Raisonner et filtrer avec un choix probabiliste partiellement connu

Matthieu Petit, Arnaud Gotlieb

Raisonner et filtrer avec un choix probabiliste partiellement connu

Matthieu Petit* **Arnaud Gotlieb**

INRIA-IRISA, Campus Beaulieu, 35042 Rennes Cedex
{petit,gotlieb}@irisa.fr

Résumé

La Programmation Concurrente par Contraintes Probabilistes (PCCP) étend la Programmation Concurrente par Contraintes (CCP) par l'ajout d'un opérateur de choix probabiliste. Cet opérateur permet d'introduire de l'aléa dans l'exécution d'un processus de CCP. Dans cet article, les fonctionnalités de cet opérateur sont étendues afin de raisonner avec un choix probabiliste partiellement connu. Pour cela, nous définissons l'opérateur de choix probabiliste comme un nouveau combinateur de contraintes et lui associons un algorithme de filtrage. Cet algorithme de filtrage permet, la plupart du temps, d'accélérer la propagation de contraintes. L'implantation sous la forme d'une nouvelle librairie d'opérateurs de choix probabilistes pour SICStus Prolog ainsi qu'une validation expérimentale est présentée.

1 Introduction

La Programmation Concurrente par Contraintes Probabiliste (PCCP) a été conjointement introduite par les travaux de Di Pierro et Wiklicky [3] et les travaux de Gupta, Jagadeesan et Saraswat [6]. PCCP est une extension probabiliste de la Programmation Concurrente par Contraintes, permettant d'une part l'implantation d'algorithmes randomisés sous une forme déclarative [4] et d'autre part la modélisation de processus stochastiques [5]. Dans les travaux de Di Pierro et Wiklicky [3], l'opérateur standard de choix non déterministe [12] est remplacé par un choix probabiliste d'exécution de processus tandis que l'opérateur de choix probabiliste des travaux de Gupta, Jagadeesan et Saraswat introduit une variable aléatoire interne à un processus à exécuter. Il est important de noter que dans les deux cas, la distribution de probabilité doit être complètement déterminée, c'est à dire que la probabilité de choisir chaque valeur ou processus doit être connue. À notre connaissance, une seule implantation de PCCP est disponible, sous forme d'un méta interpréteur Prolog [1]. Cette approche implémente uniquement l'opérateur de choix probabiliste des travaux de Di Pierro et Wiklicky [3].

Dans notre travail, nous étendons les fonctionnalités de l'opérateur de choix probabiliste de Gupta, Jagadeesan et Saraswat afin de raisonner et filtrer avec un choix probabiliste partiellement connu. Nous supposons que la loi de probabilité associée à la variable aléatoire peut-être (partiellement) définie par l'intermédiaire de contraintes. Notre objectif est de résoudre de nouveaux problèmes pour lesquels le choix probabiliste est un paramètre partiellement connu du problème. A titre d'exemple, considérons la modélisation d'un jet de dé. Lorsque le dé est non biaisé, le processus PCCP suivant représente le jet :

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

La probabilité associée à chaque valeur possible du dé est la même $\frac{1}{6}$. Elle est représentée par $[1,1,1,1,1,1]$ une liste de poids associés à chaque face du dé. Par contre, lorsque le dé est biaisé et que le biais est exprimé par des contraintes (par exemple, la face 6 du dé est chargée deux fois plus que la face 1), PCCP devient inopérante pour la modélisation du jet. L'extension décrite dans ce papier permet justement de modéliser un tel jet, par l'intermédiaire du processus suivant :

$$tell(P_6 = 2 * P_1) \;||\; choose(X, [1,2,3,4,5,6] - [P_1,P_2,P_3,P_4,P_5,P_6], tell(Dé = X))$$

Notre travail est poussé par la volonté de modéliser et résoudre des problèmes pour lesquels les différents choix probabilistes sont les inconnues du problème. Une application particulièrement pertinente de notre

*Ce travail est financé par la région Bretagne dans le cadre du projet GENETTA : http://www.irisa.fr/lande/petit/genetta.

approche concerne le test automatique de logiciels. En effet, la technique du test statistique structurel [14] conduit à rechercher une distribution de probabilité sur le domaine d'entrée d'un programme qui permet de maximiser la probabilité de couverture structurelle du programme sous test. La distribution de probabilité est contrainte par différentes informations extraites de l'analyse du programme et représente l'inconnue du problème que nous voulons résoudre. C'est pour attaquer de tels problèmes que nous proposons de définir l'opérateur de choix probabiliste comme un combinateur de contraintes capable de raisonner et de filtrer avec un choix probabiliste seulement partiellement connu.

La contribution principale de ce travail est la mise en oeuvre d'un algorithme de filtrage efficace dédié à l'opérateur de choix probabiliste. Le filtrage est effectué sur les valeurs pouvant être prises par le choix probabiliste. L'objectif sous-jacent est l'obtention efficace d'information sur le choix probabiliste lors du mécanisme de propagation. Ceci a pour but d'accélérer la résolution du problème en retardant la recherche d'une instanciation solution de la distribution de probabilité lors du mécanisme d'énumération. Dans certains cas, cet algorithme permet même d'effectuer le choix probabiliste alors que la distribution de probabilité n'est pas complètement instanciée. D'un certain point de vue, cela revient à obtenir la valeur d'un jet de dé alors que le biais du dé n'est pas complètement connu. Dans le cadre d'un résolveur de contraintes à domaines finis, notre étude de complexité assure que cet algorithme s'exécute en temps linéaire par rapport au domaine de variation du choix aléatoire. Une implémentation prenant la forme d'une nouvelle librairie de combinateurs de contraintes du résolveur clp(FD) de SICStus Prolog est présentée ainsi qu'une première validation expérimentale pour un exemple simple : le jeu du 421 avec des dés biaisés de biais inconnu.

