A Distributed and Probabilistic Concurrent Constraint Programming Language

Luca Bortolussi[1] and Herbert Wiklicky[2]

[1] Dep. of Maths and Computer Science, University of Udine, Udine, Italy.
bortolussi@dimi.uniud.it
[2] Dep. of Computing, Imperial College, London, UK
herbert@doc.ic.ac.uk

Abstract. We present a version of the CCP paradigm, which is both distributed and probabilistic. We consider networks with a fixed number of nodes, each of them possessing a local and independent constraint store. While locally the computations evolve asynchronously, following the usual rules of (probabilistic) CCP, the communications among different nodes are synchronous. There are channels, and through them different objects can be exchanged: constraints, agents and channel themselves. In addition, all this activities are embedded in a probabilistic scheme based on a discrete model of time, both locally and globally. Finally we enhance the language with the capability of performing an automatic remote synchronization of variables belonging to different constraint stores.

1 Introduction

In this paper we present a probabilistic and distributed version of Concurrent Constraint Programming (CCP). This version integrates CCP [17] in a network framework which resembles KLAIM [3], but which has also some features of π-calculus [14]. In addition, the language is provided with a probabilistic semantic.

Concurrent Constraint Programming was introduced in [17], and it is based on a computational paradigm founded on constraints. In particular, the usual Von Neumann's store is replaced by a store that contains constraints on the variables into play. Computations evolve monotonically: constraints can only be added in the store, but never removed. The idea behind this approach is to attach to variables not a single value, but rather an interval of possible values, which is refined as long as new information, in the form of constraints, is available. An important feature about this language is that it allows a concurrent reasoning about problems which naturally involve constraints, like optimization problems. In a CCP program, different agents can run in parallel, and the communication between them is performed through shared variables in the constraint store. In particular, agents can either tell a constraint in the constraint store, or they can ask if a particular relation is satisfied by the current configuration of the system. Constraint system are recalled in Section 3.

In this work we extend CCP in two directions: following [5], we provide it with a probabilistic semantic and we introduce the possibility of distributing programs in a network of computing machines. There are several reasons for having at disposal a probabilistic and distributed version of CCP. First of all, distributed randomized algorithms

are getting more and more popular. In fact, several languages have been proposed to describe them, like [11] and [15]. In particular, a lot of efforts are put in developing distributed metaheuristics for combinatorial optimization algorithms (cf. [16] and [13]), which involve an objective function to optimize and constraints on the variables into play. These features are naturally described in a constraint-based framework, like the one offered by CCP, while in other languages their description could be cumbersome. Therefore, extending CCP with both probabilistic and distributed features makes possible a fast prototyping of these algorithms, and also permits to perform some reasoning on them.

The extension of CCP in a distributed setting presents some obstacles. The main problem is essentially related to the way communication occurs in CCP. In fact, processes interact by posting constraints on global shared variables, and therefore communication modifies globally the system. But this form of communication is very unsuitable in a distributed environment: here we need a clear distinction between the evolution of the computation at the local level (the nodes of the network), and the global interactions of the single nodes. Hence there's the necessity of providing CCP with new primitives taking into account this distinction between the global level and the local one. Approaches in this direction can be found in [9, 8, 1, 12, 20], and we discuss them in more detail in Section 2.

Our main idea is to cast CCP into a KLAIM-like description of the network (cf. [3]). The KLAIM language is a distributed version of the LINDA paradigm, with the peculiar characteristic that it presents a clear distinction between the local level and the network level. The basic distributed components are nodes, which are then combined together to form a network. The topology of the network is encoded in the *environment* functions, which assign to each locality variable the address of a particular node. In the language presented, we define the basic distributed entity as a node, and then we compose nodes together to form a network. In each node the computation evolves (locally) according to the CCP paradigm: we have several agents which interact with ask and tell primitives. Moreover, each node has its own constraint store. While composing together nodes, we have to model properly the "union" of the local constraint store, and we do this via a direct product construction (cf. Section 3). This means that constraint stores located at different nodes are completely independent, hence the local actions performed in a node cannot interfere with the computations going on in other localities. The choice of the direct product, in particular, is suitable for the application of probabilistic abstract interpretation techniques [6] to perform a statical analysis of programs, like in [7]. The main problem we have to face, however, relates to the flow of information between different nodes. In fact, in CCP information can be exchanged only using global shared variables. But all our variables are local, and CCP offers no solution to transfer information between local and independent variables. In fact, the first, naive, idea of letting an agent tell constraints to other constraint stores is not going to work, due to variable name clashing. In addition, even if the name conflicts can be solved in some way, an agent will still be unable to match pieces of information located in different parts of the network.

Therefore, to let nodes exchange information, we must define new primitives. Following the approach in [9] and [1], we define a synchronous message-based commu-

nication which resembles the π-calculus paradigm: there are channels and messages flowing along them.

We pinpoint that the mechanisms presented up to now create a neat distinction between the local level, consisting of nodes and of computations localized in them, and the global level, related to the network, its topology, its dynamics. This distinction allows us to model separately the evolution of the single nodes and of the network, thus simplifying the task of adding probabilities to the system. In detail, we decide to use a discrete time both for local and global dynamics (which means probabilistic scheduling policies), but we impose an asynchronous local communication versus a synchronous global one. The price of this distinction is an increased complexity both in the description of the configurations and in the transition rules.

In this language there are three basic objects that can be communicated along the network: constraints, channel names and agents. To communicate constraints, however, we must tackle the problem of variable name conflicts between different nodes. This is achieved by "abstracting" a constraint with respect to a subset of its free variables (hiding all the others), thus creating a kind of template, whose (template) variables can be replaced by suitable ones specified by the receiver. We give more details in Section 4.1. Channels can be exchanged in a way that resembles closely the π-calculus. Their communication allows to reconfigure the topology of the network, by changing the communication links. In addition, new channel names can be created dynamically, providing a form of private communication. The third object that can be communicated are agents. As for the case of constraints, we need to abstract them in order to avoid variables clashes. Therefore, (some of) the free variables of a transmitted agent are replaced by suitable ones belonging to the receiving node. In addition, an agent that migrates can carry with itself some information about (part of) its free variables.

Another feature that may be needed in a distributed CCP is the possibility of synchronizing variables belonging to different constraint stores. This would allow for an automatic information flow between different nodes, and would greatly simplify the writing of programs. This linking of variables is realized by adding information about linked variables directly in the configuration of the system, at the network level. We refer to Section 6 for further details.

