Un Langage Distribué Concurrent avec Contraintes

Jean-Hugues Réty

INRIA - Rocquencourt,
B.P. 105,
F–78153 Le Chesnay cedex
E-mail : rety@margaux.inria.fr

***Résumé* :** *Nous étudions le problème de la modélisation de systèmes distribués dans les langages concurrents avec contraintes (CC). Les langages CC forment un modèle de calcul par contraintes à mémoire partagée et certaines de leurs caractéristiques, en particulier la globalité du* store, *font obstacle à leur implantation distribuée. Nous pensons qu'il est nécessaire d'y introduire des traits nouveaux afin de construire un langage adapté à la programmation distribuée. Nous proposons l'introduction d'un mécanisme de communication asynchrone par messages. Celui-ci s'intègre d'une manière très homogène dans les langages CC et donne lieu à des notions de calcul local, de store local et de site tout en évitant le recours à la résolution distribuée de contraintes, les contraintes permettant une modélisation logique de la communication par messages. Nous donnons des exemples de programmes issus des domaines de la concurrence et de la distribution montrant le style de programmation obtenu.*

***Mots-clés* :** *Langages distribués, Langages concurrents, Langages Concurrents avec Contraintes.*

1 Introduction

Les langages concurrents avec contraintes (CC) forment un modèle de calcul concurrent par contraintes qui suscite un intérêt certain. Certains de leurs traits essentiels semblent cependant faire obstacle à leur implantation distributée. En particulier, la communication et la synchronisation se font par l'intermédiaire d'un *store* global. Certains modèles de calcul distribué sont basés sur une mémoire partagée (par ex. Linda ([CAR 89])). L'effectivité de cette approche repose cependant sur la possibilité d'implanter de manière distribuée cette mémoire partagée, or cette condition semble difficile à satisfaire dans le cas des langages CC : d'une part l'opérateur *ask* nécessite de tester la relation de conséquence sur les contraintes de façon globale ; en effet, le fait qu'une contrainte soit conséquence du store dépend à priori de la totalité de l'information contenue dans le store. D'autre part, l'opérateur *tell* considéré par Saraswat dans [SAR 89] est ce qu'il appelle le *tell atomique* : une contrainte n'est publiée que si elle est cohérente avec le store, dans le cas contraire l'agent *tell* est *aborté*. Dans un système distribué, cela conduit soit à la résolution distribuée de contraintes soit à la diffusion[1] de toute l'information publiée. La résolution distribuée de contraintes apparaît comme étant un problème difficile : aucun algorithme efficace et assez général n'existe actuellement. Bien que cela soit très coûteux, on pourrait envisager de satisfaire aux problèmes liés au *ask* en maintenant une copie du *store* sur chaque site. Cependant, même en employant cette solution extrême, le *tell atomique* poserait toujours problème. Partant de ce constat, Saraswat propose ([SAR 89], Chap. 7) l'*eventual tell* qui est le *tell* sans test de cohérence. Ainsi, dans un système distribué, il n'est plus requis que les contraintes soient immédiatement communiquées à tous les sites : il est suffisant que cela soit effectué en un temps fini. Malheureusement, l'*eventual tell* à pour conséquence de rendre difficile la construction de programmes sans échecs. Comment, en effet, assurer alors la cohérence du store ? Or cette propriété apparaît souhaitable dans un cadre distribué afin d'assurer la résistance aux pannes. Un système ne doit pas s'arrêter dès qu'un problème survient sur un site. Le langage Janus ([SAR 90]) à été créé dans le but de répondre à ce problème. Cependant, les conditions qu'il impose sur le langage et le système de contraintes (en particulier, une variable ne doit avoir qu'un producteur et qu'un consommateur) sont assez restrictives.

Un système distribué se doit de fournir la possibilité d'effectuer du calcul local, ainsi qu'un mécanisme de transmission du résultat. Les langages CC offrent cela par restriction de la portée des variables (opérateur de restriction), ce qui donne lieu à une certaine notion de variable locale. Mais celle-ci n'est vraiment effective qu'au niveau syntaxique : le résultat d'un calcul local ne peut-être transmis que par l'intermédiaire du partage des variables, ce qui signifie que ce résultat est inséparable de l'information concernant les variables locales, des contraintes portant sur celles-ci pouvant contraindre des variables globales. Il faudrait pouvoir séparer l'information purement locale de l'information significative au niveau global, définir ce que Saraswat appelle des "fire walls" dans la discussion à ce sujet, chapitre 7 de sa thèse. Cependant, décider de ce qu'il est nécessaire de diffuser et de ce qui ne l'est pas risque d'être très difficile en général. Une approche serait d'utiliser des propriétés spécifiques à certains systèmes de contraintes – par exemple des conditions syntaxiques permettant d'avoir des propriétés d'indépendance de variables ; on retrouve d'ailleurs cette idée dans Janus.

Nous pensons que la question de la distribution dans les langages CC doit être considérée d'un point de vue différent. Nous proposons de remettre en cause certains traits fondamentaux concernant la globalité du store, la communication et la synchronisation par l'introduction d'une notion de communication par messages.