Le papier est organisé de la manière suivante : la section 2 rappelle rapidement les fondements de la programmation concurrente par contraintes probabilistes. La section 3 présente l'opérateur de choix probabiliste en tant que combinateur de contraintes. En section 4, l'algorithme de filtrage dédié au combinateur de contraintes est décrit dans le cadre d'un résolveur de contraintes à domaines finis. La section 5 présente la librairie d'opérateurs de choix probabiliste que nous avons implémentée ainsi qu'une première validation expérimentale. La section 6 conclut ce travail et présente quelques perspectives.

2 Programmation concurrente par contraintes probabilistes

2.1 Programmation concurrente par contraintes

Nous commençons par rappeler la syntaxe et la sémantique des différents opérateurs de CCP [11].

Dans CCP, les processus sont exécutés de manière concurrente et intéragissent entre eux par un magasin de contraintes commun. Le langage de CCP est paramétré par un système de contraintes [12] composé d'un ensemble de contraintes primitives et d'une relation de satisfaction. La syntaxe abstraite d'un processus de CCP est donnée par la grammaire suivante :

$$\begin{aligned} Process ::= \quad & tell(C) \mid if\ C\ then\ Process \\ & \mid new\ X\ in\ Process \\ & \mid Process \| Process. \end{aligned}$$

où $tell(C)$ ajoute C au magasin de contraintes, $if\ C\ then\ Process$ ajoute les contraintes de $Process$ au magasin de contraintes si C est impliquée, $new\ X\ in\ Process$ ajoute les contraintes de $Process$ tout en cachant la variable de X des autres processus et pour finir $\|$ représente la composition parallèle et peut être interprété comme une conjonction logique. Comme exemples d'implantation de ce paradigme de programmation, on peut citer cc(FD) [7], AKL [8] ou Oz/Mozart [13].

2.2 Opérateur de choix probabiliste

Dans [6], Gupta et al. ont proposé d'ajouter un opérateur de choix probabiliste à CCP. Cet opérateur, appelé *choose*, introduit une variable aléatoire dans un processus. La variable aléatoire est définie par sa loi de probabilité, i.e son domaine de variation et sa distribution de probabilité, permettant de caractériser le choix probabiliste :

$$choose(X, Loi_X, Process)$$

où X est une variable aléatoire, Loi_X est composée d'une liste des valeurs possibles de la variable aléatoire $[v_1, \ldots, v_n]$ et d'un ensemble de poids positifs $[p_1, \ldots, p_n]$ associés à chaque v_i et $Process$ un processus concurrent de PCCP. La distribution de probabilité $\{pr_i\}_{i \in 1 \ldots n}$ où pr_i est la probabilité de l'évènement $X = v_i$ est calculée par normalisation :

$$pr_i = \frac{p_i}{\sum_{j=1}^{n} p_j}.$$

Dans la suite, $Process_{X \leftarrow v}$ dénote le processus $Process$ pour lequel chaque occurence de X a été remplacée par v. D'un point de vue opérationnel, $choose(X, [v_1, \ldots, v_n] - [p_1, \ldots, p_n], Process)$ signifie

que le processus $Process_{X \leftarrow v_i}$ a la probabilité pr_i d'être exécuté. Notons que X ne peut être contraint par des processus autres que $Process$, i.e X est locale à $Process$.

Formellement, le comportement d'un processus de PCCP est décrit par un système de transitions probabilistes $(\Gamma, \mapsto_{pr})$. Dans la suite, on note par σ un magasin de contraintes. Γ est l'ensemble des états du système de transitions, appelés configurations. Une configuration est un couple $\langle Process, \sigma \rangle$, où $Process$ dénote le processus restant à exécuter et σ le magasin de contraintes courant. Le comportement de chaque opérateur de PCCP est décrit par les règles d'inférence données dans [5]. Grâce à ce système de transitions, il est possible de définir les états résultant de l'exécution d'un processus, appelés configurations terminales.
En considérant $Process$ un processus de PCCP, σ_0 le magasin initial de contraintes, l'ensemble des configurations terminales, noté $tc(Process, \sigma_0)$ est défini ainsi :

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

où $\bar{pr} \neq 0$ dénote la probabilité d'atteindre l'état σ et $\mapsto^*_{\bar{pr}}$ la fermeture transitive de la relation de transition.

Dans le cadre PCCP, les résultats considérés sont les configurations terminales consistantes, c'est à dire l'ensemble des magasins de contraintes après exécution du processus consistant.
L'ensemble des configurations terminales consistantes, noté *ctc*, est défini ainsi :

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

Exemple 1 (extrait de [5]).

$$P = choose(X, [0,1] - [1,1], tell(X = Z)) \parallel choose(Y, [0,1] - [1,1], if\ Z = 1\ then\ tell(Y = 1)).$$

Supposons que σ_0 soit égal à $true$, le résultat de l'exécution du processus P est : Z est contraint à 0 avec une probabilité $\frac{1}{2}$ (évènement $X = 0$), à 1 avec une probabilité $\frac{1}{4}$ (évènement $X = 1 \wedge Y = 1$) et le processus échoue avec une probabilité $\frac{1}{4}$ (évènement $X = 1 \wedge Y = 0$). Ainsi, $tc(P, true) = \{Z = 0, Z = 1, false\}$ et $ctc(P, true) = \{Z = 0, Z = 1\}$.