We want to remark the important fact that the language not only is distributed, but it also evolves following probabilistic rules, meaning that every form of non-determinism is solved probabilistically. As already said, this feature allows for an easy description of distributed randomized algorithms and especially, due to the constraint-based framework, parallel stochastic optimization algorithms. In addition, the different primitives for local and global communication make possible to model such algorithms in different ways, permitting a rapid comparison between them.

The paper is organized as follows: in Section 2 we present some related work. In Section 3 we recall the concept of constraint system, while in Section 4 we introduce the syntax of the language. In Sections 4.1 and 4.2 we spend some words on constraint and agent abstractions, and we discuss some issues related with communication channels. The Structural Operational Semantics is introduced and discussed in detail in Section 5, while Section 6 deal with the linking mechanisms. Finally in Section 7 we draw the final conclusions.

2 Related Work

In this Section we discuss briefly some related work. In particular, CCP has been extended both in the probabilistic setting and in the distributed one. As far as we know, there is no extension covering both aspects together.

Probabilistic CCP (pCCP) has been introduced in [5] to declaratively model randomized algorithms. It is now a quite well established language, with its own operational [5] and denotational [4] semantic. There is also a probabilistic version of abstract interpretation [6] that can be used to perform some static analysis of pCCP programs.

On the other hand, there has been a lot of work also for devising a distributed version of CCP. This is not an easy task, due to the fact that communication in CCP proceeds by posting constraints acting on global variables. To tackle this problem, different forms of communication between agents have been added. In [9], Réty proposes a π-calculus approach to model the communication between agents located in different nodes. In particular, new instructions for exchanging messages are introduced, even if there is still a single, global, constraint store, and the independence among variables is imposed as a further condition in the definition of the agents. In addition, the concepts of constraint abstraction and linking are defined. In [1], De Boer et alt. present a similar approach, with global communication performed via message passing and local communication performed in CCP style. They also introduce a concept of independent local stores, even if the global configuration of a network is taken to be the least upper bound of the local constraint stores (cf. Section 3). In [12] a different type of synchronous communication is discussed, based on a redefinition of the semantics of ask and tell primitives: a constraint is told only if there in an agent asking for it. Also this version makes use of a single global constraint store.

Our approach towards distribution shares some features with the previous approaches, but we impose a strict independence between local constraint stores, such that the interaction between localized variables must always be made explicit. This division creates a clear distinction between the local and the global level, at the price of introducing more instructions and a more complex representation of the configuration of the system. This increased complexity, however, allows a clear modelling the probabilistic evolution of the system, by distinguishing between local and global time. In addition, this separation can be exploited in the context of probabilistic abstract interpretation [6], in particular in the static analysis of distributed networks, as done by Di Pierro et alt. in [7]. Finally, having a concept of computational site at disposal, we can define easily mobility of agents.

Regarding the migration of CCP agents, work has been done by Palamidessi et alt. in [8], where CCP is enriched by a hierarchical network structure and agents can move from one node to another one (bringing with them their subagents). Our network structure, instead, is simpler (there is no hierarchical organization), as we focused mainly on the communication issues.

3 Background and Preliminaries

Computations in CCP are performed through a monotonic update of the so called constraint store, which is usually modeled as a constraint system. We follow here a well

established approach (cf. [18] or [2]), which represents a constraint system as a complete algebraic lattice, where the ordering $\sqsubseteq$ is given by the inverse of the entailment relation $\vdash$. Usually, such a constraint system is derived from a first-order language together with an interpretation, where constraints are formulas of the language, and a constraint c entails a constraint d, $c \vdash d$, if every valutation satisfying c satisfies also d. In this case we write $d \sqsubseteq c$. Clearly, in every real implementation the predicate $\vdash$ must be decidable. In addition, to model hiding and parameter passing, the previous lattice is enriched with a cylindric algebraic structure (cf. [10]), i.e. with cylindrification operators and diagonal elements.

Formally, a constraint system $\mathcal{C} = (Con, Con_0, Var, \sqsubseteq, \sqcup, true, false, \exists_x, d_{xy})$ is a complete algebraic lattice where Con is the set of constraints, ordered by $\sqsubseteq$, Con_0 is the set of finite elements, $\sqcup$ is the least upper bound (lub) operation, Var is the set of variables, $true$ and $false$ are respectively the bottom and the top element, $\{\exists_x \mid x \in Var\}$ are the cylindrification operators and $\{d_{xy} \mid x, y \in Var\}$ are the diagonal elements.

In our distributed version of CCP, we assign an independent constraint store to each node composing the network: if we have m nodes in our system, then we have m constraint systems $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Therefore, we can model a global configuration of the network by the direct product $\mathcal{C}_1 \times \ldots \times \mathcal{C}_m$.This product, where all the operations are performed elementwise, is still a cylindric algebra. The proof of this fact is straightforward, and follows from more general results in [10]. A fundamental property of this construction is that the variables are independent between different constraint stores.

4 Syntax

The distributed network is basically composed by a set of nodes, linked between them by communication channels. Every node contains CCP-based processes, while the CCP syntax is enriched with communication mechanisms at the network level.

We need several different syntactic objects:

1. *physical localities*, or *sites*, which are the addresses of the nodes of the network, and are taken from a set S. Every node must have a distinct address, and we can assume S to be countable and indexed by the natural numbers (i.e. $S = \{s_1, \ldots, s_n, \ldots\}$). We will impose, however, a restriction on the actual number of nodes that can be used in a program, i.e. we will assume that they are finite. Often we refer to a node with address s_j simply as node j;
2. *channel names* and *channel variables*. Channels specify the topology of the network, and they are communication routes between nodes. Channel names are taken from the set $\mathcal{L}$, and are indicated by Greek letters α, β, and so on. Channel variables are taken from the set $Var_{\mathcal{L}}$, and are indicated by underlined Greek letters, like $\underline{\alpha}$, $\underline{\beta}$;
3. *environments*. They are functions which associate to each channel a probability distributions over S. Formally, they are indicated by ϱ, and $\varrho : \mathcal{L} \to \mathcal{D}(S)$, where $\mathcal{D}(S)$ is the set of probability distributions over S. Therefore, environments specify how transmissions are performed through shared channels. We comment more on this fact in Section 5.

