Fundamenta Informaticae 34 (1998) 321–344
IOS Press

Distributed Concurrent Constraint Programming

Jean-Hugues Réty*
INRIA - Rocquencourt, B.P. 105,
F–78153 Le Chesnay cedex,
e-mail: rety@margaux.inria.fr

Abstract. We investigate the problem of building distributed systems within the Concurrent Constraint programming framework (CC). CC languages form a shared memory constraint-based model of calculus and some of its main features, especially the global nature of the *store* and the communication mechanism, are impediments to distributed implementation. We think it to be necessary to introduce some new features in CC languages to build up a language suitable for distributed programming. We first discuss the solutions previously studied within the Logic Programming framework. Then, we propose the introduction of an asynchronous message-based communication mechanism. It smoothly integrates within CC, constraints allowing for a logical formulation of message-based communication, and induces notions of local computation, local store, and site.

Keywords: Distributed languages, Concurrent languages, Concurrent Constraint languages.

1. Introduction

The design of declarative programming languages suitable for tackling the problems of distribution is a very important issue. Research in the field of distribution, however, is hardly influenced by declarative programming: distributed programming languages are still mainly based on imperative programming.

A straightforward way of designing such a language is to take a declarative programming language and to distribute it in some way, without modifying its syntax and semantics. Such

*Current affiliation: CWI, P.O. Box 94079, 1090 GB, The Netherlands. e-mail:jean-hugues.rety@cwi.nl. Address for correspondence: CWI, P.O. Box 94079, 1090 GB, The Netherlands

an approach is attractive because of its simplicity but may not fulfill our expectations: the programmer wants to have some control on runtime distribution, and to reason on distribution as well. Moreover, we must realize that efficient distributed programs are not directly derived from non-distributed ones but are built along completely different ways of thinking: the language must propose the right tools to the programmer. A possible approach is to introduce some annotations in the language (with no modification of the semantics) but this seems to be inappropriate to Concurrent Constraint languages (CC) where the notion of global store is by itself an impediment to distributed implementation.

Another approach, the one we consider in this article, is to modify the language and its semantics. Most of the work concerning distribution within the Logic Programming framework falls into this class – mainly by adding some new communication mechanisms. This approach allows to escape from the – too rigid – framework of existing declarative programming languages. The risk is then to add the language with some ad hoc features: a major issue is that this extension must remain *as declarative as possible*.

Designing such a programming language requires some commitments among which we think the following points to be central:

- What kind of communication do we want between sites (by *site*, we mean here: a local memory together with some local processes acting on it.). This issue can be subdivided as follows:

  How source and target sites are referred to for communication?
  Standard methods used in the field of distributed programming are naming of sites and channel-based communication. In both cases, referencing may be static or dynamic. i.e. names (of sites or channels) can be physical or logical, logical naming making it easier to model network reconfiguration.

  How communication of information is achieved: is it synchronous, asynchronous, uni- or bi-directional? Do we make use of message-based communication, communication via some blackboard, shared-variables based *ask/tell* communication from CC languages, ...

- How can independence of distributed computations be insured?
  We need some criteria to decide whether it is possible to execute a set of processes in a distributed way or not. For a "distributable" program under these criteria, the language must offer the guarantee of an efficient distributed implementation.

Different answers can be given to these questions inside the Logic Programming framework. Below, we give an overview of the main proposed methods.

1.1. Logic Programming and distribution

We consider two large classes of approaches and in each case we give a brief description of the choices made with respect to the questions posed above.

Blackboard approaches and LINDA-based ([7]) approaches:
The notion of blackboard, initially developed in artificial intelligence, is used in logic program-

ming for communication between distributed processes (e.g. μLog [9], shared-Prolog [5],...). The blackboard is here a space of terms (tuples) and basic operations are writing, reading and reading with consumption of a tuple. Communication of information is achieved via matching between a tuple in the blackboard and a tuple argument of some reading operator. Then, a central issue is not to create, via this mechanism, new links between non closed variables from different processes (in order to insure independence of further computations). This is secured by communicating ground information only (except in μLog where non-ground terms are renamed before being put in the blackboard).

Let us notice that the referencing of communication actors in LINDA can be specialized, for instance to be turned into a channel-based communication mechanism with matching, by adding each tuple with an extra field (the channel). An equivalent solution, used in [5] is to consider a set of blackboards referred to by names.

Approaches inspired by the CSP:
The Communicating Sequential Processes ([13]) model was proposed by Hoare in the late 80's. [20, 6] extend Logic Programming with a communication model inspired by the CSP. They assume a set of communication channels (it can be defined as a subset of the constants of the language) and a notion of message-based synchronous communication via these channels. For instance, in [6], *event goals* of the form $U?e$ and $U'!e'$ (where U, U' are terms and e, e' are channel names) can communicate by simultaneous reduction and unification of U and U' if $e = e'$ and $mgu(U,U')U$, $mgu(U,U')U'$ are ground terms. This latter condition insures independence of distributed computations. Contrary to blackboard and LINDA approaches, communication is here bi-directional. However, the communicated information still is ground.

1.2. Concurrent Constraint Languages and distribution

Concurrent Constraint languages (CC) arose in the late eighties ([23]). They came up from Concurrent Logic Languages and Constraint Logic Programming. CC languages form a shared memory constraint-based model of calculus. As we have mentioned before, some of their main features seem to be impediments to distributed implementation. In particular, communication and synchronization take place via a global *store*. Some computation models are based on a shared memory (e.g. Linda ([7])). This approach relies on the possibility to implement this shared memory in a distributed way but this condition hardly seems to be satisfied by CC languages: on the one hand, the *ask* operator needs a global testing of the entailment relation on constraints. On the other hand, in Saraswat's seminal work, communication is achieved via the so-called *atomic tell* agent: a constraint is published only if it is consistent with the *store*, otherwise *the atomic tell* agent is skipped. In a distributed setting, this either leads to broadcast publication or to distributed constraint solving. Distributed constraint solving appears to be a difficult problem: no efficient and general enough method is expected at now. Although it would be very costly, one could imagine to maintain a copy of the store on each site, but even with this extreme solution, the *atomic tell* would still be a problem. With these motivations, Saraswat proposes (in [23], Sect. 7) the *eventual tell* which is the *atomic tell* with the consistency requirement removed. So, in a distributed setting, whenever an *eventual tell* agent is executed at some site, the constraint may not be immediately communicated to all sites but in finite time.

Another angle from which to consider this problem is the notion of locality of information. A distributed system has to give the possibility of running local computations together with some mechanism for transmitting the results. This is achieved in CC languages by restricting the scope of variables (hiding operator) but this notion of locality is not very effective: the result of a local computation is transmitted via shared variables. This means that this result cannot be separated from the information concerning local variables. Local variables may be simulated inside the classical CC framework by means of the hiding operator: it is enough to use some hidden variables at a site and to communicate to other sites with global variables. However, this would be very difficult to implement because having local and global variables and not publishing local information would need strong "fire walls" (as Saraswat says) to separate the local part from the global part of the *store*. Indeed, constraints on local variables may constrain global variables and, deciding what constraints need to be published may be very complex. A solution might be to use some specific properties of constraint systems – for instance syntactic conditions that capture some independence properties of variables.

