constraint *choose* is introduced into the propagation queue of the constraint solver and then, a filtering algorithm is launched to prune $val(X)$. When no more pruning can be performed, the constraint falls asleep. The constraint is awoken whenever the domain of at least one variable of the distribution probability is modified and it is reintroduced in the propagation queue. This process iterates until a fix point is reached, i.e. a state where no more deduction on the domain of $X$ is obtained.

## 5 The *choose* filtering algorithm

As usual, the behaviour of random variables over a finite domain is simulated with the values of a uniform random variable over $[0; 1]$, noted $U$. The algorithm exploits an *a priori* random value of $U$ to prune $val(X)$. Given a value for $U$, we reason of the element of $\mathcal{SL}_X$ to deduce information on $val(X)$.

### 5.1 Simulation of a random variable with a finite probability law

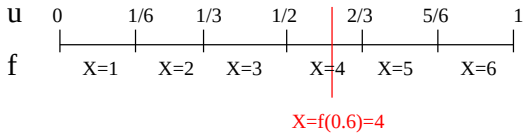
The value of a random variable with a finite law is generated from the value of a uniform random variable $U$ over $[0; 1]$. The link between the values of $X$ and $U$ is established by the distribution function f as follows.

$$\begin{array}{rccl} f: & [0;1] & \rightarrow & \{v_1, \ldots, v_n\} \\ & u & \mapsto & \begin{cases} v_1 & \text{if } u \in [0, pr_1[ \\ \vdots & \vdots & \\ v_n & \text{if } u \in \left[\sum_{j=1}^{n-1} pr_j, 1\right[ \end{cases} \end{array} . \tag{1}$$

It is trivial to see that the probability of the event $X = v_i$ is equal to $P(u \in [pr_1 + \ldots + pr_{i-1}, pr_1 + \ldots + pr_i]) = (pr_1 + \ldots + pr_i) - (pr_1 + \ldots + pr_{i-1}) = pr_i$, as expected.

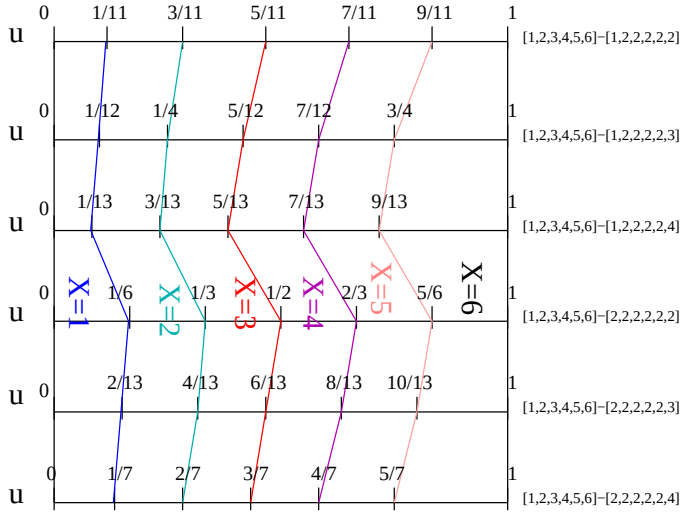
As an example, consider the distribution function $f$ associated to an unbiased dice given by FIG. 2. Here, the distribution probability is represented by $[1,2,3,4,5,6] - [1,1,1,1,1,1]$.Let 0.6 be a random value generated for $U$, then we get $X = 4$.

**Figure 2** Simulation of the random variable $X$ with a uniform probability distribution on $\{1,2,3,4,5,6\}$




The set of distribution functions is noted $\mathcal{F}_X$. Note that for each element of $\mathcal{SL}_X$ corresponds a distribution function of $\mathcal{F}_X$. For example, the set $\mathcal{F}_X$ of the example 2 dice is represented by FIG.3.