3 Le combinateur de contraintes probabiliste : *choose*

L'opérateur de choix probabiliste permet d'introduire de l'aléa dans l'exécution d'un processus de CCP. Cet aléa prend la forme d'une variable aléatoire interne à l'exécution d'un processus. Définir l'opérateur comme un combinateur de contraintes nous permet d'augmenter sa déclarativité. En effet, il est ainsi possible de raisonner avec un choix probabiliste partiellement connu, i.e une connaissance partielle de la loi suivie par la variable aléatoire X. Cette connaissance partielle est représentée par des contraintes portant sur la distribution de probabilité associée à la variable aléatoire X.

Dans la suite, nous supposons que le domaine de variation des variables est fini. Nous nous plaçons ainsi dans le cadre théorique de CC(*FD*) [7]. L'opérateur de choix probabiliste *choose* est décrit comme un nouveau combinateur de contraintes du système de contraintes.

3.1 Description opérationnelle

La distribution de probabilité de Loi_X est considérée comme une liste de variables à domaine fini, ainsi le combinateur $choose(X, [v_1, \ldots, v_n] - [P_1, \ldots, P_n], Process)$ est inséré dans le mécanisme de propagation de contraintes à domaines finis. L'objectif est d'exploiter la connaissance partielle du choix probabiliste lors de la phase de propagation. Le combinateur de contraintes *choose* est considéré comme étant résolu lorsque le choix probabiliste est effectué, i.e lorsque la variable aléatoire est instanciée. Si cette instanciation n'est pas possible, l'algorithme de filtrage dédié au combinateur de contraintes est exécuté. Dans le cas où le combinateur de contraintes n'est pas résolu, celui-ci est suspendu. Le réveil est déclenché lors d'une modification du domaine de variation d'une variable de la distribution de probabilité, i.e une connaissance plus précise de la loi suivie par X.

3.2 Connaissance partielle d'un choix probabiliste

La simulation (tirage) de valeurs pour la variable aléatoire X est toujours possible lorsque Loi_X est complètement connue. La connaissance partielle sur le choix probabiliste apparaît lorsque les variables de la distribution de probabilité associées à Loi_X ne sont pas instanciées. Dans ce cas, cette connaissance est représentée par un ensemble de lois pouvant être suivies par la variable aléatoire X. Cet ensemble de lois correspond à l'ensemble des instanciations possibles des poids représentant la distribution de probabilité. La connaissance partielle, portant sur la loi suivie par la variable aléatoire X, est donc définie comme l'ensemble des lois de probabilité pouvant être suivies par X.

Définition 1. *Supposons X une variable aléatoire de loi de probabilité $Loi_X = [v_1, , \ldots, v_n] - [P_1, \ldots, P_n]$. On définit l'ensemble des lois de probabilité pouvant être suivies par X, noté $\mathcal{SL}_X$, ainsi :*

$$\mathcal{SL}_X = \{[v_1,,\ldots,v_n] - [p_1,\ldots,p_n] \mid p_1 \in dom(P_1),\ldots,p_n \in dom(P_n)\}$$

où $dom(V)$ dénote le domaine de variation d'une variable V.

Exemple 2. Reprenons l'exemple du dé biaisé de biais inconnu donné en introduction :

$$tell(P_6 = 2 \cdot P_1) \;||\; choose(X, [1,2,3,4,5,6] - [P_1,P_2,P_3,P_4,P_5,P_6], tell(Dé = X))$$

Supposons que $dom(P_1) = 1..2$, $dom(P_2) = 2..2$, $dom(P_3) = 2..2$, $dom(P_4) = 2..2$, $dom(P_5) = 2..2$ et $dom(P_6) = 2..4$. L'incertitude dans le biais du dé est modélisée par les six lois pouvant être suivies par la variable aléatoire X :

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

Il est important de noter que seules deux lois de $\mathcal{SL}_X$ vérifient la contrainte $P_6 = 2 \cdot P_1$. $\mathcal{SL}_X$ est une sur approximation conservative des lois possibles pour X. Cette approximation est effectuée pour des raisons d'optimisation. Elle est obtenue en assurant une consistance de bornes sur le domaine des variables associées à la distribution de probabilité.

4 Filtrage associé à *choose*

L'objectif de l'algorithme de filtrage est de réduire l'ensemble des valeurs possibles pour la variable aléatoire X. L'algorithme repose sur le tirage a priori de la valeur de la variable aléatoire de loi uniforme sur $[0;1]$ simulant la valeur de X. En effet, le comportement d'une variable aléatoire de loi discrète[1] est simulé à partir de valeurs prises par une variable aléatoire de loi uniforme sur $[0;1]$, notée U par la suite. Le but principal de notre approche est d'exploiter au maximum la connaissance sur le choix probabiliste lors de la phase de propagation. L'algorithme de filtrage est appelé itérativement à chaque réveil du combinateur de contraintes.

[1] Par définition, une variable aléatoire suivant une loi discrète est à valeur dans un domaine fini ou dénombrable.