A supplementary important element is drawn out in [19] and [3]. In these papers, it is argued that CC with *eventual tell* suffers serious drawbacks in expressing many distributed algorithms. CC with *atomic tell* is better in term of expressiveness but *atomic tell* is a serious impediment in designing distributed implementations. A *get* construct which is an ask with information removal is proposed in the context of Linear Concurrent Constraint programming (cf. [3] for an extensive study of this operator). This construct is argued to offer a reasonable compromise in term of expressiveness while preserving the ability of being implemented in a distributed way. A CC-based distributed language has to face this problem and must provide the programmer with an expressive enough language.

The Janus Language:
The Janus language ([25]) was proposed at the beginning of the 90's. It relies on a completely different approach than the ones considered above for Logic Programming: the basic idea is here to restrict the language rather than to extend it. Just like in the CC model, communication occurs via shared variables. Freedom of communicating information is, however, restricted: only two occurrences of a variable are allowed inside a clause and communication is directional (one of these occurrences being the producer and the other one the consumer). On the other hand, the Janus language imposes some requirements on the constraint system in order to allow for a distributed implementation: a constraint $X = s$ where s contains a variable Y must not constrain Y. The domain of infinite trees satisfies this property but not the Herbrand domain. Global communication via the sharing of variables also forces the language to be very restrictive with constraints in *tell* agents: they have to be of the form $X = E$, E being an expression.

Distributed Oz:
Some recent work aims at designing a distributed version of the Oz language ([28, 10, 30]). A

distributed semantics is proposed, without modification of the language[1], defining migration properties of elements of the language. For instance, variables whose values are defined are replicated to all sites referencing them, and whenever an equality between two variables is produced at one site, it is broadcasted to all sites knowing these variables. As in the case of the Janus language, this choice to keep with communication via shared variables has led to restrictions in the use of constraints. The constraint system is a feature structures domain, but, like in Janus, constraints in *tell* agents must be of the form $X = E$ (see, e.g. [30]) in order not to impose "complex" constraints (e.g. inequality constraints) that would rise up implementation problems in a distributed setting.

1.3. Towards an approach inspired from the π-calculus

We think that the problem of distribution in the CC framework has to be taken up from a different point of view, closer to what has been proposed for logic programming. We do not want to impose conditions on the constraint system. This can only be successfully achieved by escaping from the idea that variables of the constraint system are by themselves channels of communication and that communication occurs via global constraints between these variables. We propose to escape from some basic features of CC languages concerning the global nature of the store, communication and synchronization, by introducing a notion of message-based communication. Just like some works in Logic Programming were inspired from the CSP model, we will take our inspiration from the π-calculus ([17]). The π-calculus is a model of concurrency that models message communication. It stands by itself as an active research domain. We show that the communication mechanism of the π-calculus can be integrated into the CC framework in a homogeneous way, giving the way to a calculus combining concurrent constraint calculus at the local level and message-based communication at the global level.

On the other hand, we think that in a constraint-based framework, it would be too restrictive to communicate values only. We propose a communication mechanism with *constraint abstractions* which allows for communicating general constraints while insuring the independence of distributed computations by absence of sharing of (constraint) variables between sites.

Section 2 defines constraint systems. Section 3 presents a calculus, called *MCC*, combining CC languages and π-calculus style message-based communication. Section 4 defines MCC's operational semantics. Section 5 proposes a novel mechanism for propagating the result of a local computation. Section 6 presents a sub-language of MCC, called *DCC*, for distributed programming. We define in Section 7 a notion of *distributability* and a sufficient syntactic condition for a program to be *distributable*. Some examples of programming in DCC are given in Section 8. Section 9 discusses the contributions of this article.

[1] Note that the Oz language is by itself an extension of the CC paradigm. In particular, it integrates a notion of non monotonic variable called *cell*.

2. Constraint systems

Different formalizations of constraint systems were proposed in the literature ([14, 24, 4]). In this article, we make use of a simple one where a constraint system is a first order language together with a fixed interpretation.

Definition 2.1. A constraint system $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ is composed with a first order language $\mathcal{L}$ closed by conjunction and existential quantification and containing the predicate "=", and by an interpretation $\mathcal{I}$ where "=" is interpreted as equality.

Let $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ be a constraint system, constraints are the formulas in the language. c being a constraint, we note $fv(c)$ its free variables. $\vec{x}$ and $\vec{y}$ being vectors of variables of equal length, $c[\vec{y}/\vec{x}]$ denotes the renaming avoiding capture, of the occurrences of the variables in $\vec{x}$ by the variables in $\vec{y}$.

We define a partial pre-order on the set of constraints, called *entailment relation* and denoted by $\vdash_{\mathcal{C}}$. (Although our formalization of constraint systems, that could be called *logical formalization*, should lead us to use the symbol $\models$, we keep the symbol $\vdash$ because it is the standard notation in the CC languages framework).

Definition 2.2. (entailment relation)
Let $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ be a constraint system. Let c, d be constraints of $\mathcal{C}$.
$c \vdash_{\mathcal{C}} d$ if for every valuation satisfying c, the constraint d is also satisfied (in $\mathcal{I}$).

Let $\mathcal{C}$ be a constraint system, an equivalence relation $\sim_{\mathcal{C}}$ is induced by $\vdash_{\mathcal{C}}$ on the set of constraints. ¿From now on, we will note $\vdash$ the entailment relation and $\sim$ its induced equivalence relation. It does not generate ambiguity since we will consider a single constraint system.

We will need the notion of inconsistent constraint:

Definition 2.3. Let $\mathcal{C} = \langle \mathcal{L}, \mathcal{I} \rangle$ be a constraint system. A constraint c of $\mathcal{C}$ is said to be *inconsistent* if no valuation satisfies it (in $\mathcal{I}$).
Let $const$ and $const'$ be two distinct constants of $\mathcal{L}$, we will note $false$ the inconsistent constraint $const = const'$ and $true$ the tautology $const = const$.

3. The MCC model of calculus

We propose a concurrent constraint model of calculus integrating CC languages (with communication by *ask* and *tell* agents) and asynchronous message-based communication. We call it MCC (Message-based Concurrent Constraint programming).

3.1. The communication paradigm

We introduce two new communication primitives, noted *send* and *receive*, in CC languages. Communication is message-based and is achieved via communication channels. These channels will be actually modeled by variables of the constraint system. Constraints will allow for an elegant and simple formalization of the mechanisms from communication by messages.

The MCC model, although distribution oriented, does not directly specify the repartition of agents. An MCC program will be said to be *distributable* if it satisfies some properties relying on simples intuitions. Intuitively, an MCC program is distributable if it is possible to split it into a network of sites communicating by messages only, each site being a local store on which a ("piece" of) MCC program is executed. Let us consider, in an explanatory view (at the formal level, in the MCC model, variables and channels are objects of same nature and are only distinguished by their use), that variables are partitionned into a set of *constraint variables* and a set of *channel variables*. The parallel composition of some agents (*sites*) not sharing *constraint variables* (they only communicate by messages and the use of constraints is restricted to the local level) is a distributable program. Indeed, what makes it difficult to distribute CC languages is the fact that information is exchanged via constraints. We first define the general MCC model. Then, we will consider a sub-language (Section 6) for distributed programming.