**Figure 3** Set of the probability distributions of the example 2 dice




## 5.2 Properties of the *choose* constraint

In this section, we propose to characterize correction properties of any filtering algorithm on the *choose* constraint. Our filtering algorithm exploits information on the probabilistic

choice, i.e the value $u$ of $U$ the random variable associated to the simulation of values for $X$ and $\mathcal{SL}_X$, to prune $val(X)$. Then, these correction properties are dependent on the value of $u$. Given $u$ known, the first one states that each element of $val(X)$ can be randomly choosen, i.e it exists a distribution function such as this element can be chosen.

**Property 1.** *[Correction of choose]*
*Let $X$ be a random variable with a probability distribution law $[v_1, \ldots, v_n] - [W_1, \ldots, W_N]$ and $u$ the value of the random variable $U$ associated to the simulation of values for $X$. Let consider $\mathcal{F}_X$ as the set of probability distribution. Then, property 1 states that*

$$\forall v_i \in val(X), \exists f \in \mathcal{F}_X \text{ such that } f(u) = v_i$$

Given $u$, the second one states that the bounds of $val(X)$ can be randomly choosen.

**Property 2.** *[Partial Correction of choose]*
*Let us know that $v_{min} = \min(val(X))$ and $v_{max} = \max(val(X))$, then property 2 states that*

$$\exists f_1 \in \mathcal{F}_X \text{ such that } f_1(u) = v_{min} \text{ and } \exists f_2 \in \mathcal{F}_X \text{ such that } f_2(u) = v_{max}$$

From the definitions, it is clear that property 1 entails the property 2.

## 5.3 Filtering on the domain of the possible values for $X$

Given a value $u$ for $U$, the principle of the filtering algorithm is to detect values for $X$ than cannot be randomly chosen by reasoning on the set of the possible probability distributions. In this section, it assumes that $[v_1, \ldots, v_n]$ the domain associated to the probabilistic choice is sorted.

### 5.3.1 The algorithm

To detect that a value $v_i$ can not be randomly chosen, we reason on $\mathcal{F}_X$ and then on $\mathcal{SL}_X$. In fact, we reason on the range of values for $u$ associated to the event $X = v_i$ for each distribution function of $\mathcal{F}_X$. This set of ranges defines the set of values for $u$ such that $v_i$ can be randomly chosen. More formally, this set is defined as follows:

$$\forall v_i \in [v_1, \ldots, v_n],\ Int_{v_i} = \bigcup_{[v_1 \ldots, v_n] - [w_1, \ldots, w_n] \in \mathcal{SL}_X} \left[ \frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j},\ \frac{\sum_{j=1}^{i} w_j}{\sum_{j=1}^{n} w_j} \right[ \quad (2)$$

If $u$ does not belong into $Int_{v_i}$, then $v_i$ cannot be randomly chosen for $X$. However, the computation of $Int_{v_i}$ is based on the enumeration of each element $\mathcal{SL}_X$. In the worst case, enumerating each element of $\mathcal{SL}_X$ is equivalent to enumerate each element of the cartesian product of the weights domain. When the size of the domains of the weights increase, this computation becomes intractable. We decides to approximate the computation of $Int_v$ in computing its bounds.

**Algorithme 1** : FILTERALGO

**Input** : $X$,$Law_X$ and $u$

**forall** $v \in Val(X)$ **do**
  $[Min\_Int_v; Max\_Int_v] \leftarrow$ `IntProba`$(v, Law_X)$;
  **if** $u \notin [Min\_Int_v; Max\_Int_v]$ **then**
    $v$ is removed from $Val(X)$
  **end**
**end**

The filtering algorithm takes as inputs $X$, $Law_X$ and $u$ and removes values of $val(X)$ which cannot randomly chosen. For each value $v \in val(X)$, the bounds of $Int_v$, $Min\_Int_v$ and $Max\_Int_v$ is computed by INTPROBA. If $u \notin [Min\_Int_v; Max\_Int_v]$, $v$ is removed from $val(X)$.

The efficiency of filtering algorithm is due to our efficiency to compute the bounds of each $Int_{v_i}$. Hopefully, theses bounds are obtained by an analytical result.