4.1 Simulation d'une variable aléatoire suivant une loi discrète

Les valeurs d'une variable aléatoire de loi discrète sont classiquement générées à partir de valeurs générées pour une variable aléatoire U de loi uniforme sur $[0;1]$. Une fonction de transfert établit le lien entre les valeurs prises par X et celles prises par U.

Supposons X une variable aléatoire de loi $[v_1,\ldots,v_n] - [p_1,\ldots,p_n]$. La correspondance entre les deux variables aléatoires est établie par la fonction f donnée par l'équation (1).

$$X = f(U) \text{ où } f : [0;1] \to \{v_1,\ldots,v_n\},\quad u \mapsto \begin{cases} v_1 & \text{si } u \in [0, pr_1[\\ \vdots & \\ v_n & \text{si } u \in [\sum_{j=1}^{n-1} pr_j, 1[\end{cases} \quad (1)$$

où u est une valeur prise par U.

Cette méthode permet de simuler la loi voulue. En effet, on a $\mathbb{P}(U \in [pr_1 + \ldots + pr_{i-1}, pr_1 + \ldots + pr_i])$, i.e la probabilité que $X = v_i$, égale à $(pr_1 + \ldots + pr_i) - (pr_1 + \ldots + pr_{i-1}) = pr_i$.

En reprenant l'exemple de la modélisation d'un jet de dé non biaisé,

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

La fonction de transfert établissant le lien entre U et X est représentée par la FIG. 1.

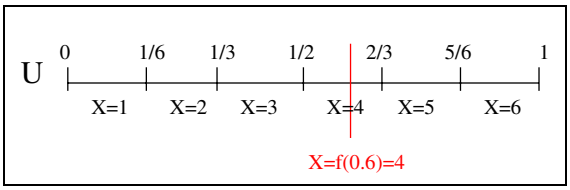

FIG. 1 – Simulation du comportement d'une variable aléatoire X de loi uniforme sur $\{1,2,3,4,5,6\}$

4.2 Propriété pour le filtrage des valeurs prises par X

L'objectif de l'algorithme de filtrage est d'exploiter l'information sûre sur le choix probabiliste : la valeur prise par U et $\mathcal{SL}_X$. La valeur de U étant fixée, il nous faut raisonner sur $\mathcal{SL}_X$ l'ensemble des lois pouvant être suivies par X. Basé sur l'équation 1, on associe à chaque élément de $\mathcal{SL}_X$ la fonction de transfert établissant le lien entre la valeur prise par X et la valeur

tirée pour U. Cet ensemble de fonctions de transfert est noté $\mathcal{F}_X$.

Intuitivement, si pour chaque élément f de $\mathcal{F}_X$, on a $f(u)$ différent de v_i appartenant au domaine de X, alors il est détecté que le choix probabiliste ne peut pas prendre la valeur v_i. Donc, on peut retirer du domaine de X la valeur v_i.

Le filtrage est basé sur la propriété suivante :

Propriété 1. *Soient X une variable aléatoire de loi $[v_1,\ldots,v_n]-[P_1,\ldots,P_n]$ et u une valeur générée pour U variable aléatoire de loi uniforme sur $[0;1]$ associée à la simulation de valeurs pour X. Alors, on a :*

$$u \notin \bigcup_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left[\sum_{j=1}^{i-1} pr_j, \sum_{j=1}^{i} pr_j\right] \Rightarrow X \neq v_i$$

Démonstration. Supposons

$$u \notin \bigcup_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left[\sum_{j=1}^{i-1} pr_j, \sum_{j=1}^{i} pr_j\right].$$

Alors,

$$\forall [v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X, u \notin \left[\sum_{j=1}^{i-1} pr_j, \sum_{j=1}^{i} pr_j\right].$$

Or à chaque élément de $\mathcal{SL}_X$ est associé un élément de $\mathcal{F}_X$. On a donc $\forall f \in \mathcal{F}_X, f(u) \neq v_i$. On peut conclure donc $X \neq v_i$. □

4.3 Algorithme de filtrage

L'algorithme de filtrage proposé pour l'opérateur de choix probabiliste est basé sur le résultat de la propriété 1. Ce résultat apporte une méthode pour réduire le domaine des valeurs pouvant être prises par la variable aléatoire X. Dans la suite, on note Int_{v_i} l'ensemble suivant $\bigcup_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left[\sum_{j=1}^{i-1} pr_j, \sum_{j=1}^{i} pr_j\right]$ associée à chaque v_i du domaine de X.

Algorithme 1 : Algorithme de filtrage pour l'opérateur *choose*

Entrée : X,Loi_X et U
Sortie : Domaine de X réduit

forall $v_i \in$ `Dom`(X) **do**
  `IntProba`(v_i, Loi_X, Int_{v_i});
  if $U \notin Int_{v_i}$ **then**
    $X \neq v_i$;
  end
end

Le point clé de l'algorithme réside dans le calcul effectué par IntProba. IntProba calcule Int_{v_i} pour chaque valeur du domaine de X. La propriété 1 permet ensuite de réduire le domaine de X en cas de non appartenance de U à Int_{v_i}. Notons cependant qu'une méthode approximant Int_{v_i} pour des raisons d'efficacité est présentée dans la sous-section 4.4.