Send and *receive* agents take as arguments the channel on which communication is to occur and the computing object to be transmitted. Inside a constraint programming framework, it seems that this object shall be a constraint by itself. However, this raises a problem: what is the meaning of the received constraint? Previous work in Logic Programming with message-based communication has solved this problem by communicating ground information only. We think this to be too restrictive in a constraint-based calculus. We propose the notion of constraint abstraction to overcome this problem. Let us consider a site A communicating some constraint $x < 5$ to a site B, x being a local variable of A. What we want to communicate is indeed that some variable y, local to B, is lower than five; and then add the constraint $y < 5$ to the store in B. The communicated objects shall thus not be constraints but "constraint abstractions", i.e. constraints abstracted from (some of) their free variables, the receiving site then renaming them with some local variables (Figure 1).

This notion of constraint abstraction may be related to the blackboard communication in μLog ([9]), a LINDA-based Logic Programming language, where terms are renamed with fresh variables before putting them in the blackboard. We extend this idea to general constraints but we allow for constraint abstractions where not all variables are "abstracted". This possibility, restricted to channel variables (see Section 6), will permit the dynamical reconfiguration of the communication network in a π-calculus like style (cf. Example 8.1) while preserving independence of local computations.

We formally define constraint abstractions to be pairs formed with a linear vector of variables and a constraint. What we actually want is some mechanism establishing a correspondence between the variables of the sent constraint and the variables of the receiving site. Among the different possible formalizations, one could assume a total order on variables. This would allow for transmitting constraints rather than constraint abstractions, renaming at the reception of a message being made with respect to that order. The notion of constraint abstraction we use defines a partial order local to each message. Another formalization would be to use a notation

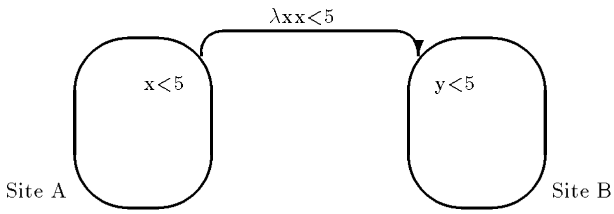

Figure 1. Communication by constraint abstractions

à la De Bruijn. This would be another way to define an order local to each message.

Let us consider a constraint system $\mathcal{C}$.

Definition 3.1. The set of *constraint abstractions* on $\mathcal{C}$ is :
$\mathcal{C}_{abs} = \{(\vec{x}, c) | c \in C,\ \vec{x} \subseteq fv(c)\ and\ \vec{x}\ linear\}$

¿From now on, we will note $\lambda\vec{x}c$ the constraint abstraction $(\vec{x}, c)$. (This notation is not to be confused with lambda abstraction.)
Application of a vector of variables to a constraint abstraction is defined to be the constraint obtained by renaming the abstracted variables:

Definition 3.2. Let $\lambda\vec{x}c$ be a constraint abstraction. Let $\vec{y}$ be a vector of variables of the same length as $\vec{x}$.
The application of $\vec{y}$ to $\lambda\vec{x}c$ is the constraint $c[\vec{y}/\vec{x}]$.

3.2. Syntax

Let us consider a constraint system $\mathcal{C}$.
MCC programs on $\mathcal{C}$ are defined by the following syntax:

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

c is a constraint, x is a variable, $\vec{x}$ and $\vec{y}$ are vectors of variables, $\lambda\vec{z}c$ is a constraint abstraction, p is a symbol of procedure belonging to a set of symbols of procedure associated with an arity function.

A program is a set of (procedure) declarations together with an agent called *goal* or *initial agent*. Let $Prog = Decl.Goal$ be a program, for every symbol of procedure p with arity n appearing in an agent of $Prog$, there exists a single declaration $p(x_1, ..., x_n) :: A$ in $Decl$ satisfying $fv(A) \subseteq \{x_1, ..., x_n\}$. We note $Agents$ the set of agents defined by this syntax.

Send and *receive* agents are called *communication agents*. The first argument of a communication agent A is called the *channel of* A. Other agents and agent constructors are the ones from CC languages with *eventual tell*. $\exists$ is the hiding operator, it corresponds to restriction of variables scoping.

4. Operational semantics

The operational semantics style we use is inspired by the Chemical Abstract Machine ([2]). Actually, we only retain from the CHAM the idea of working with a structural congruence which takes into account equivalence of constraints, variable scoping, and syntactic features like the commutative and associative nature of parallel composition or α-renaming. This style of operational semantics was already present in ([8]) and was intensively used for the definition of the underlying model of calculus of Oz ([27, 18]).

This is equivalent to defining a transition system with SOS style inference rules on the quotient set of agents by the structural congruence. The concept of "floating molecules" from the CHAM is well suited to message-based communication whose reduction rules require the syntactic juxtaposition of several atomic agents. Note also that many of the ideas developed in the CHAM were already present in [1].

4.1. Structural congruence

The *structural congruence* $\equiv$ on agents is the least congruence satisfying the following laws:

- $\langle Agents, \|, true\rangle$ is a commutative monoid.
- $\exists_x A \equiv \exists_y A[y/x]$ if y not free in A
  where $A[y/x]$ denotes the renaming of x by y within A.
  $\exists_x A \parallel B \equiv \exists_x (A \parallel B)$ if x not free in B
  $\exists_x \exists_y A \equiv \exists_y \exists_x A$
  $\exists_x A \equiv A$ if x not free in A.
- $c \equiv d$ if $c \sim d$
  $c \parallel d \equiv c \wedge d$

4.2. Inference rules

Inference rules for MCC are presented Figure 2. The (communication) rule reduces the parallel composition of a *send* and a *receive* agent in the presence of a constraint entailing the equality of their communication channels. The (ask) rule is the one from CC languages.

Let $Prog = Decl.Goal$ be an MCC program. These rules define a transition system $T = \langle Agents, \rightarrow\rangle$, where $\rightarrow$ is the least relation on agents satisfying the inference rules.

Note that the language we define has no choice operator. As the examples in Section 8 will show, an explicit choice operator is not necessary because of the non-determinism of message-based communication. Indeed, blind choice (denoted below $+_b$) can be simulated in the following way ($send(a)$ (resp. $receive(a)$) is a syntactic sugar for $send(a, \lambda\emptyset true)$ (resp. $receive(a, \emptyset)$). i.e. pure synchronization):

$$A +_b B \equiv \exists_a(send(a) \parallel receive(a).A \parallel receive(a).B),$$
where a is not free in A and in B.

It is also possible to define a guarded choice operator:

$$\Sigma_1^n ask(c_i).A_i \equiv \exists_a(send(a) \parallel ask(c_1).receive(a).A_1 \parallel ... \parallel ask(c_n).receive(a).A_n),$$
where $\forall 1 \leq i \leq n$, a is not free in A_i.