### 5.3.2 An efficient method to compute an approximation of $Int_{v_i}$

The method is looking for the element of $\mathcal{SL}_X$ such that $Int_{v_i}$ is minimizes and maximizes. It is based on the computation of the minimum (resp. maximum) value of $\frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j}$ (resp. $\frac{\sum_{j=1}^{i} w_j}{\sum_{j=1}^{n} w_j}$) for each of $\mathcal{SL}_X$. A efficient way to compute $\min_{[v_1...,v_n]-[w_1,...,w_n]\in\mathcal{SL}_X}\left(\frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j}\right)$ and $\max_{[v_1...,v_n]-[w_1,...,w_n]\in\mathcal{SL}_X}\left(\frac{\sum_{j=1}^{i} w_j}{\sum_{j=1}^{n} w_j}\right)$ is given by the two analytical results:

Let $X$ be a random variable and its probabilistic law $[v_1, \ldots, v_n] - [W_1, \ldots, W_N]$. Let us assume that $\sum_{j=1}^{n} min(W_j) > 0$. Then,

$\forall v_i \in [v_1, \ldots, v_n]$,

$$\min_{[v_1...,v_n]-[w_1,...,w_n]\in\mathcal{SL}_X}\left(\frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j}\right) = \frac{\sum_{j=1}^{i-1}\min(W_j)}{\sum_{j=1}^{i-1}\min(W_j)+\sum_{j=i}^{n}\max(W_j)} \quad (3)$$

and

$\forall v_i \in [v_1, \ldots, v_n]$,

$$\max_{[v_1...,v_n]-[w_1,...,w_n]\in\mathcal{SL}_X}\left(\frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j}\right) = \frac{\sum_{j=1}^{i}\max(W_j)}{\sum_{j=1}^{i}\max(W_j)+\sum_{j=i+1}^{n}\max(W_j)} \quad (4)$$

*Proof.* Proof by refutation. Suppose that $\exists\, [v_1, \ldots, v_n] - [w_1, \ldots, w_n] \in \mathcal{SL}_X$ such as

$$\frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j} < \frac{\sum_{j=1}^{i-1}\min(W_j)}{\sum_{j=1}^{i-1}\min(W_j)+\sum_{j=i}^{n}\max(W_j)} \quad (5)$$

By hypothesis, $\sum_{j=1}^{n} min(W_j) > 0$, then $\sum_{j=1}^{n} w_j$ and $\sum_{j=1}^{i-1} \min(W_j) + \sum_{j=i}^{n} \max(W_j)$ are superior to nought. From (5), we can deduce that

$$\sum_{j=1}^{i-1} w_j \cdot \left( \sum_{j=1}^{i-1} \min(W_j) + \sum_{j=i}^{n} \max(W_j) \right) < \sum_{j=1}^{i-1} \min(W_j) \cdot \sum_{j=1}^{n} w_j$$

Then,

$$\sum_{j=1}^{i-1} w_j \cdot \sum_{j=i}^{n} \max(W_j) < \sum_{j=1}^{i-1} \min(W_j) \cdot \sum_{j=i}^{n} w_j$$

But,

$$0 \leq \sum_{j=1}^{i-1} \min(W_j) \leq \sum_{j=1}^{i-1} w_j$$

and

$$0 \leq \sum_{j=i}^{n} w_j \leq \sum_{j=i}^{n} \max(W_j)$$

We obtain a contradiction because of $\sum_{j=1}^{i-1} w_j \cdot \sum_{j=i}^{n} \max(W_j)$ cannot be lesser than $\sum_{j=1}^{i-1} \min(W_j) \cdot \sum_{j=i}^{n} w_j$. Then, we prove that

$$\forall [v_1, \ldots, v_n] - [w_1, \ldots, w_n] \in \mathcal{SL}_X,\ \frac{\sum_{j=1}^{i-1} \min(W_j)}{\sum_{j=1}^{i-1} \min(W_j) + \sum_{j=i}^{n} \max(W_j)} \leq \frac{\sum_{j=1}^{i-1} w_j}{\sum_{j=1}^{n} w_j}$$

The following equality

$$\forall v_i \in Dom(X), \max_{[v_1 \ldots, v_n] - [W_1, \ldots, w_n] \in \mathcal{SL}_X} \left( \sum_{j=1}^{i} pr_j \right) = \frac{\sum_{j=1}^{i} \max(W_j)}{\sum_{j=1}^{i} \max(W_j) + \sum_{j=i+1}^{n} \max(W_j)}$$

can also be proved with the same reasoning. □

### 5.3.3 Termination and correctness

Termination is trivially obtained just by noticing that the set $val(X)$ is finite.