Exemple. Reprenons l'exemple 2 avec $dom(P_1) = 1..2$, $dom(P_2) = 2..2$, $dom(P_3) = 2..2$, $dom(P_4) = 2..2$, $dom(P_5) = 2..2$ et $dom(P_6) = 2..4$.

$$tell(P_6 = 2 \cdot P_1) \;||\; choose(X, [1,2,3,4,5,6]-[P_1,P_2,P_3,P_4,P_5,P_6], tell(Dé = X))$$

On rappelle que

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

L'ensemble des éléments de $\mathcal{F}_X$ est donné par la Fig. 2.

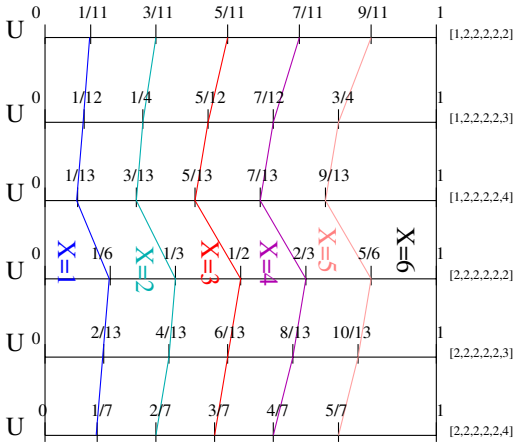

Fig. 2 – Simulation de valeur pour un dé biaisé

On a d'après la propriété 1 les résultats suivants :

$u \notin [0;\frac{1}{6}[\Rightarrow X \neq 1$
$u \notin [\frac{1}{13};\frac{1}{3}[\Rightarrow X \neq 2$
$u \notin [\frac{3}{13};\frac{1}{2}[\Rightarrow X \neq 3$
$u \notin [\frac{5}{13};\frac{2}{3}[\Rightarrow X \neq 4$
$u \notin [\frac{7}{13};\frac{5}{6}[\Rightarrow X \neq 5$
$u \notin [\frac{9}{13};1[\Rightarrow X \neq 6$

4.4 Méthode efficace pour le calcul de Int_{v_i}

L'efficacité de l'algorithme de filtrage est étroitement liée au calcul de Int_{v_i} effectué par IntProba.

Dans le cas général, construire l'ensemble $\mathcal{SL}_{\mathcal{X}}$ revient à énumérer tous les éléments du produit cartésien du domaine de variation des différentes variables de la distribution de probabilité associée à Loi_X. Pour des raisons d'efficacité, nous approximons le calcul de Int_{v_i} associée à chaque élément du domaine de X.

L'approximation de Int_{v_i} se base sur la recherche de l'élément de $\mathcal{SL}_X$ pour lequel $\sum_{j=1}^{i-1} pr_j$ (resp. $\sum_{j=1}^{i} pr_j$) est minorée (resp. majorée). Plus concrètement, l'approximation consiste à calculer analytiquement les bornes de Int_{v_i}. La méthode employée repose donc sur la propriété suivante :

$$\forall v_i \in dom(X)$$
$$Int_{v_i} \subseteq \left[\min_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i-1} pr_j \right), \max_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i} pr_j \right) \right].$$

Comme l'intervalle calculé est une sur approximation de Int_{v_i}, on a la propriété suivante :

$$\forall [v_1 \ldots, v_n] - [p_1, \ldots, p_n] \in \mathcal{SL}_X,$$
$$u \notin \left[\min_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i-1} pr_j \right), \max_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i} pr_j \right) \right]. \Rightarrow X \neq v_i.$$

où u est la valeur prise par U.

Le calcul de l'approximation de Int_{v_i} est efficace car $\min_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i-1} pr_j \right)$ et $\max_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i} pr_j \right)$ sont obtenus par un résultat analytique. D'après l'exemple présenté par la FIG. 2 et en fixant v_i à 3, l'échelle pour laquelle le minimum de Int_{v_i} est atteint est $[1, 2, 2, 2, 2, 4]$, i.e l'échelle pour laquelle les poids P_1 et P_2 ont leurs valeurs minimales et P_3, P_4, P_5 et P_6 leurs valeurs maximales.

En considérant $Loi_X = [v_1, v_2, \ldots, v_n] - [P_1, P_2, \ldots, P_n]$, l'échelle pour laquelle le minimum de Int_{v_i} est atteint est :

$$[v_1, \ldots, v_n]-$$
$$[\min(P_1), \ldots, \min(P_{i-1}), \max(P_i), \ldots, \max(P_n)]$$

On a donc

$$\min_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i-1} pr_j \right) = \frac{\sum_{j=1}^{i-1} \min(P_j)}{\sum_{j=1}^{i-1} \min(P_j) + \sum_{j=i}^{n} \max(P_j)}$$

Par un raisonnement similaire, on a

$$\max_{[v_1\ldots,v_n]-[p_1,\ldots,p_n]\in\mathcal{SL}_X} \left(\sum_{j=1}^{i-1} pr_j \right) = \frac{\sum_{j=1}^{i} \max(P_j)}{\sum_{j=1}^{i} \max(P_j) + \sum_{j=i+1}^{n} \min(P_j)}.$$

4.5 Etude de complexité de l'algorithme de filtrage

L'algorithme 1 est détaillé dans son principe général. Toutefois comme cela a été présenté dans la section 4.4, des optimisations sont mises en oeuvre pour assurer sa bonne efficacité. Entre autre le calcul des approximations des différents Int_{v_i} est effectué en calculant les valeurs de $\sum_{j=1}^{i} \min(P_j)$ et $\sum_{j=1}^{i} \max(P_j)$ pour $\forall i \in 1 \ldots n$. Ce calcul est obtenu en temps linéaire par rapport à la taille du domaine des valeurs pouvant être prises par X. Comme le reste de l'algorithme ne consiste qu'à tester si la valeur tirée pour U appartient aux différentes approximations de Int_{v_i} calculées, on peut donc conclure que l'algorithme de filtrage s'exécute en temps linéaire par rapport à la taille du domaine de variation de X.