It is not hard to show these simulations to be safe with respect to the notion of *observables* defined in [22]: let us consider some context $C[.]$ (i.e. an agent with a hole to place a sub-agent). Let us consider two agents A and B and a communication channel (i.e. a variable) not free in A and B. The agents $C[A +_b B]$ and $C[\exists_a(send(a) \parallel receive(a).A \parallel receive(a).B)]$ have the same observables.
Likewise, let us consider n agents $ask(c_1).A_1, ..., ask(c_n).A_n$. The agents $C[\Sigma_1^n ask(c_i).A_i]$ and $C[\exists_a(send(a) \parallel ask(c_1).receive(a).A_1 \parallel ... \parallel ask(c_n).receive(a).A_n)]$ have the same observables. Proofs are by induction and do not raise major difficulties.

Let us also notice that Milner ([16]) shows that a minimal version of the π-calculus without choice operator has an important expressiveness (especially, it allows for the encoding of the λ-calculus). However, it hardly seems possible to simulate the one step choice operator (which is used in the standard presentation of the π-calculus) with the mechanism we used above: the one step choice operator would require not only an ask guard checking but a test of non-suspension of a somewhat agent (such a test is not decidable).

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

$$\frac{A \equiv B \quad A \rightarrow A' \quad A' \equiv B'}{B \rightarrow B'} \quad \text{(equiv)}$$

$$\frac{d \vdash c}{d \parallel ask(c).A \rightarrow d \parallel A} \quad \text{(ask)}$$

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

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

Figure 2. MCC inference rules

5. Store operators

The language defined above compels the programmer to completely specify the constraints to be communicated. An actual distributed system will need to communicate parts of local *stores* rather than user-defined constraints. Typically, one would like to run a computation at a particular site and then communicate the result to another site. In such a case, what is sent is still a constraint abstraction but it cannot be expressed in a direct way because it depends on the computed *store*. The tools we propose below satisfy this need. In particular, we extend *send* agents to take as argument not only constraints but also *store operators*.

Definition 5.1. A *store operator* is an application $\theta : \vec{v} \times c \mapsto d$,
c, d being constraints and $\vec{v}$ being a vector of variables, such that:
if $\theta(\vec{v}, c) = d$ then $fv(d) \subseteq \vec{v}$.

We do not discuss here what kind of store operators would be useful. The simplest one – and its practical interest is obvious – is the projection operator.

Definition 5.2. Let c be a constraint. The projection of c on a vector of variables $\vec{x}$ is:
$proj(\vec{x}, c) = \exists_{fv(c)/\vec{x}} c$

We introduce a new kind of agents called *link agents* and noted $link((\theta, \vec{x}), \vec{y})$. Such agents set up a "link" between two vectors of variables: information produced on the variables $\vec{x}$ is transmitted to the variables $\vec{y}$ via the store operator θ. We are talking about "links" because, once introduced, a *link* agent cannot be removed and creates some kind of a "gateway" between variables of different "sites". The semantics of *link* agents is described by a rule which propagates the available information on the variables $\vec{x}$ to the variables $\vec{y}$. A *link* agent may be either directly introduced in the program or set up by message communication. We extend message-based communication to handle store operators.

The MCC syntax is extended with the following agents:

$send(u, (\theta, \vec{x}))$ for θ store operator and $\vec{x}$ vector of variables,
$link((\theta, \vec{x}), \vec{y})$ for θ store operator and $\vec{x}, \vec{y}$ vectors of variables of equal lengths.

The semantics of these new agents is defined as follows:

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

$$\frac{d \vdash u = v \qquad length(\vec{x}) = length(\vec{y})}{\begin{array}{c} d \parallel send(u, (\theta, \vec{x})) \parallel receive(v, \vec{y}).B \\ \rightarrow \\ d \parallel link((\theta, \vec{x}), \vec{y}) \parallel B \end{array}} \qquad (\text{communication}_{link})$$

The (communication$_{link}$) rule describes the introduction of a *link* agent via message communication. The (propagation$_{link}$) rule represents a simple operational mechanism: the propagation of the result of a local computation. This rule is not very satisfactory because it introduces infinite derivations. We have: if the rule (propagation$_{link}$) applies to an agent A then there exists an infinite derivation for A. Practically, it does not raise problems (the underlying operational mechanism is clear). However, it raises a semantical problem. A solution can be given by defining an operational semantics whose states are pairs formed with an agent and a store, like for instance in [26]. However, this would complicate the description of message-based communication which requires the syntactic juxtaposition of *send* and *receive* agents.

One may argue that *link* agents reintroduce in the calculus the communication via shared variables we were willing to escape from. Indeed, it hardly seems possible to completely suppress this kind of communication while not dramatically cutting back the expressiveness. Then, the issue is to provide the programmer with a precise and flexible tool to control it. On the other hand, like in concurrent logic languages (e.g. [12]) and Janus where shared variables are moded, this incremental form of communication has to be unidirectional. These languages impose this requirement on all variables. The MCC model gives the programmer the entire CC programming paradigm while providing unidirectional incremental communication where it is needed.

This comparison with concurrent logic languages and Janus gives a hint about implementation. Our claim (although no implementation work has been done so far) is that the DCC language (see next section) can be efficiently implemented in a distributed way. Distributed implementations of concurrent logic languages show that our *link* agent construct is not a stumbling block to distributed implementation.

Examples in Section 8 show some practical use of link agents. Projection operators in Examples 8.3 and 8.4 (Section 8) transmit the value of a variable via message communication. This is very important because it replaces shared variables (like for instance in Janus or Distributed Oz) by a more flexible tool. Note that with this mecanism, in the case where the domain of the variable is progressively shrunk by the (local) constraint solver, this partial information (the domain of the variable) can be progressively transmitted as long as it shrinks. Then, what we do communicate here are constraints and not only values (the domain of a variable is by itself a constraint on this variable). The second version of Example 8.2 gives a more complexe use of link agents.

6. DCC: a sub-language for distributed programming

The model we have defined so far does not impose any restriction on the use of constraints and constraint abstractions. The distributed model of calculus we have in mind, as explained in the discussion Subsection 3.1, requires a precise distinction of which variables are used as communication channels and which ones are used in local constraint computations. Therefore, we define a sub-language of MCC called DCC (Distributed Concurrent Constraint Programming). We suppose a partition of the set of variables into a set of *channel variables* $\mathcal{V}_{ch}$ and a set of *constraint variables* $\mathcal{V}_{ct}$. Then, we impose constraints c in *tell* and *ask* agents to satisfy $fv(c) \subseteq \mathcal{V}_{ct}$ except if c is an equality between two channel variables. In the same way, we impose constraint abstractions $\lambda\vec{x}c$ to satisfy $fv(c) \subseteq \vec{x}$ except if c is $x = y$ where x and y are channel variables.

DCC programs on some constraint system $\mathcal{C}$ are defined by the following syntax:

$$
\begin{array}{lll}
Program & ::= & Declaration.Agent \\
Declaration & ::= & \epsilon \mid p(\vec{x}) :: Agent \mid Declaration.Declaration \\
Agent & ::= & c \mid ask(c).Agent \mid u = v \mid ask(u = v).Agent \\
 & & \mid send(u, \lambda\vec{t}c) \mid receive(u, \vec{y}).Agent \\
 & & \mid send_{ch}(u, \vec{v}) \mid receive_{ch}(u, \vec{v}).Agent \\
 & & \mid send(u, (\theta, \vec{y})) \mid link((\theta, \vec{y}), \vec{z}) \\
 & & \mid Agent \parallel Agent \mid p(\vec{x}) \mid \exists_x Agent
\end{array}
$$

u and v are channel variables, c is a constraint satisfying $fv(c) \subseteq \mathcal{V}_{ct}$, $\vec{y}$ and $\vec{z}$ are vectors of constraint variables, $\vec{v}$ is a vector of channel variables, $\lambda\vec{t}c$ is a constraint abstraction satisfying $fv(c) \subseteq \vec{t}$, x is a variable, $\vec{x}$ is a vector of variables, θ is a store operator.