Correction of the filtering is more difficult to state. As an approximation is done during the computation of $Int_{v_i}$, the constraint *choose* is said partially correct after the FILTER-ALGO, i.e. we guarantee that the bound of $Val(X)$ can be randomly chosen after the propagation process.

*Proof.*
**Case 1:** $v_i = v_1$ Let us consider the probability law:

$$[v_1, \ldots, v_{i-1}, v_i, \ldots, v_n] - [\max(W_1), \min(W_2), \ldots, \min(W_n)].$$

The maximum of $Int_{v_1}$ is obtaineed from this probability law. We notes by $f_1$ its distribution function. Let us prove that $f_1(u) = v_0$.

Given the equation 3 and 4, the computation of $Int_{v_0}$ is approximated by $[0;\ \frac{max(W_1)}{\max(W_1)+\sum_{i=2}^{n}\min W_j}\Big[$. As $v_0 \in Val(X)$ after FILTERALGO run, then $u \in [0;\ \frac{max(W_1)}{\max(W_1)+\sum_{i=2}^{n}\min W_j}\Big[$. Then, we can conclude that

$$f_1(u) = v_1$$

**Case 2:** $min(Val(X) = v_n$ Let us consider the probability law:

$$[v_1, \ldots, v_{i-1}, v_i, \ldots, v_n] - [\min(W_1), \ldots, \max(W_n)].$$

The minimum of $Int_{v_n}$ is obtained from this probability law. We notes by $f_1$ its distribution function. Let us prove that $f_1(u) = v_n$.

Given the equation 3 and 4, the computation of $Int_{v_n}$ is approximated by $\Big[\frac{\sum_{j=1}^{n-1}(W_j)}{\sum_{j=1}^{n-1}(W_j)+\max(W_n)};\ 1\Big[$. As $v_n \in Val(X)$ after FILTERALGO run, we can conclude that

$$f_1(u) = v_n$$

**Case 3:** Suppose that $v_i = min(Val(X))$ after a FILTERALGO run and that $v_1 < v_i < v_n$ Let us consider the probability law:

$$[v_1, \ldots, v_{i-1}, v_i, v_{i+1}, \ldots, v_n] - [\max(W_1), \ldots, \max(W_{i-1}), \max(W_i), \min(W_{i+1}), \ldots, \min(W_n)].$$

The maximum of $Int_{v_i}$ is obtained with this probability law. We notes by $f_1$ its distribution function. Let us prove that $f_1(u) = v_i$, i.e. after FILTERALGO run

$$u \in \left[\frac{\sum_{j=1}^{i-1}\max(W_j)}{\sum_{j=1}^{i-1}\max(W_j)+\sum_{j=i}^{n}\min(W_j)};\ \frac{\sum_{j=1}^{i}\max(W_j)}{\sum_{j=1}^{i}\max(W_j)+\sum_{j=i+1}^{n}\min(W_j)}\right[.$$

We obtains immediatly that $u \leq \frac{\sum_{j=1}^{i}\max(W_j)}{\sum_{j=1}^{i}\max(W_j)+\sum_{j=i=1}^{n}\min(W_j)}$ because of $v_i \in Val(X)$ after FILTERALGO run.