$Program = Decl.A$
$D = \varepsilon \mid Decl.Decl \mid p(\overrightarrow{x}) : -A$
$A = \mathbf{0} \mid \mathrm{tell(c).A} \mid \exists_{\mathrm{x}}\mathrm{A} \mid \mathrm{p}(\overrightarrow{x}) \mid \sum_{\mathrm{i=1}}^{\mathrm{k}} \mathrm{q_i} : \mathrm{ask(c_i).A_i} \mid \mid_{\mathrm{i=1}}^{\mathrm{k}} \mathrm{q_i} : \mathrm{A_i} \mid$
$\mathrm{out_c}(\lambda\overrightarrow{x}c)@\alpha.A \mid \mathrm{in_c}(\overrightarrow{y})@\alpha.A \mid \mathrm{out_{loc}}(\beta)@\alpha.A \mid \mathrm{in_{loc}}(\underline{\beta})@\alpha.A \mid \mathrm{new}(\underline{\beta}).A \mid$
$\mathrm{out_A}(\lambda\overrightarrow{x}A, \overrightarrow{x_0})@\alpha.A \mid \mathrm{in_A}(\overrightarrow{y})@\alpha.A \qquad (\overrightarrow{x_0} \subseteq \overrightarrow{x} \subseteq fv(A))$

Table 1. Syntax of pDCCP

A node in the network is defined as:

$$n \ ::= \ s ::_{\varrho}^{p} P,$$

where P is the current agent that runs on this node, s is the address of the node, taken from S, ϱ is the environment associated to node n and p is a probability associated to n. This is the probability with which node n is chosen for execution by the global scheduler.

A network N is defined as a parallel composition of a finite number of nodes:

$$N \ ::= \|_{i=1}^{m} \ n_i.$$

To have a probability distribution over the network, the probabilities associated to each node should sum up to one. However, we omit this request, as we are going to deal with normalization structurally, i.e. via a congruence relation (cf. next Section). Moreover, the scheduling probabilities are fixed, and cannot change during the execution of the program.

We note that posing an upper bound on the number of nodes which compose a network is not a real limitation, as we can choose this number very high, with possibly many inactive nodes at the beginning of the computation which can be populated by processes in the future.

In Table 1 we describe the syntax of an agent. It is composed by a declaration of procedures and by an initial goal.

The declaration section is standard, and we ask that $fv(A) \subseteq \overrightarrow{x}$, for all declarations of the form $p(\overrightarrow{x}) : -A$. The first 6 instructions of agent syntax are also standard and they are taken from the probabilistic version of CCP (cf. [5]). The remarkable facts are the probabilistic choice operator (thus non-determinism is replaced by probabilistic choice), and the probabilistic version of the parallel operator. The probability distribution associated to it introduces priorities in the local scheduler, and thus may bias the interleaving of processes. In both these instructions, q_i represents the probability associated either to the branch i or to the parallel agent i.

The out and in instructions are the primitives for communication. They appear in three different forms, as there are three different objects that we want to send over channels, i.e. constraint (abstractions), channel names and agent (abstractions). Constraint and agent abstractions are indicated respectively with $\lambda\overrightarrow{x}c$ and $\lambda\overrightarrow{x}A$; cf. below for an

explanation. Finally, the new instruction creates new channel names, in order to dynamically reconfigure the topology of the network.

All global communication instructions are of the form $\mathrm{out}(\cdot)@\alpha$ or $\mathrm{in}(\cdot)@\alpha$, where α can be both a channel name or a channel variable. Clearly, actual communications can be performed only on real channels, not on channel variables. Therefore, we must impose that all channel variables are bounded, i.e. they appear in the scope of an $\mathrm{in}_{\mathrm{loc}}$ or a new instruction. We write $A[\beta/\underline{\beta}]$ to indicate that the channel variable $\underline{\beta}$ has been replaced by the channel name β.

4.1 Constraint and Agent Abstractions

The independence of variables among different constraint stores poses some problems in the communications performed between nodes. In particular, if a process at node i wants to communicate a constraint c, then we have the problem that variables in c are related to the constraint store $\mathcal{C}_i$. If c is sent to another node, say j, then we must specify which variables of $\mathcal{C}_j$ it refers to. This choice can be performed by the receiving process at node j. However, to perform this renaming, we have to abstract the variables present in c, in order to create a kind of constraint template, where the new variables can be put. We follow the approach introduced in [9], making use of constraint abstractions.

A *constraint abstraction* is a couple $(\overrightarrow{x}, c)$, where $\overrightarrow{x} \subseteq fv(c)$, and will be denoted by $\lambda\overrightarrow{x}c$. The variables $\overrightarrow{x}$ are the template variables, and all other free variables of c are hidden (that is to say, c is projected over $\overrightarrow{x}$). We can define the projection operator $\mathit{\Pi}_{\overrightarrow{x}}c = \exists_{fv(c)\setminus\overrightarrow{x}}c$, i.e. we hide all the variables except the ones in $\overrightarrow{x}$. Then, if $\overrightarrow{y}$ is a vector of variables of the same length of $\overrightarrow{x}$, the application of $\overrightarrow{y}$ to $\lambda\overrightarrow{x}c$ is the constraint $(\mathit{\Pi}_{\overrightarrow{x}}c)[\overrightarrow{y}/\overrightarrow{x}]$.

The independence among variables creates an analogous problem if we want to move entire agents along the network. Therefore, we need to define also an abstract version of agents. Given an agent A and a subset $\overrightarrow{x}$ of its free variables $fv(A)$, an agent abstraction is a couple $(A, \overrightarrow{x})$, denoted by $\lambda\overrightarrow{x}A$. The variables $\overrightarrow{x}$ are "templates" that must be substituted by variables $\overrightarrow{y}$ belonging to the constraint store of the receiving node. All the free variables of A different from $\overrightarrow{x}$, instead, must be hidden, and consequently we define a projection operator also for agents: $\mathit{\Pi}_{\overrightarrow{x}}(A) = \exists_{fv(A)\setminus\overrightarrow{x}}A$.

4.2 Communication Channels

The communication we have at the global level is synchronous and message-based. The exchange of information between nodes in performed through communication channels shared among them. These channels, however, cannot be represented by the usual variables (as in [9]), because constraint stores are totally independent. Therefore, we need to define a new syntactical category of objects, with their own properties (as in [1]). Channels will be denoted by names, identified by Greek letters α, β, and so on, all belonging to the set $\mathcal{L}$. Each channel name identifies a unique channel, i.e. an idealized mean or link which represents communication bridges between different nodes of the network. To allow a dynamical reconfiguration of the topology of the network, we have also channel variables (indicated by $\underline{\alpha}, \underline{\beta} \in Var_{\mathcal{L}}$), which can be substituted, during the

execution of the program, by a channel name. Because of the fact that communications can only happen in channels, we must ask that each channel variable present in an agent is bounded, i.e. it appears under the scope of an $\mathrm{in}_{\mathrm{loc}}$ or new instruction. We suppose that the set of channel's names is infinite.