La section 2 définit les systèmes de contraintes. La section 3 présente un modèle

1. broadcast

de calcul appelé *MCC* qui intègre communication par messages et langages CC. La section 4 définit sa sémantique opérationnelle. La section 5 présente un mécanisme nouveau de propagation du résultat d'un calcul local. Des exemples de programmation en MCC sont donnés dans la section 6. La section 7 discute brièvement d'une notion de *distribuabilité* qui capture une intuition simple dont le but est de faciliter une implantation distribuée.

2 Systèmes de contraintes

Différentes formalisations des systèmes de contraintes ont été proposées dans la littérature ([JAF 87, SAR 92, FdB 97]). Nous utilisons ici une formalisation simple dans laquelle un système de contraintes est un langage du premier ordre muni d'une interprétation fixée.

Définition 1 *Un système de contraintes $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ est formé d'un langage du premier ordre $\mathcal{L}$ clos par conjonction et quantification existentielle et qui contient le prédicat "=", et d'une interprétation $\mathcal{I}$ dans laquelle "=" est interprété comme étant l'égalité. Nous supposerons que le langage contient au moins deux constantes distinctes (elles serviront à définir les contraintes $vrai$ et $faux$).*

Etant donné un système de contraintes $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$, les contraintes sont les formules du langage. c étant une contrainte, nous notons $vl(c)$ ses variables libres. $\vec{x}$ et $\vec{y}$ étant des vecteurs de variables de même longueur, $c[\vec{y}/\vec{x}]$ dénote le renommage dans c des variables de $\vec{x}$ par celles de $\vec{y}$.
Nous définissons un pré-ordre partiel sur l'ensemble des contraintes de $\mathcal{C}$, appelé relation de conséquence et noté $\vdash_{\mathcal{C}}$. (Bien que notre formalisation, que nous pourrions appeler *formalisation logique*, des systèmes de contraintes et de la relation de conséquence devrait nous conduire à utiliser le symbole $\models$, nous conservons $\vdash$ qui est la notation traditionnelle dans les cadre des langages CC.)

Définition 2 *(Relation de conséquence)*
Soit $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ un système de contraintes. Soient c, d deux contraintes de $\mathcal{C}$.
$c \vdash_{\mathcal{C}} d$ si pour toute valuation qui satisfait c, la contrainte d est elle aussi satisfaite (dans $\mathcal{I}$).

Etant donné un système de contraintes $\mathcal{C}$, une relation d'équivalence $\sim_{\mathcal{C}}$ est induite par $\vdash_{\mathcal{C}}$ sur l'ensemble des contraintes. Par la suite, cela ne générant pas d'ambiguïté (nous travaillerons avec un unique système de contraintes), nous noterons $\vdash$ la relation de conséquence et $\sim$ sa relation d'équivalence induite.
Nous aurons besoin de la notion de contrainte incohérente.

Définition 3 *Soit* $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ *un système de contraintes. Une contrainte* c *de* $\mathcal{C}$ *est dite* incohérente *s'il n'existe aucune valuation qui la satisfait (dans* $\mathcal{I}$*).*
Soient $const$ *et* $const'$ *deux constantes distinctes de* $\mathcal{L}$*, nous noterons* $faux$ *la contrainte incohérente* $const = const'$ *et* $vrai$ *la tautologie* $const = const$*.*

3 Le modèle de calcul MCC

Nous proposons un modèle de calcul concurrent par contraintes qui intègre langages CC (communication par agents ask et $tell$) et communication asynchrone par messages. Nous l'appelons MCC (Message-based Concurrent Constraint programming). Cette section définit sa syntaxe et sa sémantique opérationnelle.

3.1 *Le paradigme de communication*

Nous introduisons dans les langages CC deux nouvelles primitives de communication que nous notons $send$ et $receive$. La communication s'effectue par messages et intervient par l'intermédiaire de canaux de communication. Ces canaux seront en fait modélisés par des variables du système de contraintes. Celui-ci nous permettra de formaliser d'une manière simple et élégante les mécanismes liés à la communication par messages, montrant ainsi – s'il en est encore besoin – le grand intérêt que présente la notion de contrainte en théorie des langages de programmation.

Le modèle MCC, bien qu'orienté vers la programmation distribuée, n'explicite pas directement la répartition des agents. Un programme MCC sera dit *distribuable* s'il vérifie certaines propriétés qui reposent sur des intuitions simples. Intuitivement, un programme MCC est distribuable s'il est "décomposable" un réseau de sites ne communiquant par messages, chaque site étant un store local sur lequel est exécuté un ("morceau" de) programme MCC. Considérons, dans un but explicatif (rappelons qu'au niveau formel variables et canaux sont des objets de même nature et ne se distinguent que par leur emploi), que les variables du système de contraintes sont partionnées en un ensemble de *variables de contraintes* et un ensemble de *variables de canaux*. Le cas simple de programme distribuable est celui d'un programme qui est la composition parallèle de plusieurs agents (*sites*) qui ne partagent aucune *variable de contraintes*. En effet, ce qui rend difficile la distribution dans les langages CC est le fait que l'information soit échangée par l'intermédiaire des liens entre variables définis par les contraintes.
L'absence de partage de *variables de contraintes* ne suffit cependant pas à assurer l'indépendance opérationelle des sites (au niveau de la résolution de contraintes : il ne faut pas qu'une incohérence sur un site libère un agent $ask(c).A$ sur un autre site). Nous devons donc imposer en prémisse de la règle de réduction des agents ask que l'opérateur ask ne soit pas libéré par la contrainte $faux$.