The intuitive meaning of $send_{ch}$ (resp. $receive_{ch}$) is "send the channels" (resp. "receive the channels"). The *send* and *receive* agents are restricted to communicate information on constraint variables only.

The DCC semantics is defined by the one of MCC. Indeed, DCC is a proper subset of MCC: a DCC program is transformed into an MCC program as follows:
$receive_{ch}$ agents are transformed into MCC's *receive* agents.
$send_{ch}(u, \vec{v})$ is transformed into the MCC agent $\exists_{\vec{w}} send(u, \lambda\vec{w}\vec{v} = \vec{w})$.

7. Distributability

In this section, we pay attention to a particular class of DCC programs: the programs we call *distributables.* Intuitively, a program is distributable if it is a parallel composition of agents that execute independently from each other except for message communication.

This notion can be formally defined at the level of the transition system associated to a program: an agent $A \parallel B$ is *distributable* into A and B if, recursively, every reduction of $A \parallel B$ is either a reduction of A, a reduction of B, the communication of a message or a communication via a *link* agent between A and B.

Definition 7.1. An agent $A \equiv \exists_{\vec{y}}(A_1 \parallel A_2 \parallel A_3 \parallel ... \parallel A_n)$ is distributable into $A_1, A_2, A_3, ..., A_n$ if every reduction $A \to B$ satisfies:

- either there exists an agent B_1 such that $A_1 \to B_1$,
  $B \equiv \exists_{\vec{y}}(B_1 \parallel A_2 \parallel A_3 \parallel ... \parallel A_n)$, and
  B is distributable into $B_1, A_2, A_3, ..., A_n$, or

- $A_1 \equiv \exists_{\vec{x1}}(send(u, \lambda\vec{y}c) \parallel \tilde{A}_1)$, $A_2 \equiv \exists_{\vec{x2}}(receive(v, \vec{z}).\hat{A}_2 \parallel \tilde{A}_2)$,
  $B \equiv \exists_{\vec{y}}\exists_{\vec{x1}}\exists_{\vec{x2}}(\tilde{A}_1 \parallel c[\vec{z}/\vec{y}] \parallel \hat{A}_2 \parallel \tilde{A}_2 \parallel A_3 \parallel ... \parallel A_n)$,
  and B is distributable into
  $\tilde{A}_1, (c[\vec{z}/\vec{y}] \parallel \hat{A}_2 \parallel \tilde{A}_2), A_3, ..., A_n$, or
- $A_1 \equiv \exists_{\vec{x1}}(send(u, (\theta, \vec{t})) \parallel \tilde{A}_1)$, $A_2 \equiv \exists_{\vec{x2}}(receive(v, \vec{z}).\hat{A}_2 \parallel \tilde{A}_2)$,
  $B \equiv \exists_{\vec{y}}\exists_{\vec{x1}}\exists_{\vec{x2}}(link((\theta, \vec{t}), \vec{z}) \parallel \tilde{A}_1 \parallel \hat{A}_2 \parallel \tilde{A}_2 \parallel A_3 \parallel ... \parallel A_n)$,
  and B is distributable into
  $(link((\theta, \vec{t}), \vec{z}) \parallel \tilde{A}_1), (\hat{A}_2 \parallel \tilde{A}_2), A_3, ..., A_n$, or
- $A_1 \equiv \exists_{\vec{x1}}(send_{ch}(u, \vec{w}) \parallel \tilde{A}_1)$, $A_2 \equiv \exists_{\vec{x2}}(receive_{ch}(v, \vec{t}).\hat{A}_2 \parallel \tilde{A}_2)$,
  $B \equiv \exists_{\vec{y}}\exists_{\vec{x1}}\exists_{\vec{x2}}(\vec{w} = \vec{t} \parallel \tilde{A}_1 \parallel \hat{A}_2 \parallel \tilde{A}_2 \parallel A_3 \parallel ... \parallel A_n)$,
  and B is distributable into
  $(\vec{w} = \vec{t} \parallel \tilde{A}_1), (\vec{w} = \vec{t} \parallel \hat{A}_2 \parallel \tilde{A}_2), (\vec{w} = \vec{t} \parallel A_3), ..., (\vec{w} = \vec{t} \parallel A_n)$, or
- $A_1 \equiv \exists_{\vec{x}}(d \parallel link((\theta, \vec{t}), \vec{z}) \parallel \tilde{A}_1)$,
  $B \equiv \exists_{\vec{y}}(\exists_{\vec{x}}(d \parallel link((\theta, \vec{t}), \vec{z}) \parallel \tilde{A}_1) \parallel (\theta(\vec{t}, d)[\vec{z}/\vec{t}] \parallel A_2) \parallel A_3 \parallel ... \parallel A_n)$,
  and B is distributable into
  $A_1, (\theta(\vec{t}, d)[\vec{z}/\vec{t}] \parallel A_2), A_3, ..., A_n$.

The parts of this definition concerning message communication are of particular interest. Indeed, they highlight the way a distributed semantics of DCC may be built. After communication of a constraint abstraction, the renamed constraint is added to the store of the receiving site. After communication of a store operator, the link agent is added to the agents acting on the sending site. After communication of a channel, the equality between channels is added to all sites – In fact, all we need is to propagate the information (the channels equality) to the sites knowing these variables.

This notion of distributability relies on the simple intuition that absence of distributed constraint solving should greatly facilitate a distributed implementation. The definition above is clear enough to show that a distributable program only requires distributed constraint solving on equalities between communication channels – this is a very simple case of constraint solving. A simple (but not unique) method to implement this equality channel solving is to duplicate the equality constraints between channel variables to the sites referencing them. Absence of shared *constraint variables* is sufficient to insure operational independence of sites (at the constraint resolution level: if the store is consistent, an $ask(c).A$ agent at some site cannot be released by some constraint at another site). Remain the *link* agents. We think that they could benefit from the successful previous work on distributed implementation of (moded) logic variables (e.g. [12]).

Distributability is not decidable but below we define a sufficient syntactic condition for a DCC program to be distributable.

Definition 7.2. An agent is a *site* if it is closed on its constraint variables.

This notion of site is not so-called by accident. The proposition below shows that a parallel composition of sites is distributable into these sites.

Proposition 7.1. *Let $A_1, ..., A_n$ be n sites,*
The agent $A_1 \parallel ... \parallel A_n$ is distributable into $A_1, ..., A_n$.

The proof of this proposition is by structural induction. While the result is intuitively obvious, it is rather long and does not present technical interest. We refer the interested reader to [22] where it is detailed for $n = 2$.

The programs in Section 8 are distributable with respect to this notion.
For the *client-server* program, the parallel composition of some clients and some servers is distributable into the set of its clients and its servers since these ones are sites. Similarly, the initial goal in Example 8.4 is clearly a parallel composition of sites. After initialisation of the dining philosophers problem, we have also a parallel composition of sites.