Channels introduce a form of non-determinism in the network communications. In fact, a channel α can be shared by several nodes, and at some time of the execution of the program, we can have a process that wants to send a message along α and a bunch of agents which may be able to receive it. This non-determinism is solved probabilistically by the environment ϱ, which is a function assigning to each channel name a probability distribution over nodes (or better, over node's addresses). Note, however, that the requirement that each node has an a-priori defined probability distribution assigned to each channel name is too strong. To relax this condition, we can proceed as follows: we define a global environment ϱ_N, which assigns to every channel the uniform probability over nodes. Then we allow the environments ϱ'_i assigned to nodes to be partial functions from $\mathcal{L}$ to $\mathcal{D}(S)$. The "real" environment ϱ_i associated to a node is $\varrho_i = \varrho_N \bullet \varrho'_i$, where $\bullet$ means that ϱ'_i is extended by ϱ_N, wherever it is undefined. In particular, we ask that the environment of every node is undefined for channels created by the instruction new.

We observe that the fact having probability distributions attached to channels let us define "unidirectional" communications, i.e. channels where the communication can happen in just one sense. In fact, consider a channel α shared between nodes i and j, such that $\varrho_i(\alpha)(s_j) > 0$ and $\varrho_j(\alpha)(s_i) = 0$: node j can only receive messages from node i along α, but can never send something to i ($\varrho(\alpha)(s) > 0$ is a condition in communication rules, cf. Section 5).

5 Operational Semantics

The operational semantic of the language is given by a congruence relation between agents, and by a transition relation between configurations, labeled with probabilities. As for the syntax, both these objects have two versions, one local and one global.

A configuration of the system at one particular node will be an element of $\mathcal{P} \times \mathcal{C}$, where $\mathcal{P}$ is the space of processes and $\mathcal{C}$ is the constraint store associated to the node. Therefore, it is a couple $\langle A, c\rangle$, with $A \in \mathcal{P}$ and $c \in \mathcal{C}$. Consequently, a configuration of the network will be of the form $s_1 ::^{p_1}_{\varrho_1} \langle A_1, c_1\rangle \parallel \ldots \parallel s_m ::^{p_m}_{\varrho_m} \langle A_m, c_m\rangle$, which can be written also ${}^{s_1}_{p_1}\langle A_1, c_1\rangle_{\varrho_1} \parallel \ldots \parallel {}^{s_m}_{p_m}\langle A_m, c_m\rangle_{\varrho_m}$, or equivalently as $\langle A_1, \ldots, A_m, c_1, \ldots, c_m\rangle$. That is to say, a global configuration will be a point of $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, where $\mathcal{P}_i$ and $\mathcal{C}_i$ indicate respectively the space of processes and the constraint store of node i.

The congruence relation is twofold. One relation regards agents, and it splits them into an equivalence class. The other relation, instead, regards network configurations. They are both defined in Table 2.

Rules (**CR1**) and (**CR2**) simply state that the order of a sum and of a parallel composition is immaterial. In fact, we ask that two agents are congruent if one is obtained by the other simply permuting the composing terms. (**CR3**) and (**CR4**) deal with normalization issues. In fact, a basic requirement of probabilistic declarative languages is that the values associated to the choice or parallel operator are positive and add up to

$(CR1)$	$\sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i \ \equiv \ \sum_{i=1}^{k} q_{\pi(i)} : \mathrm{ask}(c_{\pi(i)}).A_{\pi(i)},$	for all permutations π
$(CR2)$	$\|_{i=1}^{k} q_i : A_i \ \equiv \|_{i=1}^{k} q_{\pi(i)} : A_{\pi(i)},$	for all permutations π
$(CR3)$	$\sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i \ \equiv \ \sum_{i=1}^{k} \widetilde{q}_i : \mathrm{ask}(c_i).A_i$	where $\widetilde{q_j} = \frac{q_j}{\sum_{i=1}^{k} q_i}$
$(CR4)$	$\|_{i=1}^{k} q_i : A_i \ \equiv \|_{i=1}^{k} \widetilde{q}_i : A_i$	where $\widetilde{q_j} = \frac{q_j}{\sum_{i=1}^{k} q_i}$
$(CR5)$	$q_1 : \mathbf{0} \mid q_2 : A_2 \mid \ldots \mid q_k : A_k \ \equiv \ q_2 : A_2 \mid \ldots \mid q_k : A_k,$	
$(CR6)$	$\exists_x \exists_y A \ \equiv \ \exists_y \exists_x A$	
$(CR7)$	$\exists_x A \ \equiv \ \exists_y A[y/x]$	if y is not free in A
$(CR8)$	$\exists_x A \ \equiv \ A$	if x is not free in A
$(CR9)$	$\|_{i\in S} \, {}^{s_i}_{p_i} \langle A_i, c_i \rangle_{\varrho_i} \ \equiv \|_{i\in S} \, {}^{s_{\pi(i)}}_{p_{\pi(i)}} \langle A_{\pi(i)}, c_{\pi(i)} \rangle_{\varrho_{\pi(i)}}$	$\forall \pi$ permutation of S
$(CR10)$	$\|_{i\in S} \, {}^{s_i}_{p_i} \langle A_i, c_i \rangle_{\varrho_i} \ \equiv \|_{i\in S} \, {}^{s_i}_{\widetilde{p_i}} \langle A_i, c_i \rangle_{\varrho_i}$	where $\widetilde{p_j} = \frac{p_j}{\sum_{k\in S} p_k}$

Table 2. Congruence relation

one, i.e. they are a probability distribution. This implies, for instance, that, whenever an agent is added or removed from a parallel composition, we have to go through a process of renormalization of the associated probabilities. Instead of doing this explicitly, we define the normalization in the congruence relation: two sums or two parallel compositions, different just in the numerical weights, are congruent if they are equal after a normalization of the coefficients. In practice, the congruence class of every choice or parallel agent, with associated vector of weights $\overrightarrow{q}$, contains all processes which coefficients are positive multiples of $\overrightarrow{q}$. We always choose as representative of this class the unique agent for which $\overrightarrow{q}$ is a p.d.

(**CR5**), instead, states that adding a null agent to a parallel composition does not modify the program. Note that adding or removing a null agents modifies the vector of coefficients. However, we do not need to renormalize it, as this is done automatically by rule (**CR4**). Rules (**CR6**), (**CR7**) and (**CR8**) simply state the basic properties of the hiding operator. Finally, rule (**CR9**) affirms the the order of parallel composition of nodes in a network is irrelevant, while rule (**CR10**) implements the automatic normalization trick also at the network level. We observe also that the algebraic laws associated to a constraint system (cf. [2]) induce an implicit congruence relation over the constraint store at each node.