5 Implantation de combinateur de contraintes probabilistes

Dans cette section, une librairie formée de trois combinateurs de contraintes probabilistes dérivés de *choose* est présentée. Cette implantation utilise la librairie clp(FD) de SICStus Prolog [2]. Dans la suite, un processus concurrent de PCCP est considéré comme un but Prolog, l'opérateur $if \ldots then \ldots$ comme l'opérateur d'implication *ask* et $\|$ comme une conjonction de clp(FD) [7].

Avant la description de trois nouveaux combinateurs, le prédicat `ptc(Goal,Var_List,Result)` est présenté. Etant donné un but Prolog, `ptc` calcule des configurations terminales et la probabilité d'atteindre chaque configuration terminale de manière empirique. Après un nombre donné d'exécutions de `Goal`, ce prédicat conserve le domaine de chacune des variables de `Var_List` (i.e la projection du magasin de contraintes

sur `Var_List`) et calcule le taux d'occurrences de chacun des magasins de contraintes obtenus.

Pour chaque combinateur de cette famille, un algorithme de filtrage dédié est implanté basé sur l'algorithme 1. Des combinateurs ont été créés afin de traiter différentes formes prises par Loi_X.

- `choose` où le domaine de X est une liste de valeurs et la distribution de probabilité une liste de variables à domaine fini ;
- `choose_range` où le domaine de X est un intervalle et la distribution de probabilité est une différence de listes de variables à domaine fini ;
- `choose_decision` où un choix aléatoire booléen est effectué entre l'exécution de deux buts.

5.1 Le combinateur `choose`

Le combinateur `choose` implémente l'opérateur de choix probabiliste de PCCP en tant que nouveau combinateur de contraintes. Le domaine de variation et la distribution de probabilité de la variable aléatoire `X` sont représentés par `[v1,..vn]` et un ensemble de variable à domaines finis `[P1,..Pn]`.

```
choose(X,[v1,..vn]-[P1,..,Pn],Goal,Options)
```

Notons que `Options` permet de prendre en compte différents paramètres de filtrage pouvant être demandées par l'utilisateur.

- `no_filtering` permet d'ignorer le filtrage lors des réveils de la contrainte. Cette option est utilisée lors des expérimentations pour obtenir un but référence ;
- `inconsistency_check` déclenche un test d'inconsistance entre les valeurs pouvant être prises par `X` et `Goal`. Cette option coûteuse est utilisée pour augmenter la déduction lors de la phase de propagation ;
- `lvar(L)` ajoute les variables de la liste L aux variables de réveil. Cette option est utilisée pour ajouter des variables de `Goal` à l'ensemble des variables de réveil ;
- `rv(U)` permet de récupérer la valeur tirée pour la variable aléatoire U générée uniformément sur $[0;1]$.

Exemple 3. La modélisation du dé biaisé décrit dans l'exemple 2 prend la forme suivante :

```
dice(De) :-
  P1 in 1..2,P2 in 2..2,P3 in 2..2,
  P4 in 2..2,P5 in 2..2,P6 in 2..4,
  2*P1#=P6,
  choose(X,[1,2,3,4,5,6]-
       [P1,P2,P3,P4,P5,P6],[X=De],[]).

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

Result=[(De=1,0.07735),(De in 1..2,0.09065),
        (De=2,0.06285),(De in 2..3,0.10175),
        (De=3,0.05135),(De in 3..4,0.11795),
        (De=4,0.03860),(De in 4..5,0.12775),
        (De=5,0.02575),(De in 5..6,0.1401),
        (De=6,0.1659)]
```

Le résultat obtenu correspond aux différents domaines possibles pour `X` après propagation. Par exemple, `(De=1,0.07735)` signifie `X` est valuée à 1 avec une probabilité `0.07735`. En reprenant Fig. 2 pour l'évènement `X=1`, deux cas sont possibles : U est à valeur dans $[0, \frac{1}{13}[$ ou U est à valeur dans $[\frac{1}{13}, \frac{1}{6}[$. Dans le premier cas, on peut conclure que `De=1`. Dans le deuxième, on peut réduire le domaine de `De` à `1..2`.

5.2 Le combinateur `choose_range`

Le combinateur `choose_range` implémente un opérateur de choix probabiliste pour lequel le domaine de variation est un intervalle. Cet intervalle est représenté par le terme `[Xmin,Xmax]` où `Xmin` (resp. `Xmax`) est sa borne inférieure (resp. supérieure). La distribution de probabilité prend la forme d'une différence de liste de variables à domaine fini, `[P1,..,Pn|Q]-Q`. Comme la longueur de la distribution est partiellement connue lorsque `Xmin` ou `Xmax` ne sont pas instantiées, ceci permet de la faire évoluer selon la connaissance sur les bornes de l'intervalles.

```
choose_range(X,[Xmin,Xmax]-[P1,..,Pn|Q]-Q,
                          Goal,Options)
```

`choose_range` est utilisé dans l'implantation d'un algorithme randomisé : une version faible du test de primalité de Miller-Rabin. L'objectif de cette modélisation est de contraindre une variable à ne prendre comme valeurs que des nombres premiers. Comme une méthode probabiliste est utilisée, la variable est contrainte à être un nombre premier avec une certaine probabilité. Cette modélisation est disponible en ligne [10].