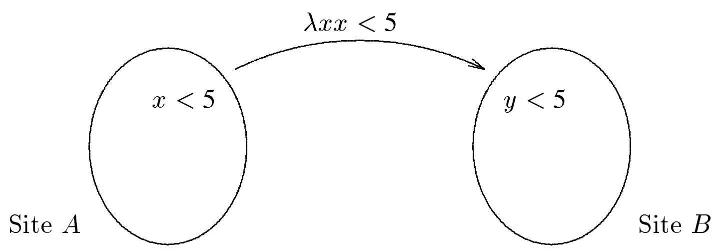

Figure 1 : *Communication par abstractions de contraintes*

Les agents $send$ et $receive$ prennent en argument le canal sur lequel la communication doit avoir lieu et l'objet informatique qui doit être communiqué. Dans le cadre de programmation par contraintes qui est le nôtre, il paraît évident que cet objet doit être lui-même une contrainte. Cela pose cependant un problème : celui de la signification de la contrainte reçue. En effet, supposons qu'un site A communique à un site B la contrainte $x < 5$. La variable x étant locale au site A, ce que l'on veut communiquer alors est qu'une certaine variable y, locale au site B, est inférieure à cinq ; et ajouter la contrainte $y < 5$ au store du site B. Les objets communiqués ne doivent donc pas être des contraintes mais des “abstractions de contraintes”, c'est à dire des contraintes abstraites de (certaines de) leurs variables ; le site récepteur renommant celles-ci par des variables locales (figure 1). Ce mécanisme à aussi l'intérêt de permettre la reconfiguration dynamique du réseau de communication à la manière du π-calcul (cf. l'exemple 6.1).

Formellement, nous définissons les abstractions de contraintes comme étant des couples formés d'un vecteur linéaire de variables et d'une contrainte. Nous avons besoin de ce vecteur de variables pour, lors de la réception d'un message, remplacer les variables abstraites par les variables locales spécifiées par l'agent de réception. Ce dont nous avons besoin en fait est d'un mécanisme qui établisse une correspondance entre les variables de la contrainte envoyée et celles du site de réception. Nous aurions pu utiliser à cet effet une formalisation différente qui consisterait en la donnée d'un ordre total sur l'ensemble des variables. Cela permettrais de transmettre des contraintes (et non pas des abstractions de contraintes), le renommage à la réception s'effectuant alors selon cet ordre et l'ordre dans lequel les variables de réception sont spécifiées. La notion d'abstraction de contrainte que nous utilisons définit pour sa part un ordre sur les variables qui est local à chaque message. Une autre idée de formalisation de ce mécanisme serait d'utiliser une notation à la De Bruijn, ce qui serait une autre façon de spécifier un ordre local à chaque message.

Considérons un système de contraintes $\mathcal{C}$.

Définition 4 *L'ensemble des* abstractions de contraintes *de* $\mathcal{C}$ *est :*
$\mathcal{C}_{abs} = \{(\vec{x}, c) | c \in C,\ \vec{x} \subseteq vl(c)\ et\ \vec{x}\ linéaire\}$

Nous noterons par la suite $\lambda\vec{x}c$ l'abstraction de contrainte $(\vec{x}, c)$.
L'application d'un vecteur de variables à une abstraction de contrainte est la contrainte obtenue par renommage des variables abstraites :

Définition 5 *Soit* $\lambda\vec{x}c$ *une abstraction de contrainte. Soit* $\vec{y}$ *un vecteur de variables de même longueur que* $\vec{x}$*.*
*L'*application de $\vec{y}$ à $\lambda\vec{x}c$ *est la contrainte* $c[\vec{y}/\vec{x}]$.

3.2 *Syntaxe*

Considérons un système de contraintes $\mathcal{C}$.
Les programmes MCC sur $\mathcal{C}$ sont définis par la syntaxe suivante :

$$\begin{array}{lll} Programme & ::= & Declaration.Agent \\ Declaration & ::= & \epsilon \mid p(\vec{x}) :: Agent \mid Declaration.Declaration \\ Agent & ::= & c \mid ask(c).Agent \\ & & \mid send(x, \lambda\vec{y}c) \mid receive(x, \vec{y}).Agent \\ & & \mid Agent \parallel Agent \mid Agent + Agent \mid p(\vec{x}) \mid \exists_x Agent \end{array}$$

c est une contrainte, x est une variable, $\vec{x}$ est un vecteur de variables, $\vec{y}$ est un vecteur linéaire de variables, $\lambda\vec{y}c$ est une abstraction de contrainte, p est un symbole de procédure appartenant à un ensemble dénombrable de symboles de procédures auquel est associé une fonction d'arité.

Un programme est constitué d'un ensemble de déclarations (de procédures) et d'un agent appelé *but* ou *agent initial*. Etant donné un programme $Prog = Decl.But$, pour chaque symbole de procédure p d'arité n apparaissant dans un agent de $Prog$, il existe dans $Decl$ une unique déclaration $p(x_1, ..., x_n) :: A$. Etant donné un système de contraintes $\mathcal{C}$ et un ensemble de symboles de procédures Π, nous notons $Agents$ l'ensemble des agents défini par cette syntaxe.