The main ingredient of the SOS is the labeled transition relation (LTR). There are two of such relations, one for the local evolutions and one for the network's one. Both are labeled by a real number in $[0, 1]$, representing the *probability associated to each transition*. Local transitions are stated in Table 3, while in Table 4 the global transition system is described. Formally, the local transition relation is indicated by $\longrightarrow$ and it is a subset of $\mathcal{P} \times \mathcal{C} \times [0, 1] \times \mathcal{P} \times \mathcal{C}$, while the global transition relation is represented by $\Longrightarrow$, and it is a subset of $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m \times [0, 1] \times \mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$.

Before entering into a detailed description of each transition rule, we must explain a general trick in the notation. Suppose at some point of the computation the current

$$
\begin{array}{lll}
(LR1) & \langle \mathrm{tell}(c).A, d\rangle \longrightarrow_1 \langle A, d \sqcup c\rangle & \\
(LR2) & \left\langle \sum_{i=1}^{k} q_i : \mathrm{ask}(c_i).A_i, d \right\rangle \longrightarrow_{\tilde{q}_j} \langle A_j, d\rangle & \text{if } d \vdash c_j \\
(LR3) & \dfrac{\langle A, d\rangle \longrightarrow_p \langle A', d'\rangle}{\left\langle q_1 : A \mid_{i=2}^{k} q_i : B_i, d \right\rangle \longrightarrow_{p\cdot\tilde{q}_1} \left\langle q_1 : A' \mid_{i=2}^{k} q_i : B_i, d' \right\rangle} & \\
(LR4) & \dfrac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow_p \langle B, d'\rangle}{\left\langle \exists_x^d A, c\right\rangle \longrightarrow_p \left\langle \exists_x^{d'} B, c \sqcup \exists_x^{d'} \right\rangle} & \\
(LR5) & \langle p(\overrightarrow{y}), c\rangle \longrightarrow_1 \left\langle \Delta_{\overrightarrow{x}}^{\overrightarrow{y}} A, c \right\rangle & \text{if } p(\overrightarrow{x}) : -A \in Decl \\
(LR6) & \left\langle \mathrm{new}(\underline{\beta}).A, d\right\rangle \longrightarrow_1 \left\langle A[\beta_{new}/\underline{\beta}], d\right\rangle, & \text{where } \beta_{new} \text{ is fresh.}
\end{array}
$$

Table 3. Labeled transition system at the local level

agent is a probabilistic choice. Generally, not all ask guards will be entailed, so we have to renormalize the probability distribution (p.d.) over the entailed branches. Formally, if the current configuration of the node is $\left\langle \sum_{i=1,\ldots,k} q_i : ask(c_i).A_i, d \right\rangle$, we define the subset $Active \subset \{1,\ldots,k\}$ as $Active = \{i | d \vdash c_i\}$, and then, if $j \in Active$, its normalized probability is $\tilde{q}_j = q_j / \sum_{i \in Active} q_i$. Similar considerations apply also to the local and global parallel constructs. In general, we adopt the notational convention that, if q is a probability distribution associated to an instruction, then $\tilde{q}$ is the same probability distribution normalized over active components.

The transition rules that are sketched out in Table 3 resemble closely the one presented for pCCP in [5]. Rule (**LR1**) models the tell operation. This operation adds the constraint c to the store, and always succeeds with probability one. (**LR2**) deals with the probabilistic choice, and the probability associated to the transition is the normalized probability over active guards (cf. above). Rule (**LR3**) regards parallel composition of local agents. It states that, if a local transition can happen with probability p, then the same action can happen in a parallel composition, with probability $p\tilde{q}_j$, where $\tilde{q}_j$ is the scheduling probability normalized among active agents. Rules (**LR4**) and (**LR5**) are the usual rules for modelling the action of the hiding operator and the parameter passing (for a discussion of the operator $\Delta_y^x A = \exists_y^{d_{xy}} A$ see [2]). Actually, the delta operator defined here links vector of variables and not single variables, but it is a straightforward generalization: $\Delta_{\overrightarrow{x}}^{\overrightarrow{y}} = \Delta_{x_k}^{y_k} \cdots \Delta_{x_1}^{y_1}$, if $|\overrightarrow{x}| = |\overrightarrow{y}| = k$. Rule (**LR6**) is the only outsider, as it gives the semantics of the new instruction, which deals with dynamic reconfiguration of the network connections. In particular, its effect is that of creating a fresh channel name β_{new} and binding the channel variable $\underline{\beta}$ to it. Note that, according to the convention of Section 4.2, all environments associate a uniform p.d. to new channel names.

Table 4 contains the definition of the transition relation at the network level. Rule (**GR1**) links the local transition and the global ones. It says that, whenever a local action can be performed in node i with probability p, then a global transition of the network

$$(GR1)\qquad \frac{\langle A,d\rangle \longrightarrow_p \langle A',d'\rangle}{{}^{s_i}_{p_i}\langle A,d\rangle_{\varrho_i} \parallel N \implies_{p\cdot\tilde{p}_i} {}^{s_i}_{p_i}\langle A',d'\rangle_{\varrho_i} \parallel N}$$

$$(GR2)\qquad \frac{|\overrightarrow{x}| = |\overrightarrow{y}| \text{ and } \varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_c}(\lambda\overrightarrow{x}\,c)@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : \mathrm{in_c}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N \implies_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : A_j \mid A'_j, d_j \sqcup \left((\Pi_{\overrightarrow{x}} c)[\overrightarrow{y}/\overrightarrow{x}]\right)\right\rangle_{\varrho_j} \parallel N\end{array}}$$

$$(GR3)\qquad \frac{\varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_{loc}}(\beta)@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : \mathrm{in_{loc}}(\underline{\beta})@\alpha.A_j \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N \implies_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : A_j[\beta/\underline{\beta}] \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N\end{array}}$$

$$(GR4)\qquad \frac{|\overrightarrow{x}| = |\overrightarrow{y}| \text{ and } \varrho_i(\alpha)(s_j) > 0}{\begin{array}{c}{}^{s_i}_{p_i}\langle \mathrm{out_A}(\lambda\overrightarrow{x}\,A, \overrightarrow{x_0})@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : \mathrm{in_A}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N \implies_{\tilde{q}_1\tilde{p}_i\cdot\tilde{\varrho}_i(\alpha)(s_j)} \\ {}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle \tfrac{1}{2}q_1 : A_j \mid \tfrac{1}{2}q_1 : (\Pi_{\overrightarrow{x}} A)[\overrightarrow{y}/\overrightarrow{x}] \mid A'_j, d_j \sqcup \left((\Pi_{\overrightarrow{x_0}} d_i)[\overrightarrow{y_0}/\overrightarrow{x_0}]\right)\right\rangle_{\varrho_j} \parallel N \\ \text{where } \overrightarrow{y_0} \subseteq \overrightarrow{y} \text{correspond to } \overrightarrow{x_0} \subseteq \overrightarrow{x}\end{array}}$$

Table 4. Labeled transition system at the network level

can be performed with a probability p multiplied by $\tilde{p}_i$, i.e. by the probability that the global scheduler chooses node i for execution, normalized among active nodes. Clearly a node is active if and only if it can perform a transition, be it a local update or a global communication.