Recall that $[v_1, \ldots, v_n]$ is an ordered set. Let suppose that this following affirmation is true.

$$\forall k, j \in \{1, \ldots, n\},\ \text{such as } k < j, \min(Int_{v_k}) \leq \min(Int_{v_j}) \text{ and } \max(Int_{v_k}) \leq \max(Int_{v_j}). \tag{6}$$

This result can be proved in using of inductive reasonning, developping each terms and comparing them. The equations 3 and 4 give the following equalities:

$$\min(Int_{v_{i-1}}) = \frac{\sum_{j=1}^{i-1}\min(W_j)}{\sum_{j=1}^{i-1}\min(W_j)+\sum_{j=i}^{n}\max(W_j)}$$

and

$$\max(Int_{v_{i-1}}) = \frac{\sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i}^{n} \max(W_j)}.$$

As $\forall j$ such as $1 \leq j < i, v_j \notin Val(X)$ after FILTERALGO run because of $v_i$ is the minimum of $Val(X)$. Then, $\forall j$ such as $1 \leq j < i$, $u \notin [\min(Int_{v_j}), \max(Int_{v_j})]$. By extension in using of the result 6, we can deduce that $\max(Int_{v_{i-1}}) < u$. Moreover,

$$\begin{aligned}
&\max(Int_{v_{i-1}}) - \frac{\sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i+1}^{n} \min(W_j)} \\
&= \frac{\sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i}^{n} \min(W_j)} - \frac{\sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i-1}^{n} \min(W_j)} \\
&= \frac{(\max(W_i) - \min(W_i)) \cdot \sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i}^{n} \min(W_j) + \sum_{j=1}^{i} \max(W_j) + \sum_{j=i+1}^{n} \min(W_j)}
\end{aligned}$$

As $(\max(W_i) - \min(W_i)) \cdot \sum_{j=1}^{i-1} \max(W_j)$ and $\sum_{j=1}^{i-1} \max(W_j) + \sum_{j=i}^{n} \min(W_j) + \sum_{j=1}^{i} \max(W_j) + \sum_{j=i+1}^{n} \min(W_j)$ are non negative number, we conclude that

$$\frac{\sum_{j=1}^{i-1} \max(W_j)}{\sum_{j=1}^{i} \max(W_j) + \sum_{j=i+1}^{n} \min(W_j)} \leq \max(Int_{v_{i-1}})$$

Then, we can conclude that

$$u \in \left[ \frac{\sum_{j=1}^{i} -1 \max(W_j)}{\sum_{j=1}^{i} \max(W_j) + \sum_{j=i=1}^{n} \min(W_j)}; \ \frac{\sum_{j=1}^{i} \max(W_j)}{\sum_{j=1}^{i} \max(W_j) + \sum_{j=i=1}^{n} \min(W_j)} \right[$$

and then that

$$f_1(u) = v_i$$

Finally, we have proved that after FILTERALGO run that

$$\exists f_1 \in \mathcal{F}_X \text{ such that } f_1(u) = min(Val(X))$$

In using the same reasonning, we can prove that

$$\exists f_2 \in \mathcal{F}_X \text{ such that } f_2(u) = max(Val(X))$$

□

#### 5.3.4 Example

Consider again the example 2.
The set $\mathcal{F}_X$ is given by FIG. 3. The values of $Int_{v_i}$ can be easily computes.

| | |
|---|---|
| $Int_1 = [0\ ; \frac{1}{6}[$ | $Int_4 = [\frac{5}{13}\ ; \frac{2}{3}[$ |
| $Int_2 = [\frac{1}{13}\ ; \frac{1}{3}[$ | $Int_5 = [\frac{7}{13}\ ; \frac{5}{6}[$ |
| $Int_3 = [\frac{3}{13}\ ; \frac{1}{2}[$ | $Int_6 = [\frac{9}{13}\ ; 1[$ |

$X = 1$ is removed from $dom_u(X)$ by the filtering algorithm when $u \notin [0; \frac{1}{6}[$, $X = 2$ is removed from $dom_u(X)$ when $u \notin [\frac{1}{13}\ ; \frac{1}{3}[$ and so one.