Les agents *send* et *receive* sont appelés *agents de communication*. Le premier argument d'un agent de communication A est appelé *canal de* A. Les autres agents et constructeurs d'agents correspondent à ceux des langages CC.

4 Sémantique opérationnelle

Nous décrivons la sémantique opérationnelle MCC dans un style inspiré de la *Machine Abstraite Chimique (CHAM)* ([BER 92]). Nous ne conservons en fait de la

CHAM que l'idée de travailler modulo une congruence structurelle sur les agents qui prend en compte les aspects syntaxiques que sont par exemple la commutativité des opérateurs ou l'α-renommage. Cela revient à définir un système de transitions par l'intermédiaire de règles d'inférence de type SOS sur l'ensemble quotient des agents par la congruence structurelle.

4.1 *Congruence structurelle*

La congruence structurelle $\equiv$ sur les agents est la plus petite relation de congruence vérifiant les axiomes suivants :

- $\langle Agents, \|, vrai\rangle$ et $\langle Agents, +, vrai\rangle$ sont des monoïdes commutatifs.
- $\exists_x A \equiv \exists_y A[y/x]$ si y n'est pas libre dans A
  $A[y/x]$ est le renommage de x par y dans A.
  $\exists_x A \parallel B \equiv \exists_x(A \parallel B)$ si x n'est pas libre dans B
  $\exists_x A + B \equiv \exists_x(A + B)$ si x n'est pas libre dans B
  $\exists_x \exists_y A \equiv \exists_y \exists_x A$
  $\exists_x A \equiv A$ si x n'est pas libre dans A
- $c \parallel d \equiv d \wedge c$

4.2 *Règles d'inférence*

La figure 2 présente les règles d'inférence MCC. La régle (communication) réduit la composition parallèle de deux agents $send$ et $receive$ en présence d'une contrainte qui établit l'égalité de leurs canaux de communication. C'est-à-dire lorsque l'égalité de leurs canaux de communication est conséquence du store. La règle (ask) est légèrement modifiée par rapport à celle des CC : il est ici demandé que la contrainte qui libère le ask soit cohérente : dans un cadre distribué, l'incohérence d'un store local sur un site ne doit pas débloquer des agents ask sur les autres sites. Cette version de l'opérateur ask correspond au ask $atomique$ proposé dans [FdB 93]. Notons que nous utilisons ici l'opérateur de choix à un pas (règle (choix)). L'introduction d'un opérateur de choix gardé serait d'un intérêt évident et fera l'objet de travaux futurs.

Etant donné $Prog = Decl.But$ un programme MCC, ces règles définissent un système de transitions $T = \langle Agents, \rightarrow\rangle$, où $\rightarrow$ est la plus petite relation sur les agents vérifiant les règles d'inférence définies ci-dessus. Nous l'appelons système de transitions associé à $Prog$.

$$\frac{A \rightarrow A'}{A \parallel B \rightarrow A' \parallel B} \quad \text{(parallel)} \qquad \frac{A \rightarrow A'}{\exists_x A \rightarrow \exists_x A'} \quad \text{(exists)}$$

$$\frac{A \rightarrow A'}{A + B \rightarrow A'} \quad \text{(choix)} \qquad \frac{A \equiv B \quad A \rightarrow A' \quad A' \equiv B'}{B \rightarrow B'} \quad \text{(équiv)}$$

$$\frac{d \vdash c \quad d \not\vdash faux}{d \parallel ask(c).A \rightarrow d \parallel A} \quad \text{(ask)}$$

$$\frac{p(\vec{x}) :: A \in Decl \quad \vec{x} \cap \vec{y} = \emptyset}{p(\vec{y}) \rightarrow \exists_{\vec{x}}(\vec{x} = \vec{y} \parallel A)} \quad \text{(récursion)}$$

$$\frac{d \vdash u = v \quad longueur(\vec{x}) = longueur(\vec{y})}{d \parallel send(u, \lambda\vec{x}c) \parallel receive(v, \vec{y}).A \rightarrow d \parallel c[\vec{y}/\vec{x}] \parallel A} \quad \text{(communication)}$$

Figure 2 : *Règles d'inférence MCC*

5 Opérateurs sur les stores

Le langage que nous avons défini impose à l'utilisateur la complète spécification des contraintes à communiquer. En pratique, un système distribué aura besoin de communiquer des "parties" de stores locaux plutôt que des contraintes. L'exemple typique en est la communication du résultat d'un calcul local exécuté sur un *site* particulier. Dans un tel cas, ce qui est envoyé est toujours une abstraction de contrainte mais celle-ci ne peut-être directement exprimée dans le langage car elle dépend du store calculé. L'outil qui nous proposons ci-dessous répond à ce besoin. Nous donnons en particulier la possibilité d'utiliser dans un agent $send$ non plus seulement des contraintes mais aussi des *opérateurs de store*.

Définition 6 *Un* opérateur de store *est une fonction $\theta : \vec{v} \times c \mapsto d$, c, d étant des contraintes et $\vec{v}$ un vecteur de variables, vérifiant : si $\theta(\vec{v}, c) = d$ alors $vl(d) \subseteq \vec{v}$.*

Nous ne discutons pas ici quels opérateurs seraient utiles. Le plus simple d'entre eux – et son intérêt pratique est évident – est l'opérateur de projection.