Rules (**GR2**) to (**GR4**) concern the global communication between nodes of the network.

Rule (**GR2**) gives the semantics of the in and out instructions devoted to send constraints abstractions among the network. This communication is synchronous, therefore we need a process at one node, i say, willing to communicate a constraint (abstraction) $\lambda\overrightarrow{x}\,c$ over a channel α, and an agent at another node, say j, ready to receive a communication along the *same* channel. In addition, the *length* of the template vector $\overrightarrow{x}$ must coincide with the length of the vector $\overrightarrow{y}$ specified by the receiver. If these conditions are satisfied, then the communication is performed and the constraint store at node j is updated by adding to it $(\Pi_{\overrightarrow{x}} c)[\overrightarrow{y}/\overrightarrow{x}]$. The probability associated to this transition is the product of the (normalized) probability of choosing node i multiplied by the normalized probability that the transmission along channel α reaches node j, and not another node with an active process waiting for a communication along α, with a matching template. This is the probability assigned to the channel by the environment ϱ_i. Observe that the premise of rule (**GR2**) asks explicitly for $\varrho_i(\alpha)(s_j) > 0$, so transmissions cannot be directed towards nodes s_k with $\varrho_i(\alpha)(s_k) = 0$. In addition, the above probabilities are also multiplied by $\tilde{q}_1$, which is the local parallel probability of the in agent, normalized

between the other locally parallel agents capable of receiving the same transmission along α.

Communication of channel names is performed synchronously by the $\mathrm{out}_{\mathrm{loc}}(\beta)@\alpha$ and $\mathrm{in}_{\mathrm{loc}}(\beta)@\alpha$ instructions, which semantics is presented in rule (**GR3**). When the transmission happens, the sent channel name is received and bound to the the channel variable specified in the $\mathrm{in}_{\mathrm{loc}}$ instruction. The probability and preconditions associated to this transition are the same as in rule (**GR2**).

The last objects that can be moved along the network are agent abstractions. In addition to sending agents, we can also transmit the current status of some of their free variables. In particular, if $\overrightarrow{x}$ is the vector of template variables of an agent abstraction $\lambda\overrightarrow{x}A$, then we can choose a subsequence $\overrightarrow{x_0} \subseteq \overrightarrow{x}$ and carry the information about $\overrightarrow{x_0}$ together with the agent. To move an agent from node i to node j, we need to specify which variables $\overrightarrow{y}$ of the constraint store of node j will be substituted to the template variables $\overrightarrow{x}$. Note that the subsequence $\overrightarrow{x_0}$ of $\overrightarrow{x}$ determines a subsequence $\overrightarrow{y_0}$ of $\overrightarrow{y}$, where the information carried about $\overrightarrow{x_0}$ will be stored. Rule (**GR4**) specifies the transition associated with agent migration. Communication can be performed between two active $\mathrm{out}_{\mathrm{A}}$ and in_{A} instructions, and is realized by adding in parallel at the receiving node the agent $(\Pi_{\overrightarrow{x}}A)[\overrightarrow{y}/\overrightarrow{x}]$. The information about $\overrightarrow{x_0}$ is added by posting to the constraint store of the receiving node the constraint $(\Pi_{\overrightarrow{x_0}}d_i)[\overrightarrow{y_0}/\overrightarrow{x_0}]$. The preconditions and the probability associated to the transition are the same as (**GR2**) and (**GR3**). While putting the agent $\lambda\overrightarrow{x}A$ in parallel with other agents, we have to assign to it the probability for the local parallel operator. This is done by splitting in two the probability of the agent that performed the in_{A} instruction. In rule (**GR4**) this is realized by giving probability 0.5 to this agent and probability 0.5 to the newly added one. Compositionality of local parallelism and the automatic renormalization induced by the congruence relation assure that we end up with a coherent probability distribution in node j.

Computational Paths and Observables A configuration of the network is a point in the space $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, and we indicate it as $(\overrightarrow{P}, \overrightarrow{c})$. The SOS defines a labeled transition relation between configurations of the network, where $(\overrightarrow{P_1}, \overrightarrow{c_1}) \Longrightarrow_p (\overrightarrow{P_2}, \overrightarrow{c_2})$ means that we can go from $(\overrightarrow{P_1}, \overrightarrow{c_1})$ to $(\overrightarrow{P_2}, \overrightarrow{c_2})$ in one step with probability p. Given a path $(\overrightarrow{P_1}, \overrightarrow{c_1}) \Longrightarrow_{p_1} (\overrightarrow{P_2}, \overrightarrow{c_2}) \Longrightarrow_{p_2} \ldots \Longrightarrow_{p_n} (\overrightarrow{P_{n+1}}, \overrightarrow{c_{n+1}})$, its probability p is the product of the probabilities of the single transitions, $p = \prod_{i=1}^{n} p_i$.

The $*$-closure of this relation is defined in the usual way: $(\overrightarrow{P_a}, \overrightarrow{c_a}) \Longrightarrow_p^* (\overrightarrow{P_b}, \overrightarrow{c_b})$ means that we can reach configuration $(\overrightarrow{P_b}, \overrightarrow{c_b})$ from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ in one or more steps. However, particular care must be put in defining the probability p associated to it. In fact, in general we can have more than one path from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ to $(\overrightarrow{P_b}, \overrightarrow{c_b})$, each one with its own probability. Therefore, the probability p will be the sum of the probabilities of all paths leading from $(\overrightarrow{P_a}, \overrightarrow{c_a})$ to $(\overrightarrow{P_b}, \overrightarrow{c_b})$.

A computation of the system is successful if it reaches a state where the vector of agents to be executed is $\overrightarrow{\mathbf{0}} = (\mathbf{0}, \ldots, \mathbf{0})$, i.e. when computations in every node have stopped correctly.