5.3 Le combinateur `choose_decision`

L'opérateur `choose_decision` permet un choix probabiliste booléen entre deux processus. Ce choix probabiliste booléen est représenté par la liste `[W1,W2]` de deux variables à domaine fini. Le choix probabiliste est effectué entre l'exécution de `C,Goal1` et `Not_C,Goal2`.

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

```
quatre_deux_un([De1,De2,De3],[P1,P2,P3,P4,P5,P6],Options) :-
       2*P1#=P6,
       2*P2#=P5,
       2*P4#=P3,
       choose(X, [1,2,3,4,5,6]-[P1,P2,P3,P4,P5,P6], [De1=X], Options),
       choose(Y, [1,2,3,4,5,6]-[P1,P2,P3,P4,P5,P6], [De2=Y], Options),
       choose(W, [1,2,3,4,5,6]-[P1,P2,P3,P4,P5,P6], [De3=W], Options),
       all_different([De1,De2,De3]),
       De1+De2+De3#=7.
```

FIG. 3 – Modélisation du jeu 421 avec de dés biaisés de biais partiellement connu

Notons que `choose_decision(C,[W1,W2],Goal1, Goal2,Options)` peut être décrit par l'utilisation du combinateur `choose` : `choose(X,[0,1]-[W1,W2],[ask(X=0,Goal1), ask(X=1,Goal2)],Options)`. Cependant, notre approche permet d'implémenter un algorithme de filtrage dédié au combinateur de contraintes plus efficace.

L'utilisation de ce combinateur est liée à la modélisation d'un problème de génération automatique de cas de test probabilistes en un problème de résolution de contraintes. La technique du test statistique structurel [14] conduit à chercher une distribution de probabilité sur le domaine d'entrée d'un programme permettant de maximiser la couverture structurelle du programme sous test. L'approche consiste à modéliser chaque instruction d'un langage impératif simple (de type While) par un combinateur de contraintes [9]. `choose_decision` est introduit pour simuler le comportement d'une conditionnelle ou d'une boucle d'un programme impératif. Raisonner avec un choix probabiliste partiellement connu nous autorise à contraindre la distribution de probabilité selon diverses informations extraites de l'analyse du programme. Il est de plus nécessaire de pouvoir raisonner et filtrer avec cette connaissance partielle sur les différents choix probabilistes.

Exemple 4. Considérons le programme sous test suivant :

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

On restreint le domaine des variables x et y à être dans le domaine fini `0..1000`. La traduction de ce programme en un but de clp(FD) est la suivante :

```
foo(X,Y) :-
  X in 0..1000,
  Y in 0..1000,
  choose_decision(X#=<100#/\Y#=<100,[P1,P2],
    [choose_decision(Y#>X+50,[P3,P4],
                             [],[],[]),
    choose_decision(X*Y#<100,[P5,P6],
                             [],[],[])],
  [],[]),
  Coeff1*P1#=Coeff2*P2,
  Coeff3*P3#=Coeff4*P4,
  Coeff5*P5#=Coeff6*P6.
```

De l'information statique (connaissance de l'objectif du test) ou dynamique (la détection d'un chemin non exécutable) est utilisée pour contraindre les variables `Coeff1,..,Coeff6`. L'instanciation des variables `X` et `Y` correspond à la génération d'un cas de test. Notons que ceci ne correspond qu'à une première modélisation du problème de génération de cas de test probabiliste en un problème à contraintes et d'autres modélisations sont en cours d'étude.

Une version des opérateurs de choix probabilistes décrite dans cette section est disponible en ligne [10].

5.4 Première validation expérimentale avec le jeu de dé 421

Une première validation expérimentale de l'algorithme de filtrage est présentée dans cette section. Elle est basée sur un exemple simple : la modélisation du jeu de 421 avec des dés biaisés de biais inconnus mais identiques. En simplifiant un peu les règles, le jeu 421 consiste en un jet de trois dés qui doivent faire apparaître les faces 4, 2 et 1. Le prédicat `quatre_deux_un([De1,De2,De3],Distrib,Options)` est vrai ssi les trois choix probabilistes ont menés à l'instanciation des variables `De1,De2,De3` aux valeurs $1, 2$ et 4 indépendamment de l'ordre. L'hypothèse de jeu est la suivante : la personne nous proposant le jeu est malhonnête. Le biais du dé est donc modélisé

	Pas réduction de domaine			Détection d'inconsistance		
	$Pmax = 10$	$Pmax = 20$	$Pmax = 30$	$Pmax = 10$	$Pmax = 20$	$Pmax = 30$
Sans filtrage	1	1	1	0	0	0
Avec filtrage	0.0058	0.00906	0.1934	0.54834	0.3078	0.2006

FIG. 4 – Résultat de $Exp1$ pour $Pmax = 10$, $Pmax = 20$ et $Pmax = 30$

ainsi : la chance de tirer un 1 est deux fois plus faible que celle de tirer un 6, la chance de tirer un 4 est deux fois plus faible que celle de tirer un 3 et la chance la chance de tirer un 2 est deux fois plus faible que celle de tirer un 5. L'implantation est donnée par la FIG.3.