Définition 7 *Soit c une contrainte. La projection de c sur un vecteur de variables $\vec{x}$ est :*

$$proj(\vec{x}, c) = \exists_{vl(c)/\vec{x}} c$$

Nous introduisons un nouveau type d'agent appelé *lien* et noté $lien((\theta, \vec{x}), \vec{y})$. Un tel agent établit un "lien" entre deux ensembles de variables : l'information produite sur les variables $\vec{x}$ est transmise aux variables $\vec{y}$ par l'intermédiaire de l'opérateur θ. Nous parlons ici de "lien" car une fois introduit, un agent $lien$ ne peut être retiré et crèe un sorte de "passerelle" entre variables de différents "sites". La sémantique des agents $lien$ est décrite par une règle qui propage l'information disponible sur les variables $\vec{x}$ aux variables $\vec{y}$. Un lien peut être introduit soit directement dans un programme, soit par l'intermédiaire de la communication par messages. Nous étendons à cet effet la communication par messages aux opérateurs de store.

La syntaxe des langages MCC est étendue avec les agents suivants :
$send(u, (\theta, \vec{x}))$ pour θ opérateur de store et $\vec{x}$ vecteur de variables,
$lien((\theta, \vec{x}), \vec{y})$ pour θ opérateur de store et $\vec{x}, \vec{y}$ vecteurs de variables de même longueur.
La sémantique de ces nouveaux agents est donnée par les règles de réduction suivantes :

$$\frac{}{\begin{array}{c} d \parallel lien((\theta, \vec{x}), \vec{y}) \\ \longrightarrow \\ d \parallel lien((\theta, \vec{x}), \vec{y}) \parallel \theta(\vec{x}, d)[\vec{y}/\vec{x}] \end{array}} \quad \text{(propagation}_{lien}\text{)}$$

$$\frac{d \vdash u = v \quad d \not\vdash faux \quad longueur(\vec{x}) = longueur(\vec{y})}{\begin{array}{c} d \parallel send(u, (\theta, \vec{x})) \parallel receive(v, \vec{y}).B \\ \longrightarrow \\ d \parallel lien((\theta, \vec{x}), \vec{y}) \parallel B \end{array}} \quad \text{(communication}_{lien}\text{)}$$

La règle (communication$_{lien}$) décrit l'introduction d'un agent $lien$ par communication par messages. La règle (propagation$_{lien}$) représente un mécanisme opérationnel simple : la propagation du résultat d'un calcul local. Cette règle a le défaut d'introduire des dérivations infinies. Nous avons même : si la règle (propagation$_{lien}$) s'applique à un agent A alors il existe une dérivation infinie pour A (on peut appliquer cette même règle indéfiniment). Cela ne pose pas de problème d'un point de vue pratique (le mécanisme opérationnel ainsi représenté est évident). Cela pose cependant un problème sémantique. Une solution pourrait être apportée par la définition d'une sémantique opérationnelle dont les états soient des couples formés d'un agent et d'un store comme dans, par exemple, [SAR 91]. Cela compliquerait cependant la description de la communication par messages qui nécessite la juxtaposition syntaxique des agents $send$ et $receive$.

6 Exemples

Les exemples que nous décrivons dans cette section mettent en évidence le pouvoir d'expression des langages MCC ainsi que le style de programmation, combinant communication $ask/tell$ et communication par messages, de ces langages. Nous modélisons, entre autres, des problèmes classiques de la programmation distribuée. La notion de consommation de ressource introduite par la communication par messages se révèle d'une grande utilité pour la conception de programmes réactifs ou interactifs. Il sagit là d'une caractéristique qui fait grandement défaut dans les langages CC (cf. [BRI 96] pour une extension intégrant une notion de consommation de ressource). En particulier, elle permet la simulation de variables impératives d'une manière bien plus élégante que celle employée habituellement (liste historique des valeurs prises par la variable) dans les LLC, les langages CC et la Programmation Logique en général.

Nous utiliserons le racourcis suivant : si a est une variable ou une constante du système de contraintes alors $send(u, a)$ dénote $\exists_x send(u, \lambda xx = a)$.

6.1 *Reconfiguration*

La reconfiguration dynamique du réseau de communication revêt une grande importance pour la concurrence et la distribution. Cela s'effectue dans les langages CC par l'intermédiaire de variables impératives modélisées par des listes (cf. [LAN 92]). Dans les MCC, elle est obtenue d'une façon beaucoup plus naturelle par communication de canaux à la manière du π-calcul ([MIL 89]).
Dans l'exemple suivant, deux agents p et q partagent un canal a et p partage un canal b avec un troisième agent r. Considérons le programme dont les déclarations de procédures sont (les déclarations de s et t ne sont pas spécifiées) :

$$\begin{array}{lll} p(a,b) & :: & send(b,a) \parallel \exists_{\vec{y}} receive(a,\vec{y}).s(\vec{y}) \\ q(a) & :: & send(a,\lambda\vec{x}c) \\ r(b) & :: & \exists_d(receive(b,d).\exists_{\vec{y}} receive(d,\vec{y}).t(\vec{y})) \end{array}$$

et le but : $\exists_{a,b}(p(a,b) \parallel q(a) \parallel r(b))$

Après communication du canal a sur le canal b, a est partagé par les trois sites. $send(a, \lambda\vec{x}c)$ peut alors communiquer avec l'un ou l'autre des deux autres sites. Il y a communication d'un canal, plus précisément cela correspond à la "scope extrusion" du π-calcul.