The observables we define hereafter correspond to the input / output behaviour of the system. Given the vector of declarations $\overrightarrow{D}$, the observable $\mathcal{O}_{\overrightarrow{D}}((\overrightarrow{A}, \overrightarrow{c}))$ for an

initial configuration $(\overrightarrow{A}, \overrightarrow{c})$ is a probability distribution over $\mathcal{C}_1 \times \ldots \times \mathcal{C}_m$, defined as

$$\mathcal{O}_{\overrightarrow{D}}((\overrightarrow{A}, \overrightarrow{c})) = \left\{ (\overrightarrow{d}, p) \mid (\overrightarrow{A}, \overrightarrow{c}) \Longrightarrow_p^* (\overrightarrow{\mathbf{0}}, \overrightarrow{d}) \right\}.$$

For finite computations this notion of an observable is quite straight forward and results in a probability distribution over all possible outcomes. In the case of infinite, i.e. non-terminating, computations some probabilities could be "lost" which leads to observables corresponding so-called sub-probability distributions over the finite outcomes. One could also define a more complicated measure theoretic structure on the set of all computational paths based on so-called cylindric sets, see e.g. [19], but for the sake of simplicity we will, for the time being, avoid a further investigation of these possibilities.

6 Linking variables

Each node in the network possesses its own constraint store, with an independent variable set. Therefore, the updating of information in each constraint store evolves independently, a part from the fact that constraint abstractions can be actively communicated between agents. But sometimes a more efficient and pervasive way of transmitting information may be necessary. In particular, we may want to link variables $\overrightarrow{x}$ and $\overrightarrow{y}$ belonging to different constraint stores, in such a way that whenever a variable $x_k \in \overrightarrow{x}$ is updated (its domain is restricted by the presence of a new constraint in the constraint store), then the corresponding variable $y_i \in \overrightarrow{y}$ is updated with the same information. Note that this linking of variables is unidirectional, i.e. the information flows from $\overrightarrow{x}$ to $\overrightarrow{y}$, and therefore constraints can be posted on $\overrightarrow{y}$ without affecting $\overrightarrow{x}$.

This mechanism, which is already present in [9], seems unavoidable if one wants to synchronize the information present in different constraint stores, and enhance the features of the language. However, we proceed in a different way than Réty. In fact, in [9] variables are linked by means of an auxiliary agent that is put in parallel with the existing ones. However, this approach has some drawbacks: the link agent can generate infinite spurious computations, by broadcasting the same constraint forever, and moreover there is an uncontrollable delay between the update of linking variables and the update of linked variables. While the first problem can be somewhat limited by asking for weak fairness, the second seems much more troublesome, especially if one want to reason about the effects of outdated information in computations.

To circumvent this problems, we lift the information about linking from the agent level to the configuration of the network. In detail, we define the set $\mathbb{L} = \{(\overrightarrow{x}, \overrightarrow{y}, i, j) \mid \overrightarrow{x} \subset Var_i, \overrightarrow{y} \subset Var_j, |\overrightarrow{x}| = |\overrightarrow{y}|\}$, which contains all possible couples of variables that can be linked together. In particular, if $(\overrightarrow{x}, \overrightarrow{y}, i, j) \in \mathbb{L}$, then the flow of information goes from $\overrightarrow{x}$ to $\overrightarrow{y}$. Then, at each configuration of the network $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m$ we associate a subset $\mathbf{L} \subset \mathbb{L}$. Therefore, a network configuration is now expressed by an element of the set $\mathcal{P}_1 \times \ldots \times \mathcal{P}_m \times \mathcal{C}_1 \times \ldots \times \mathcal{C}_m \times \wp(\mathbb{L})$. The set $\mathbf{L}$ contains all the couple of currently linked variables, and can change dynamically during the execution of a program, both by adding or removing elements from it.

In Table 5 we present both the syntax and the rules for dealing with the linking mechanism. The syntax of the language is extended by three instructions, which cover

$$A = \mathrm{in}_{\mathrm{link}}(\overrightarrow{x})@\alpha \mid \mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha \mid \mathrm{remove}_{\mathrm{link}}(\overrightarrow{y})$$

$$(GR5)\quad \frac{\varrho_i(\alpha)(s_j) > 0,\ |\overrightarrow{x}| = |\overrightarrow{y}|}{\begin{array}{c}\left({}^{s_i}_{p_i}\langle \mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha.A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : \mathrm{in}_{\mathrm{link}}(\overrightarrow{y})@\alpha.A_j \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N, \mathbb{L}\right) \\ \Longrightarrow_{\tilde{q}_1 \tilde{p}_i \cdot \tilde{\varrho}_i(\alpha)(s_j)} \left({}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel {}^{s_j}_{p_j}\left\langle q_1 : A_j \mid A'_j, d_j\right\rangle_{\varrho_j} \parallel N, \mathbb{L} \cup \{(\overrightarrow{x}, \overrightarrow{y}, i, j)\}\right)\end{array}}$$

$$(GR6)\quad \frac{\left({}^{s_i}_{p_i}\langle \mathrm{remove}_{\mathrm{link}}(\overrightarrow{y}).A_i, d_i\rangle_{\varrho_i} \parallel N, \mathbb{L}\right) \Longrightarrow_{\tilde{p}_i} \left({}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel N, \mathbb{L} \setminus \{(\overrightarrow{x}, \overrightarrow{y}, j, i) \mid \overrightarrow{x} \subset Var_j\}\right)}{}$$

$$(GR7)\quad \frac{{}^{s_i}_{p_i}\langle A_i, d_i\rangle_{\varrho_i} \parallel N \Longrightarrow_p {}^{s_i}_{p_i}\langle A'_i, d'_i\rangle_{\varrho_i} \parallel N, \text{ and } d_i \neq d'_i}{\begin{array}{c}\left(s_i ::^{p_i}_{\varrho_i} \langle A_i, d_i\rangle \parallel_{j=1,\ldots,m\ j\neq i} {}^{s_j}_{p_j}\langle A_j, d_j\rangle_{\varrho_j}, \mathbb{L}\right) \Longrightarrow_p \\ \left({}^{s_i}_{p_i}\langle A'_i, d'_i\rangle_{\varrho_i} \parallel^m_{j=1,j\neq i} {}^{s_j}_{p_j}\left\langle A_j, d_j \sqcup (\bigsqcup_{\overrightarrow{x},\overrightarrow{y}:(\overrightarrow{x},\overrightarrow{y},i,j)\in\mathbb{L}}(\Pi_{\overrightarrow{x}} d'_i[\overrightarrow{y}/\overrightarrow{x}])\right\rangle_{\varrho_j}, \mathbb{L}\right)\end{array}}$$

Table 5. Syntax and rules for linking variables

the possibility of adding or removing dynamically some links during the execution. The creation of a link between variables involves two agents located at two different nodes, one of them publishing some of its variables, which are going to receive information, and the other linking some of its variables with the published ones. This bidirectional communication is implemented via the instructions $\mathrm{in}_{\mathrm{link}}(\overrightarrow{x})@\alpha$ and $\mathrm{out}_{\mathrm{link}}(\overrightarrow{y})@\alpha$, and their functioning mechanism is depicted in rule (**GR5**). This rule works similarly to (**GR2**) and (**GR3**) for what regards the preconditions and the probability of transitions, while the establishment of a link is modeled by adding the tuple $(\overrightarrow{x}, \overrightarrow{y}, i, j)$ to $\mathbb{L}$.