8. Examples

Examples in this section show the expressiveness of our language. We model some classical problems from distributed and concurrent programming. Message consumption appears to be a widely useful feature missing to CC languages. In particular, it allows for a far more natural simulation of imperative variables than the one (historical list of values) used in concurrent logic languages, CC languages and logic programming in general.

We make use of the following syntactic sugar: if d is a constant of the constraint system then $send(u, d)$ is a shorthand for $\exists_x send(u, \lambda x x = d)$.

8.1. Reconfiguration

Dynamical reconfiguration of the communication network is an important issue for concurrency and distribution. It is modeled in CC with lists (cf. [15]). We achieve reconfiguration in a more natural way by sending channels in a π-calculus like style.

In the following example, two agents p and q share a channel a and p shares a channel b with a third agent r.
Let us consider the following program (declarations for s and t are not specified):

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

and the goal: $\exists_b(\exists_a(p(a,b) \;\parallel\; q(a)) \;\parallel\; r(b))$

After communication between p and r, a is shared by the three agents. Then, $send(a, \lambda \vec{x} c)$ can communicate with one of the other two agents. There is communication of a channel, to be more precise, this corresponds to "scope extrusion" in the π-calculus.

8.2. Client-Server

Many programming problems require some non-monotonicity of information. Due to the monotonicity of the *store*, it is often not easy to solve these kinds of problems in CC. In particular, client-server programs are usually written with lists to implement imperative (i.e. non-monotonic) variables. We propose a more natural version using our communication agents.

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

The goal may be the parallel composition of some clients and some servers sharing the same parameter a. a is the request channel clients use to ask for transactions. Whenever a client sends a request, privacy is secured by defining a new channel and communicating it to the server. This new channel is then a private link between the server and its client through which the answer will be sent.

This program actually only models the communication protocol. The client does not ask for any request and the server runs no computation before sending its answer. Below we give a more effective version. This example shows the MCC mechanism for local computation and its mechanism for transmitting the result.

$$\begin{array}{lll} client(a) & :: & \exists_{b,c}(send_{ch}(a,\overrightarrow{(b,c)}) \parallel \exists_{\vec{y}} send(b,\lambda\vec{y}request) \\ & & \parallel \exists_{\vec{z}} receive(c,\vec{z}).action(\vec{z})) \\ server(a) & :: & \exists_{b,c}(receive_{ch}(a,\overrightarrow{(b,c)}).\exists_{\vec{z}}(receive(b,\vec{y}) \parallel compute(\vec{z}) \\ & & \parallel send(c,(result,\vec{x}))) \parallel server(a)) \end{array}$$

with $\vec{y} \subseteq \vec{z}$ and $\vec{x} \subseteq \vec{z}$.

$\lambda\vec{y}request$ is a constraint abstraction which contains – which codes – the request. The procedure *compute* is run on a local store formed with the variables of $\vec{z}$. The result is propagated to the client via the store operator *result*.

In a practival view, *result* will be the projection operator (or maybe something very close to the projection operator, like in Example 8.3). After having received a request, a server runs a local computation (procedure *compute*) on a local store. The information which is relevent to the client is thus produced on some variables of this local store. Let $\vec{x}$ be these variables. What we want is then to communicate this relevent information, i.e. the projection of the local store on these variables. If $\vec{x}$ is reduced to a single variable, this communication is the communication of the domain of the variable. If $\vec{x}$ is composed of several variables, this may lead to a more powerful communication mecanism of the result of a local computation. Suppose $\vec{x}$ is composed of two variables x_1 and x_2 and that the underlying constraint system is a finite domain constraint system. The local computation may, for instance, produce the constraint $x_1 + 4 = 2x_2$. Transmitting this constraint gives more information to the receiving site than the simple transmission of the domains of the variables.

8.3. The dining philosophers problem

The dining philosophers problem is a classical concurrency problem. n philosophers are sitting around a table. A fork is placed between each of them. In order to eat, a philosopher needs the two forks at his sides. The problem consists in insuring the mutual exclusion access of philosophers to the forks.

The version we give of that problem combines message-based communication and *ask/tell* communication. It relies on the existence of $n-1$ tokens for n philosophers. A philosopher needs to take up a token before trying to take up the forks. This little trick is inspired from a LINDA version of this problem ([7]) and makes it simpler to insure mutual exclusion because atomic picking of the two forks is no longer necessary. This program also shows the dynamical reconfiguration of the communication network: this one is not fixed by the goal but is created by the procedures *init* and *init*2.

Forks are modeled with channels. Availability of a fork f is symbolized by the presence of an agent $send(f, taken)$, where $taken$ is a particular constant of the constraint system. This method is used in the Pict ([21]) version of the dining philosophers problem. To take up a fork f is to consume the $send(f, taken)$ agent (reception on f). The philosopher will release the fork by introducing a new $send(f, taken)$ agent.

Tickets are modeled with a channel t on which $n-1$ $send(t)$ agents are initialized. They are used for pure synchronization, the transmitted constraint abstraction being irrelevant. $Send(t)$ (resp. $receive(t)$) is a syntactic sugar for $send(t, \lambda\emptyset true)$ (resp. $\exists_x receive(t, \emptyset)$).

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

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

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

where $proj^{+1}(x,c) = \exists_y(proj(x,c)[y/x] \wedge x = y+1)$.

The goal is: $\exists_n(n = N \parallel init(n))$

Procedures *init* and *init*2 initialize the *phil* agents and the communication network. After initialisation, the following agent is obtained for N=2 (for sake of clarity, we have suppressed the irrelevent variables coming from procedure call reduction):

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

The use of a store operator in *init*2 permits to communicate the identifiers during initialization without creating bindings between constraint variables from different philosophers.

8.4. Chang and Roberts's election algorithm

This is a classical election algorithm on unidirectional rings. We suppose each site to have a single identifier. There is a total order on identifiers. The problem is to find the site with maximal identifier and to broadcast its identity. Sites do not know any structural information about the communication network, neither its order N nor the identity of their neighbours. They only can locally distinguish between their incoming channel (preceding site) and their outgoing channel (site successor).

For simplicity, we do not model the initialization of the election. Therefore, the goal consists in sites in parallel composition, each one sending its own identifier to its successor. Whenever a site receives a message, three different situations can occur, as follows. If its own identifier is lower than the one carried on by the message then the message is stopped. If it is greater, then the message is transmitted. In both cases the site then waits for another message. If it is equal then the site has won the election and sends a message telling that it is the winner.

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

$$\begin{aligned}
\text{The goal is: } \exists_{a_1 \ldots a_N}(\ & \exists_{id_1}(CR(a_1,a_2,id_1) \parallel id_1 = Id_1 \parallel send(a_2, Id_1))\\
& \parallel\ \ldots\\
& \parallel\ \exists_{id_N}(CR(a_N,a_1,id_N) \parallel id_N = Id_N \parallel send(a_1, Id_N)))
\end{aligned}$$

Notice the use of projection operators for the transmission of values. For instance, at the third line in the definition of CR, y is a variable of the (local) store and is bound to some constant. $send(b, (proj, y))$ sends the value of this variable on the channel b.