6.2 *Client-Serveur*

De nombreux problèmes de programmation nécessitent une certaine non monotonie de l'information manipulée. Ceux-ci sont souvent difficiles à résoudre en CC à cause de la monotonie du store. En particulier, les programmes de type client-serveur sont souvent écrits en utilisant des listes pour implanter des variables impératives (i.e. non monotones). Nous proposons ci-dessous une version plus naturelle de ce problème.

$$
\begin{array}{lll}
client(a) & :: & \exists_b(send(a,b) \parallel \exists_{\vec{y}} receive(b,\vec{y}).action(\vec{y})) \\
serveur(a) & :: & \exists_b(receive(a,b).\exists_{\vec{y}} send(b,\lambda\vec{y}c) \parallel serveur(a))
\end{array}
$$

Un but type consistera en un certain nombre de clients et de serveurs partageant un même paramètre a et s'exécutant en parallèle. a est le canal sur lequel les clients envoient leurs requètes. La confidentialité est assurée par la définition, au début de la transaction, d'un nouveau canal (b) qui est envoyé au serveur. Ce canal est donc un canal privé, connu seulement du client et du serveur. La réponse sera communiquée par son intermédiaire.

Le programme précédent ne modélise en fait que le protocole de communication. Le client ne pose pas de requête et le serveur n'effectue pas de calcul avant d'envoyer sa réponse. Nous donnons ci-dessous une version plus effective. Cet exemple montre le mécanisme de calcul local en MCC ainsi que celui de transmission du résultat.

$$
\begin{array}{lll}
client(a) & :: & \exists_{b,c}(send(a,(b,c)) \parallel \exists_{\vec{y}} send(b,\lambda\vec{y}request) \\
 & & \parallel \exists_{\vec{z}} receive(c,\vec{z}).action(\vec{z})) \\
serveur(a) & :: & \exists_{b,c}(receive(a,(b,c)).\exists_{\vec{y},\vec{z}}(receive(b,\vec{y}) \parallel calcul(\vec{y},\vec{z}) \\
 & & \parallel send(c,(result,\vec{x})) \parallel serveur(a))
\end{array}
$$

avec $\vec{x} \subseteq \vec{y} \cup \vec{z}$.

$\lambda\vec{y}request$ est une abstraction de contrainte qui contient – qui code – la requète du client. la procédure $calcul$ effectue le calcul demandé sur un store local formé des variables des vecteurs $\vec{y}$ et $\vec{z}$. Le résultat est envoyé au client par l'intermédiaire d'un opérateur de store ($result$) qui porte sur certaines variables de ce store local.

6.3 *Le problème du diner des philosophes*

Le problème du diner des philosophes est classique en concurrence. n philosophes sont assis autour d'une table ronde. Il y a une fourchette entre chacun d'eux. Chaque philosophe a besoin des deux fourchettes situées sur ses cotés pour manger. Le problème consiste à réaliser l'accès en exclusion mutuelle des philosophes aux fourchettes.

Nous donnons une version du diner des philosophes qui combine communication par messages et communication ask/tell. Elle est inspirée d'une version en Linda ([CAR 89]) en ce qu'elle repose sur l'existence de $n-1$ jetons pour n philosophes. Un philosophe doit posséder un jeton pour prendre les fourchettes. Ce petit "trick" rend l'exclusion mutuelle plus facile à assurer car l'atomicité de la prise des deux fourchettes n'est plus nécessaire. Ce programme illustre aussi la reconfiguration dynamique de réseau de communication, ce dernier n'étant pas fixé par le but – agent initial – mais crée par les procédures $init$ et $init2$.

Les fourchettes sont modélisées par des canaux. La disponibilité d'une fourchette f est symbolisée par la présence d'un agent $send(f, taken)$, où $taken$ est une constante particulière du système de contraintes. L'action de prendre la fourchette f consiste en la consommation de l'agent $send(f, taken)$ (réception sur f). Le philosophe libèrera la fourchette en introduisant un nouvel agent $send(f, taken)$.

Les tickets sont modélisés par un canal t sur lequel sont initialisés $n-1$ agents $send(t)$. Ceux-ci étant utilisés dans un but de synchronisation pure, l'abstraction de contrainte transmise est sans importance.
$send(t)$ (resp. $receive(t)$) est un sucre syntaxique pour $\exists_x send(t, x = const)$ (resp. $\exists_x receive(t, x)$), $const$ étant une constante quelconque du système de contraintes.

$$
\begin{aligned}
init(n) :: \exists_{a,t,f}(\ & send(a,1) \\
& \parallel send(f, taken) \\
& \parallel init2(a,t,f,f,n))
\end{aligned}
$$

$$
\begin{aligned}
init2(a,t,f,f',n) :: \exists_{n'} receive(a,n').(& \\
& ask(n' < n).\exists_{f'',n''}(send(f'', taken) \\
& \qquad \parallel send(t) \\
& \qquad \parallel phil(f', f'', t, n') \\
& \qquad \parallel n'' = n' + 1 \\
& \qquad \parallel send(a, (proj, n'')) \\
& \qquad \parallel init2(a,t,f,f'',n)) \\
& \parallel ask(n' = n).phil(f', f, t, n))
\end{aligned}
$$

$$
\begin{aligned}
phil(f,f',t,n) :: \exists_{x,x'}\ (receive(t).(\ & receive(f,x).true \\
& \parallel receive(f',x').true \\
& \parallel ask(x = taken \wedge x' = taken).(\\
& \qquad send(f, taken) \\
& \qquad \parallel send(f', taken) \\
& \qquad \parallel send(t) \\
& \qquad \parallel phil(f,f',t,n))))
\end{aligned}
$$