#### 5.3.5 Complexity

The complexity of the pruning algorithm is associated to the complexity of the computation of $Int_{v_i}$. Each bounds of $Int_{v_i}$ is obtained in computing $\forall i \in 1 \ldots n$, $\sum_{j=1}^{i-1} min(W_j)$ and $\sum_{j=1}^{i} max(W_j)$. The result is computed in a linear time in comparaison with the size of $X$ domain. As the rest of the algorithm is restricted to a "membership testing", FILTERALGO runs in $O(n)$ where $n$ is the size of $[v_1, \ldots, v_n]$ the domain of $X$.

A version of the efficient filtering algorithm is implemented for each probabilistic choice operator of the PCC(FD) library.

# 6 The PCC(FD) library

In this section, a library of three probabilistic choice operators defined as new constraints is presented. The implementation of these constraints is based on the clp(FD) library of SICStus prolog [2], by making use of its global constraint definition interface. We presented also a first validation experimental build on a toy example. The translation of PCCP program into *clp*($FD$) program is obtained via the meta-interpreter pccfdtoclpfd wroten in Prolog. Before switching on the description of probabilistic choices, let us just explain the predicate `ptc(Goal,Var_List,Result)` which empirically computes the set of terminal configurations in PCC(FD).

## 6.1 Empirical computation of the set of terminal configurations

Given a Prolog goal `Goal` along with a list of variables `Var_List`, the `ptc` predicates launches a given number 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).

## 6.2 Probabilistic combinators in PCC(FD)

The three new constraints 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 probabilistic choices 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 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.

## 6.3 The `choose` constraint

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

```
choose(X,[v1,..vn]-[W1,..,Wn],Goal,Options)
```

where `W1,..,Wn` are finite domain variables. `Options` is used to parameterize the filtering capacities of the operator 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 ;
- `consistency_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 is less efficient;
- `lvar(L)` can be used to enrich the list of variables on which the constraint is awaked. This option is useful to parameterize the awaking conditions of the combinator.

By default, the filtering algorithm associated to the constraint is the algorithm 1.

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

**Example 3.**
```
dice(Dice) :-
  tell(W1 in 1..2)||tell(W2 in 2..2)||tell(W3 in 2..2)||
  tell(W4 in 2..2)||tell(W5 in 2..2)||tell(W6 in 2..4)||
  tell(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.

## 6.4 The `choose_range` constraint

The `choose_range` constraint implements a probabilistic choice operator for a range of values. The range is given by `[Min,Max]`, where `Min` and `Max` are two finite domain variables. `Min` denotes the lower bound of $X$ wheras `Max` denotes its upper bound. The probability distribution takes the form of a list of weights. Its size evolves with new information on `Min` or `Max`.

```
choose_range(X,[Min,Max]-[W1,..,Wn]-Q,Goal,Options)
```

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

The `choose_range` constraint has been used on an implementation of the (weak) Miller-Rabin primality test. This example shows that randomized algorithm can be employed to define new probabilistic relations which are easily implemented within PCC(FD). The implementation in PCC(FD) allows to constraint a variable to be a prime number given a certain probability. The implementation of the randomized algorithm is available on [11].

## 6.5 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 constraint has been introduced to simulate the behaviour of the conditional and loop statements in imperative programming. Indeed, we used this constraint to address a problem of Software Testing: the statiscal structural testing [15]. The purpose of the statistical structural testing is to randomly generate test data such as the coverage of structural criterion is guaranteed. A first translation of the problem into a probabilistic constraint problem has been proposed in [10]. A new model of the problem which uses the new ability of the probabilistic choice operator to reason on probabilistic choice only partially known is in progress. It seems useful to reason on probabilistic partially known to improve dynamically our problem of statistical test data generation

## 6.6 First experimental validation with the 421 dice playing

The first experimental validation of the filtering algorithm is built on a toy example: 421 dice playing with unknown biased dice. This game consists in drawing three dice. The game is won when 4, 2 and 1 is drawn without taking care of the valuation order. Your game hypothesis is: the person who proposes the dice playing is dishonest. To estimate the luck to win the game, you decide to model the dice bias as follows: the probability to draw 4 is two times lower than the probability to draw 3, the probability to draw 2 is two times lower than the probability to draw 5 and the probability to draw 1 is two times lower than the probability to draw 6. The game is modelled by `four_two_one/3` given by the Fig.4. The predicate `four_two_one([D1,D2,D3],Distribution, Options)` is said true iff the `D1`, `D2` and `D3` have been valued to 4, 2 and 1 without taking care of the valuation order.

**Figure 4** 421 dice playing with bias partially known