La première expérience, notée $Exp1$ permet de mettre en valeur la déduction apportée par l'algorithme de filtrage. Pour cela, on compare les domaines après propagation des variables `De1,De2,De3` après deux exécutions du prédicat `quatre_deux_un/3`. Dans l'une, l'option `no_filtering` est utilisée, dans l'autre l'algorithme filtrage est employé. Le prédicat `ptc/3` est utilisé pour enregistrer les différentes configurations après la propagation. La validation expérimentale se base sur l'exécution de ces deux requêtes :

```
?- ptc([quatre_deux_un([De1,De2,De3],Distrib
ution)],[De1,De2,De3],Result,[no_filtering]).
```

et

```
?- ptc([quatre_deux_un([De1,De2,De3],Distri
bution)],[De1,De2,De3],Result,Options).
```

Notons que sans filtrage, le mécanisme de propagation du résolveur de contraintes réduit le domaine de variation de `De1,De2,De3` à `1..5`. Les résultats obtenus sont donnés par la FIG.4. Le domaine de chaque `Pi` est `1..Pmax`. Les résultats correspondent à la probabilité de l'évènement décrit pour chaque colonne du tableau.

L'exécution de la requête dans laquelle l'algorithme de filtrage n'est pas employé ne réduit pas le domaine de variation de `De1`, `De2` et `De3`. Le choix probabiliste est en fait délayé en attente d'une instanciation complète des variables de `Distribution`. Entre autres, il est mis en évidence le pouvoir de déduction d'inconsistance de `quatre_deux_un/4`. Notons cependant que l'information obtenue sur le choix probabiliste est étroitement lié à la connaissance partielle sur les variables du choix probabiliste. Plus les intervalles associés à chaque face du dé calculé par *IntProba* sont disjoints, plus la connaissance sur les valeurs pouvant être prises par le choix probabiliste est précise. Dans le pire cas, i.e lorsque tous Int_{v_i} avec $i \in 1 \ldots n$ sont égaux à [0.1], l'algorithme de filtrage ne permet pas de réduction du domaine de X.

La deuxième expérimentation met en valeur le gain au niveau du temps d'exécution apporté par l'algorithme de filtrage. Notre expérimentation se base sur la recherche de l'ensemble des instanciations de `Distribution` menant au gain du jeu. Pour cela, une procédure d'énumération sur les variables de `Distribution` est associée à l'exécution de `quatre_deux_un/4`.

```
?- quatre_deux_un([De1,De2,De3],
                  Distribution,Options),
   labelling_findall([],Distribution).
```

Deux requêtes sont de nouveau comparées, l'une utilisant l'option `no_filtering`, l'autre utilisant l'algorithme de filtrage. La requête est réitérée 50000 fois en utilisant SICStus 3.11 sous Windows XP avec un ordinateur possédant un processeur Intel Pentium M 2.00 Ghz et 1 Go de mémoire vive. Le temps d'exécution moyen d'une requête nous est donné par FIG. 5 pour $Pmax$ variant de 10 à 50. $Pmax$ a été restreint à cet intervalle de variation par souci d'avoir une réponse à la requête en un temps raisonnable.

Les premiers résultats donné par $Exp2$ nous laisse supposer que l'algorithme permet d'obtenir en général un gain de temps non négligeable. Ce gain de temps est étroitement lié aux réductions du domaine des choix probabilistes obtenu lors du filtrage. Dans le pire des cas, i.e lorsque l'algorithme n'apporte aucune réduction de domaine, l'algorithme de filtrage apporte un surplus au temps d'exécution. Ce surplus reste faible car l'algorithme de filtrage s'exécute en temps linéaire par rapport à la taille du domaine du choix probabiliste.

6 Conclusion

Cet article introduit une extension des fonctionnalités de l'opérateur de choix probabiliste *choose*, en le considérant comme un nouveau combinateur de contraintes. Un algorithme de filtrage efficace associé à chaque combinateur de contraintes permet de raisonner et filtrer sur un choix probabiliste partiellement connu. Cet algorithme a été mis en oeuvre dans l'implantation d'une librairie de nouveaux opérateurs de choix probabilistes pour la librairie clp(FD) de SICStus Prolog. Les premières validations expérimentales sont encourageantes. Cependant, il est nécessaire

	Temps moyen d'exécution d'une requête en ms				
	$Pmax = 10$	$Pmax = 20$	$Pmax = 30$	$Pmax = 40$	$Pmax = 50$
Sans filtrage	5.3	38.9	129.3	309.2	603
Avec filtrage	6	12.5	33.5	66.1	123.7

FIG. 5 – Résultat de $Exp2$ pour $Pmax = 10$, $Pmax = 20$, $Pmax = 30$, $Pmax = 40$ et $Pmax = 50$

d'étendre notre validation à des problèmes plus importants. L'application principale de notre travail est l'utilisation de nos opérateurs de choix probabilistes pour la résolution d'un problème de test logiciel : la génération automatique de cas test probabilistes pour la couverture structurelle de programmes sous test. Nos efforts futurs se porteront donc sur la modélisation de ce problème en un problème à contraintes. Parallèlement, nous avons la volonté d'étoffer la librairie d'opérateurs de choix probabilistes afin d'étendre le domaine d'application de notre travail.

Références

[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, 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] 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.

[9] 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.

[10] M Petit and A. Gotlieb. Library of probabilistic constraint combinators (beta version), available at http ://www.irisa.fr/lande/petit/tools.html. 2005.

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

[12] 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.

[13] 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.

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