Le but est : $\exists_n(n = N \parallel init(n))$

Les procédures $init$ et $init2$ servent à initialiser les agents $phil$ ainsi que le réseau de communication. Après l'initialisation, nous obtenons l'agent suivant (pour N=2) :

$$
\begin{aligned}
\exists_n \exists_{t,f_1,f_2}(\ & n = 2 \parallel send(t) \parallel send(f_1, taken) \parallel send(f_2, taken) \\
& \parallel \exists_{n_1}(n_1 = 1 \parallel phil(f_1, f_2, t, n_1)) \\
& \parallel \exists_{n_2}(n_2 = 2 \parallel phil(f_2, f_1, t, n_2)))
\end{aligned}
$$

6.4 *L'algorithme d'élection de Chang et Roberts*

Il sagit d'un algorithme classique d'élection sur un anneau unidirectionnel. On suppose qu'à chaque site est associé un identificateur unique et que l'on dispose d'un ordre total sur les identificateurs. Le problème consiste à trouver le site d'identificateur maximal et à annoncer son identité. Les sites ne connaissent aucune information structurelle concernant le réseau de communication, ni l'ordre N du réseau, ni l'identité de leurs voisins. Les sites savent seulement distinguer localement leurs canaux d'entrée (site prédécesseur) et de sortie (site successeur).

Afin de simplifier cet exemple, nous ne modélisons pas l'initialisation de l'algorithme. Le but consiste en un ensemble de *sites* en composition parallèle, chacun envoyant son identificateur à son successeur. Lorqu'un site reçois un message, trois situations peuvent se produire.

- Si son identificateur propre est inférieur à celui reçu alors le message est "éteint".
- S'il est supérieur alors le message est transmis à l'identique.
- S'il est égal alors le site a gagné l'élection et envoie un message annonçant qu'il est l'élu.

$$
\begin{array}{l}
CR(a,b,id) :: \exists_{y,winner}(\ receive(a,y).(\\
\qquad \| \ ask(y < id).CR(a,b,id) \\
\qquad \| \ ask(y > id).(send(b,(proj,y)) \parallel CR(a,b,id)) \\
\qquad \| \ ask(y = id).(\ winner = id \\
\qquad\qquad \| \ \exists_x\ (x = won(id) \parallel send(b,(proj,x))) \\
\qquad\qquad \| \ \exists_x receive(a,x).vrai)) \\
\qquad \| \ ask(\exists_x y = won(x)).\exists_x(\ y = won(x) \\
\qquad\qquad \| \ winner = x \\
\qquad\qquad \| \ send(b,(proj,y))))
\end{array}
$$

Le but est : $\exists_{a_1 \dots a_N}(\ \exists_{id_1}(CR(a_1,a_2,id_1) \parallel id_1 = Id_1 \parallel send(a_2,Id_1))$
$\quad \| \ \dots$
$\quad \| \ \exists_{id_N}(CR(a_N,a_1,id_N) \parallel id_N = Id_N \parallel send(a_1,Id_N)))$

Il est à noter que les exemples présentés dans cette section sont déterministes. Le non-déterminisme est un problème délicat dans un cadre distribué : le backtracking distribué avec communication, bien qu'envisageable, semble bien difficile à implanter d'une manière raisonnablement efficace. Le modèle MCC est donc plutôt orienté vers l'indéterminisme. Notons cependant que, le système de transitions présenté à la section 4 ne faisant que décrire les calculs possibles, cette discussion (indéterminisme vs non-déterminisme) se place en aval de cet article, une notion d'observable pouvant être définie pour l'un ou l'autre cas.

7 Notion de distribuabilité

Les canaux ont été définis comme étant des variables du système de contraintes. En effet, il n'est pas nécessaire d'établir au niveau du langage la distinction entre canaux et variables utilisées dans des contraintes. Dans un cadre distribué, nous voudrions limiter au maximum – voir même totalement éviter – la résolution distribuée de contraintes. Ceci nous conduit à nous intéresser à une classe particulière de programmes : les programmes que nous appelons *distribuables*. Intuitivement, un programme est distribuable si il est une composition parallèle d'agents pouvant s'exécuter indépendamment les uns des autres sauf en ce qui concerne la communication par messages.

Cette notion peut se définir formellement au niveau du système de transitions associé à un programme. Un agent $A \parallel B$ est *distribuable* en A et B si, récursivement, toute réduction de $A \parallel B$ est soit une réduction de A, soit une réduction de B, soit une communication par messages entre A et B. Pour des raisons de place, nous ne dévelopons pas ce sujet dans cet article. Le lecteur intéressé est invité à se reporter à [RET 97]. Notons cependant que les programmes présentés dans la section 6 sont distribuables selon cette notion. Par exemple, pour le programme *client serveur*, un but qui est la composition parallèle de clients et de serveurs est distribuable en l'ensemble de ses clients et de ses serveurs.