9. Discussion and Conclusion

We propose a concurrent constraint calculus integrating message-based communication and CC languages. Our main motivation is to model distributed systems. The communication mechanism we propose extends some previous ideas proposed for distributed logic programming ([6, 9]). In a sense, our proposal lifts this work from Logic Programming to Concurrent Constraint programming and from CSP to the π-calculus. Moreover, to our knowledge, our notion of *link* agent is a novelty. The idea of our formalism is that we shall give the programmer the possibility of programming with constraints at the local level without limitation, and of specifying what information has to be exchanged between sites.

Starting from different motivations (it is intended to be a higher-order and object-oriented extension of CC programming), the Oz programming model appears to have close relationships with the MCC model. The notion of *cell* in Oz can be seen as a restriction of message-based communication and was indeed introduced in this way in an earlier presentation of the Oz computation model ([11]). In annex, we give an encoding of the ρ-calculus ([18]), which is the underlying model of calculus of Oz, into MCC. It turns out that the MCC calculus is expressive enough to simulate the higher-order mechanism of Oz. On the other hand, the reverse encoding seems to be possible except for *link* agents.

The MCC model (to be more precise: a restriction of the MCC model) may be seen as an extension of the π-calculus to handle constraints. The π^+-calculus ([29]) which was recently proposed in this aim is indeed a sublanguage of MCC; without constraint abstractions (messages are restricted to only communicate names or values) and without link agents.

The MCC and DCC models offer numerous perspectives of development. In particular, it may be made object oriented. A precise distributed implementation study has to be done. The construction of models for distributed programming is still with us. Among the possible applications, let us cite for instance the modelling of distributed constraint solving.

Acknowledgements

The author wants to thank Philippe Codognet, David Gilbert and François Fages for their advice and fruitful discussions. Thanks as well to Catuscia Palamidessi, for her detailed comments on a previous version of this article, and to some anonymous referees.

References

[1] J.A. Bergstra, J. Heering, and J.W. Klop. Object-oriented algebraic specification: Proposal

for a notation and 12 examples. Technical Report CS-R8411, CWI, P.O. Box 94079, 1090 GB Amsterdam, The netherlands, 1984.

[2] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[3] E. Best, F.S. de Boer, and C. Palamidessi. Partial order and SOS semantics for linear constraint programs. In *proc. of Coordination'97*, LNCS vol.1282, pages 256–273. Springer-Verlag, September 1997.

[4] F.S. de Boer, A. di Pierro, and C. Palamidessi. An algebraic perspective of constraint logic programming. *Journal of Logic and Computation*, 7(1):1–38, 1997.

[5] A. Brogi and P. Ciancarini. The concurrent language shared prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[6] A. Brogi and P. Ciancarini. Distributed logic programming. *Journal of Logic Programming*, 15(4):295–335, 1993.

[7] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM 32(4)*, April 1989.

[8] A. Corradini, U. Montanari, and F. Rossi. Sharm: Concurrency and hiding in an abstract rewriting machine. In *proc. of Fifth Generation Computer Systems*, pages 887–896, 1992.

[9] K. De Bosschere and J.-M. Jacquet. Multi-prolog: Definition, operational semantics and implementation. In *proc. of International Conference in Logic Programming*, pages 299–313, 1993.

[10] S. Haridi, P. Van Roy, and G. Smolka. An overview of the design of distributed Oz. In *PASCO'97*, july 1997. available electronically at http://www.info.ucl.ac.be/people/PVR/distribution.html.

[11] M. Henz, G. Smolka, and J. Würtz. Oz–a programming language for multiagent systems. In *proc. of 13th Int. Joint Conference on Artificial Intelligence*, pages 404–409. Morgan Kaufmann Publishers, 1993.

[12] K. Hirata, R. Yamamoto, A. Imai, H. Kaway, K. Hirano, T. Takagi, and K. Taki. Parallel and distributed implementations of concurrent logic programming language KL1. In *proc. of the Int. Conf. on Fifth Generation Computer Systems*, pages 436–459, 1992.

[13] C.A.R. Hoare. Communicating sequential processes. *Communication of the ACM*, 21(8):666–677, August 1978.

[14] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *proc. of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[15] C. Laneve and U. Montanari. Mobility in the CC-paradigm. In *proc. of Mathematical Foundations of Computer Science*, LNCS vol. 629, pages 336–345. Springer-Verlag, 1992.

[16] R. Milner. Functions as processes. *Mathematical Structures in Computer Science*, 2(2):119–141, June 1992.

[17] R. Milner, J.G. Parrow, and D.J. Walker. A calculus of mobile processes, parts I and II. Technical Report ECS-LFCS-89-85 and -86, LFCS, Edinburgh University, 1989.

[18] Joachim Niehren and Martin Müller. Constraints for Free in Concurrent Computation. In *Asian Computing Science Conference*, LNCS vol. 1023, pages 171–186. Springer-Verlag, 1995.

[19] C. Palamidessi. Constraint programming for concurrent and distributed computing. In *Journées Francophones de Programmation Logique et programmation par Contraintes*, pages 11–20. Hermès Editions (Paris), May 1997. Invited talk.

[20] L. Pereira and R. Nasr. Delta-prolog: A distributed logic programming language. In *proc. of International Conference on Fifth Generation Computer systems*, pages 283–291. ICOT, 1984.

[21] Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict. Available electronically, 1996.

[22] J.-H. Réty. *Langages Concurrents avec Contraintes : Communication par Messages et Distribution.* PhD thesis, Université d'Orléans, January 1997. In French.

[23] V.A. Saraswat. *Concurrent Constraint Programming.* Phd thesis, Carnegie-Mellon University, 1989.

[24] V.A. Saraswat. The category of constraint systems is cartesian closed. In *proc. 7th IEEE Symp. on Logic in Computer Science*, pages 341–345, 1992.

[25] V.A. Saraswat, K. Kahn, and J. Levy. Janus: a step towards distributed constraint programming. In *proc. of the North American Conference on Logic Programming*, 1990.

[26] V.A. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *proc. eighteen annual ACM symposium on Principles of Programming Languages*, pages 333–353, 1991.

[27] Gert Smolka. A foundation for higher-order concurrent constraint programming. In *1st International Conference on Constraints in Computational Logics*, LNCS vol. 845, pages 50–72. Springer-Verlag, 1994.

[28] Gert Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, LNCS vol. 1000, pages 324–343. Springer-Verlag, 1995.

[29] F. Valencia, J.F. Diaz, and C. Rueda. The π^+-calculus. Technical Report AV-97-02, grupo Avispa Universidad Javeriana, Universidad del Valle, Cali, 1997. available electronically: ftp://sofia.univalle.edu.co/pub/Avispa/Pi+/.

[30] P. Van Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, 1997.

A. Encoding the ρ-calculus

A.1. The ρ-calculus

The ρ-calculus ([18]) is an extension of the γ-calculus ([27]) to constraints (The γ-calculus is restricted to equality constraints).
Let us consider a constraint system $\mathcal{C}$.