```
four_two_one([D1,D2,D3],[W1,W2,W3,W4,W5,W6],Options) :-
       tell(2*W1=W6)||tell(2*W2=W5)||tell(2*W4=W3)||
       choose(X, [1,2,3,4,5,6]-[W1,W2,W3,W4,W5,W6],[D1=X],Options)||
       choose(Y, [1,2,3,4,5,6]-[W1,W2,W3,W4,W5,W6],[D2=Y],Options)||
       choose(W, [1,2,3,4,5,6]-[W1,W2,W3,W4,W5,W6], [D3=W], Options)||
       tell(all_different([D1,D2,D3]))||tell(D1+D2+D3#=7).
```

The experimental validation, named *Exp* is based on the search of solution by propagation and labelling processes of the constraint solver. The experiment is based on a toy request which tries to find all the valuation of `Distribution` such as the game is won.

**Figure 5** Result of *Exp* for $Wmax = 10$, $Wmax = 20$, $Wmax = 30$, $Wmax = 40$ and $Wmax = 50$

| | Meaning of the request run time in ms | | | | |
|---|---|---|---|---|---|
| | $W_{max} = 10$ | $W_{max} = 20$ | $W_{max} = 30$ | $W_{max} = 40$ | $W_{max} = 50$ |
| No filtering | 5.3 | 38.9 | 129.3 | 309.2 | 603 |
| Filtering | 6 | 12.5 | 33.5 | 66.1 | 123.7 |

```
?- four_two_one([D1,D2,D3],Distribution,Options),
   labelling_findall([],Distribution).
```

Note that for sake of clarity, we do not model the labelling process based on the search in a tree in PCCP framework. We used the classical backtracking algorithm to search all the solutions of the problem.

The parameter of experiment is $W_{max}$. It represents the maximal bound of non negative variables `W1,..,W6`. Two requests are compared. The first one uses the option `no_filtering` whereas the second one uses an efficient filtering algorithm. The request is reiterated 50000 times. The number of iteration has been chosen to obtain a significant sampling of request behaviours. The performance has been evalated in using of SICStus 3.11 on a Windows XP machine powered by a 2GHz Intel Pentium processor and 1Go memory.

The result given by Fig.5 allows to expecting that the filtering algorithm permits to increase the request run time. This gain in the time running is linked to the pruning capacity of the probabilistic choice domain of the filtering algorithm. In the worst case, no domain pruning is obtained when the filtering algorithm is used. As the probabilistic choice is frozen when no filtering algorithm is used, overtime can appear. However, this overtime stays reasonable because of the filtering algorithm is efficient.

# 7 Future work

This report has introduced an extension of PCCP which permits to reason on probabilistic choices partially known. In this paper, we proposed a dedicated filtering algorithm of the probabilistic choice operator to deduce information on the possible values of the probabilistic choice. An implementation under the form of a SICStus Prolog library has been built over clp(fd) and has been validated by a first experimental validation. Soon, we will propose to address a real world problem, namely statistical structural testing [15]. The problem aims at finding distribution probability over the input domain of a program that maximizes the coverage of some structural criteria, such as all_statements or all_paths. Our purpose is to model this problem as a probabilistic constraint problem where probabilistic choices are only partially known.

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

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

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

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

[7] 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.

[8] J. Jaffar and M.J. Maher. Constraint logic programming : A survey. *Journal of Logic Programming*, 20(19):503–581, 1994.

[9] 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.

[10] 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.

[11] M Petit and A. Gotlieb. Library of probabilistic constraint combinators (version 0), available at http://www.irisa.fr/lande/petit/tools.html. May 2006.

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

[13] 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.

[14] 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.

[15] 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.

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