Cette notion de distribuabilité repose sur l'intuition simple que l'absence de résolution distribuée de contraintes devrait grandement faciliter une implantation distribuée. Dans ce cas, il reste la nécessité d'établir l'égalité des canaux de communication lors de la communication par messages. Cela s'effectue cependant sur un cas très simple de résolution de contraintes.

8 Conclusion

Nous proposons un paradigme de calcul concurrent par contraintes qui intègre langages CC et communication par messages. La principale motivation de notre travail est la modélisation de systèmes distribués. Le mécanisme de communication par messages du modèle MCC est à rapprocher de celui du π-calcul ([MIL 89]) dont nous nous sommes d'ailleurs inspiré. On peut voir le premier comme une extension du second, les objets informatiques transmis n'étant plus seulement des canaux mais des (abstractions de) contraintes. D'autres part, une relation de bisimilarité a été définie sur une restriction des MCC (cf. [RET 97]). Nous encodons le π-calcul asynchrone de [AMA 96] et prouvons que cet encodage est complètement adéquat[2].

Beaucoup des travaux concernant la distribution des langages de Programmation Logique et de Programmation Logique avec Contraintes se placent d'un point de vue

2. fully abstract

implantation. Parmi les travaux concernant la construction d'un langage adapté au problème de la distribution, citons [PER 84] et [BRO 93]. Les MCC peuvent être vus comme en étant une extension. Des différences importantes existent cependant (hormis le fait de se placer dans un cadre concurrent avec contraintes). Ils utilisent aussi un ensemble de canaux. La communication s'y effectue cependant non pas par envoie unidirectionnel d'un message mais par unification de deux termes. Cette approche est plutôt de style *blackboard* et est ainsi plus proche de celle de LINDA ([CAR 89]).

Les MCC offrent de nombreuses perspectives de développement. Une étude précise d'implantation reste à faire. La construction de modèles de calcul distribués est un problème d'actualité. Parmi les applications possibles, citons par exemple la modélisation de résolution distribuée de contraintes.

Bibliographie

[AMA 96] AMADIO R. M., CASTELLANI I., et SANGIORGI D., « On Bisimulations for the Asynchronous π-calculus », In *proc. of Concur'96*, pp. 147–162, 1996.

[BER 92] BERRY G. et BOUDOL G., « The Chemical Abstract Machine », *Theoretical Computer Science*, vol. 96, :217–248, 1992.

[BRI 96] BRIM L., GILBERT D., JACQUET J., et KŘETÍNSKÝ M., New Versions of Ask and Tell for Synchronous Communication in Concurrent Constraint Programming, Rapport Technique 1996/03, City University, Dept. of Computer Science, London, 1996.

[BRO 93] BROGI A. et CIANCARINI P., « Distributed Logic Programming », *Journal of Logic Programming*, vol. 15, n° 4:295–335, 1993.

[CAR 89] CARRIERO N. et GELERNTER D., « Linda in Context », *Communications of the ACM*, vol. 32, n° 4, April 1989.

[FdB 93] BOER F., KOK J., PALAMIDESSI C., et RUTEN J., « Non-momotonic Concurrent Constraint Programming », In *Proc. of ILPS'93*, MIT Press, 1993.

[FdB 97] BOER F., PIERRO A., et PALAMIDESSI C., « An Algebraic Perspective of Constraint Logic Programming », *Journal of Logic and Computation*, vol. 7, n° 1:1–38, 1997.

[JAF 87] JAFFAR J. et LASSEZ J.-L., « Constraint Logic Programming », In *proc. of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 111–119, ACM, 1987.

[LAN 92] LANEVE C. et MONTANARI U., « Mobility in the cc-paradigm », In *proc. of Mathematics Foundations of Computer Science*, 1992.

[MIL 89] MILNER R., PARROW J., et WALKER D., A Calculus of Mobile Processes, Parts I and II, Rapport Technique ECS-LFCS-89-85 et -86, LFCS, Edinburgh University, 1989.

[PER 84] PEREIRA L. et NASR R., « Delta-PROLOG: A Distributed Logic Programming Language », In *proc. of International Conference on Fifth Generation Computer systems*, pp. 283–291, ICOT, 1984.

[RET 97] RÉTY J.-H., Langages Concurrents avec Contraintes : Communication par Messages et Distribution, thèse de doctorat, Université d'Orleans, Janvier 1997.

[SAR 89] SARASWAT V., Concurrent Constraint Programming, thèse de doctorat, Carnegie-Mellon University, 1989.

[SAR 90] SARASWAT V., KAHN K., et LEVY J., « Janus: a Step Towards Distributed Constraint Programming », In *proc. of the North American Conference on Logic Programming*, 1990.

[SAR 91] SARASWAT V., RINARD M., et PANANGADEN P., « Semantic Foundations of Concurrent Constraint Programming », In *proc. eighteen annual ACM symposium on Principles of Programming Languages*, pp. 333–353, 1991.

[SAR 92] SARASWAT V., « The Category of Constraint Systems is Cartesian Closed », In *proc. 7th IEEE Symp. on Logic in Computer Science*, 1992.