Expressions are defined by the following abstract syntax:

$Expression$ ::=	$x : \vec{y}/E$	abstraction
	$\vert x\vec{y}$	application
	$\vert E \parallel F$	parallel composition
	$\vert \exists_x E$	restriction
	$\vert c$	tell
	$\vert if\ c\ then\ E\ else\ F\ fi$	conditional
	$\vert x : y$	cell
	$\vert \top$	null expression

Let Exp be the set of expressions defined by this syntax.
The semantics of the ρ-calculus is defined as follows:

The *structural congruence* $\equiv_\rho$ is the least congruence on expressions satisfying the following laws:

- $\langle Exp, \parallel, \top \rangle$ is a commutative monoid.
- $\exists_x E \equiv_\rho \exists_y E[y/x]$ if y not free in E
  where $A[y/x]$ denotes the renaming of x by y within A.
  $\exists_x E \parallel F \equiv_\rho \exists_x (E \parallel F)$ if x not free in F
  $\exists_x \exists_y A \equiv_\rho \exists_y \exists_x A$
- $c \equiv_\rho d$ if $c \sim d$
  $c \parallel d \equiv_\rho c \wedge d$

Reduction in the ρ-calculus is defined by the rules (parallel), (exists) and (equiv) – with the congruence relation above – as in MCC (see Section 4) and the following rules:

$$\frac{c \vdash x = x' \qquad \vec{z} \cap bv(E) = \emptyset}{c \parallel x : \vec{y}/E \parallel x'\vec{z} \rightarrow_\rho c \parallel x : \vec{y}/E \parallel E[\vec{z}/\vec{y}]} \quad \text{(Application)}$$

$$\frac{c \vdash x = x'}{c \parallel x : y \parallel x'uv \rightarrow_\rho c \parallel x : u \parallel v = y} \quad \text{(Cell)}$$

$$\frac{c \vdash d}{c \parallel if\ d\ then\ E\ else\ F\ fi \rightarrow_\rho c \parallel E} \quad \text{(Then)}$$

$$\frac{c \vdash \neg d}{c \parallel if\ d\ then\ E\ else\ F\ fi \rightarrow_\rho c \parallel F} \quad \text{(Else)}$$

This presentation of the ρ-calculus follows [18] except that, in order to make the comparison with MCC simpler, we define the composition of constraints to be an axiom rather than a reduction rule.

A.2. The encoding

The ρ-calculus is encoded into MCC as follows:
$[\![x : \vec{y}/E]\!] = p(x)$,
where $p(z) :: \exists_{\vec{y}} receive(z, \vec{y}).([\![E]\!] \parallel p(z))$ is a definition in the program.

$[\![x\vec{y}]\!] = send(x, \vec{y})$
$[\![E \parallel F]\!] = [\![E]\!] \parallel [\![F]\!]$
$[\![c]\!] = c$
$[\![if\ c\ then\ E\ else\ F\ fi]\!] = ask(c).[\![E]\!] \parallel ask(\neg c).[\![F]\!]$
$[\![x : y]\!] = cell(x, y)$

The procedure $cell$ being defined as follows:
$cell(z, t) :: \exists_{z'} receive(z, (\vec{z', t})).cell(z, z')$

$[\![\top]\!] = true$

To show the correctness of this encoding, we first extend the MCC semantics with some "cleaning rules". In order to make the comparison simpler and not to bother with irrelevant syntactic points, we have to suppress disentailed ask agents. Moreover, renaming is handled in a direct way in the ρ-calculus while the MCC model makes use of diagonal elements like in the original formulation of the CC framework (cf. [26]). Operationally, this clearly comes down to the same. Let us consider the following rules whose only aim is to simplify the comparison.

$$\frac{d \vdash \neg c}{d \parallel ask(c).A \rightarrow_{MCC'} d} \quad \text{(clean)}$$

$$\frac{}{\exists_{\vec{x}}(\vec{x} = \vec{y} \parallel A) \rightarrow_{MCC'} A[\vec{y}/\vec{x}]} \quad \text{(renaming)}$$

From now on, we note $\rightarrow_{MCC'}$ the transition relation defined by the MCC rules plus the rules (clean) and (renaming).

Proposition A.1. *Let E be a ρ-expression.*
If $E \rightarrow_\rho F$ then
$[\![E]\!] \rightarrow^*_{MCC'} [\![F]\!]$.

Proof:
The proof is by structural induction on the reduction rules of the ρ-calculus.
Rules (parallel), (exists) and (equiv) are the same and the structural congruence of the ρ-calculus is included in the one of MCC$'$.
Clearly, the $if\ then\ else$ construction is equivalent to its encoding.
We focus the proof on the rules dealing with abstractions and cells.

- Application:
  Consider the following transition in the ρ-calculus.
  $c \parallel x : \vec{y}/E \parallel x'\vec{z} \rightarrow_\rho c \parallel x : \vec{y}/E \parallel E[\vec{z}/\vec{y}]$
  with $c \vdash x = x'$ and $\vec{z} \cap bv(E) = \emptyset$
  We obtain with the encoding:
  $[\![c \parallel x : \vec{y}/E \parallel x'\vec{z}]\!] = c \parallel p(x) \parallel send(x', \vec{z})$
  with $p(t) :: \exists_{\vec{y}} receive(t, \vec{y}).([\![E]\!] \parallel p(t))$
  Then, we have the following transitions in the MCC$'$ model:
  $c \parallel p(x) \parallel send(x', \vec{z}) \rightarrow_{MCC'}$

$c \parallel \exists_t (x = t \parallel \exists_{\vec{y}} receive(t, \vec{y}).(\llbracket E \rrbracket \parallel p(t))) \parallel send(x', \vec{z}) \rightarrow_{MCC'}$
$c \parallel \exists_{\vec{y}} (receive(x, \vec{y}).(\llbracket E \rrbracket \parallel p(x)) \parallel send(x, \vec{z})) \rightarrow_{MCC'}$
$c \parallel \exists_{\vec{y}} (\vec{z} = \vec{y} \parallel \llbracket E \rrbracket) \parallel p(x) \rightarrow_{MCC'}$
$c \parallel p(x) \parallel \llbracket E \rrbracket[\vec{z}/\vec{y}]$

- Cell:
  Consider the following transition in the ρ-calculus.
  $c \parallel x : y \parallel x'uv \rightarrow_\rho c \parallel x : u \parallel v = y$
  with $c \vdash x = x'$
  We obtain with the encoding:
  $\llbracket c \parallel x : y \parallel x'uv \rrbracket = c \parallel cell(x, y) \parallel send(x', (\vec{u, v}))$
  Then, we have the following transitions in the MCC$'$ model:
  $c \parallel cell(x, y) \parallel send(x', (\vec{u, v})) \rightarrow_{MCC'}$
  $c \parallel \exists_{z,t} (z = x \wedge t = y \parallel \exists_{z'} receive(z, (\vec{z', t})).cell(z, z')) \parallel send(x', (\vec{u, v})) \rightarrow_{MCC'}$
  $c \parallel \exists_{z'} receive(x, (\vec{z', y})).cell(x, z') \parallel send(x', (\vec{u, v})) \rightarrow_{MCC'}$
  $c \parallel \exists_{z'} (z' = u \wedge y = v \parallel cell(x, z')) \rightarrow_{MCC'}$
  $c \parallel v = y \parallel cell(x, u)$

$\square$