Links can also be removed by the instruction $\mathrm{remove}_{\mathrm{link}}(\overrightarrow{y})$. This instruction can be called by an agent which has previously published (one or more times) the variables $\overrightarrow{y}$. Its effect is described in rule (**GR6**): it removes all links publishing information to all variables $\overrightarrow{y}$. Note that subvectors of $\overrightarrow{y}$ can still remain linked.

The last rule of Table 5 deal with the actual transmission of information between linked variables. It states that, if the network, without any linking, can perform a transition with a probability p, and this transition modifies the content of the constraint store at a node (say i), then the linked network will automatically broadcast the information to all variables linked to some variable of node i. This transmission is an high level activity of the network, and it requires a global knowledge of the configuration of the network. In addition, it is performed simultaneously with the local evolution of node i. This can be justified by thinking that the broadcasting action is performed by the global scheduler before letting any other node evolve. Otherwise we can imagine that the linked variables represent some form of shared memory, in such a way that one process can write and another process can read it. Anyway, it would probably be better to allow some form of controlled delay for this transmission. In this way, we can model in

a more physically sound way the flow of information among the network: the more two nodes are far away, the longer the delay in receiving the information. Introducing delays, however, requires a modification of the semantics of the system. In particular, we must modify the concept of configuration: storing the actual state of the computation is no more sufficient, as now the system must have a form of (limited) memory of its past history. Therefore, one needs to remember, for instance, the last n states of the system, if $n - 1$ is the maximum delay occurring in a linking action. Note that the notions of computational path and input / output observables of the system can be extended in a straightforward way to include linking instructions.

7 Conclusions and Future Work

In this paper we extended the CCP computational paradigm with both distributed and probabilistic features. The resulting language is composed by two structural levels: one local and one global. The local entities are nodes, possessing their own constrain store, where the computation evolves according to a probabilistic version of CCP rules [5]. At this local level time is discrete and communications are asynchronous. These nodes are then connected in a global network, and they can exchange information through communication channels. Communication at this level is synchronous and performed in a π-calculus style. Three different objects can be exchanged in the network: constraint abstractions, channels and agent abstractions. Moreover, also the evolution of the network proceeds following probabilistic rules, and the time is discrete also at this level. Finally, the topology of the network is extended with a mechanism for remote synchronization of variables belonging to different constraint stores.

This language can be used for modelling distributed optimization algorithm, like simulated annealing, genetic algorithms and similar ones. Its declarative nature makes these programs very simple to write, while its stochastic nature makes possible to derive some properties of the optimization process just by looking at its semantics (or, more realistically, at some suitable abstraction of it). We have also wrote a metainterpreter in prolog for a subset of the language, and we are extending it to the full featured one.

In the future, we plan to extend the language by adding more features at the network level. In particular, we want to introduce networks of dynamic size, where the scheduling probabilities can change during time. In addition, we plan to develop also a continuous time version of the language, where scheduling probabilities among nodes are substituted by exponential rates. Finally, we want to provide the language with a denotational semantic, following the approach of [4], ad then use probabilistic abstraction techniques [6] to perform some kind of static analysis [7].

References

1. F.S. de Boer, R.M. van Eijk, W. van der Hoek, and J-J.Ch. Meyer. Failure semantics for the exchange of information in multi-agent systems. In *Proceedings of CONCUR 2000*, 1998.
2. F.S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1), 1995.

3. R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
4. A. Di Pierro and H. Wiklicky. A banach space based semantics for probabilistic concurrent constraint programming. In *Proceedings of CATS'98*, 1998.
5. A. Di Pierro and H. Wiklicky. An operational semantics for probabilistic concurrent constraint programming. In *Proceedings of IEEE Computer Society International Conference on Computer Languages*, 1998.
6. A. Di Pierro and H. Wiklicky. Probabilistic abstract interpretation and statistical testing. In H. Hermanns and R. Segala, editors, *Lecture Notes in Computer Science 2399*. Springer Verlag, 2002.
7. A. Di Pierro, H. Wiklicky, and C. Hankin. Quantitative static analysis of distributed systems. *Journal of Functional Programming*, To appear.
8. D. Gilbert and C. Palamidessi. Concurrent constraint programming with process mobility. In *Proceedings of CL 2000*, 2000.
9. Rety. J. H. Distributed concurrent constraint programming. *Fundamentae Informaticae*, 34(3):323–346, 1998.
10. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras, Part I*. North-Holland, Amsterdam, 1971.
11. O. M. Herescu and C. Palamidessi. Probabilistic asynchronous π-calculus. In J. Tiuryn, editor, *Proceedings of FOSSACS 2000*, Lecture Notes in Computer Science, pages 146–160. Springer Verlag, 2000.
12. Brim L., Gilbert D., Jacquet J., and Kretinsky M. Multi-agent systems as concurrent constraint processes. In *Proceedings of SOFSEM 2001*, 2001.
13. M. Milano and A. Roli. Magma: A multiagent architecture for metaheuristics. *IEEE Trans. on Systems, Man and Cybernetics - Part B*, 34(2), 2004.
14. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Springer Verlag, 1994.
15. C. Priami. Stochastic π-calculus. *Computer Journal*, 38(7):578–589, 1995.
16. M. Resende, P. Pardalos, and S. Duni Ekşioğlu. Parallel metaheuristics for combinatorial optimization. In R. Correa et al., editors, *Models for Parallel and Distributed Computation - Theory, Algorithmic Techniques and Applications*, pages 179–206. Kluwer Academic, 2002.
17. V. A. Saraswat. *Concurrent Constraint Programming*. MIT press, 1993.
18. V. A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proceedings of POPL*, 1991.
19. Paul C. Shields. *The Ergodic Theory of Discrete Sample Paths*, volume 13 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, Rhode Island, 1996.
20. 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.