K-Stores
A Spatial and Epistemic Concurrent Constraint Interpreter

Andrés F. Barco S.
AVISPA Research Group
Pontificia Universidad Javeriana
Santiago de Cali, Valle del Cauca, Colombia
anfelbar@javerianacali.edu.co

Sophia Knight,
Frank D. Valencia
INRIA and LIX, École Polytechnique
Palaiseau, France
sophia,frank.valencia@lix.polytechnique.fr

Abstract

Concurrent constraint programming (ccp) is a mature formalism for reasoning about concurrent systems that exhibit a constrained behavior. Spatial ccp and epistemic ccp are two novel variants of ccp currently being developed by Knight and Valencia. These variants model systems with spatial hierarchies of group information and knowledge. These systems are ubiquitous due to the advent of social networks and cloud computing where agents may share certain information with certain groups.

This paper introduces an interpreter for these extensions we call k-stores. The interpreter is a Prolog implementation of the operational semantics of the languages allowing the programmer to simulate distributed information systems. The main feature consists of an implementation of a spatial (distributed) store that allows epistemic information in it. The system supports the specification of (named) processes along with the ccp classic primitives, namely, ask and tell operations. The declarative view of processes is inherited from the ccp extensions. The orthogonal implementation of the local space abstraction and the epistemic constraint system makes further extensions possible. Special attention is paid to the representation of distributed knowledge and common knowledge.

Introduction

In current information systems sharing knowledge (information such as photos, links, phrases, or passwords) is an event that occurs on a daily basis and in a distributed fashion. From bank transactions to tags on photographs, pools of knowledge are generated from different types of interactions. Interesting applications come along with the advent of technologies such as semantic web [15], grid computing [4] and amorphous computing [1] in which agents interact using, or taking into account, some knowledge representation and premises. As these epistemic scenarios are common in computer science, i.e., computational agents trying to reach some *first-order knowledge* or construct *higher-order knowledge*, the necessity of reasoning tools increases. Tools such as interpreters, compilers and model checkers, are a remarkably important element for understanding the underlying properties of a given system.

Concurrent constraint programming is a mature formalism for reasoning about concurrent systems [20]. In this formalism agents are modeled as processes that interact with each other by means of a shared medium. Classical operations read and write are replaced by ask and tell primitives. Information in this calculus is represented as constraints (i.e., first order formulas such as $x+y=3$) and is kept shared by means of a global storage medium, a so-called constraint store. Several extensions to this model have been proposed in the literature within the last 20 years or so. Each of those extensions addresses particular notions of concurrency and reactive interactions. Process calculi extending temporal relations (tccp [18]), linear relations (lccp [6]) and nondeterminism (ntcc [17]) are good examples of such extensions.

A particularly novel extension of ccp is spatial-epistemic ccp; an extension for distributed information and knowledge [13]. Although not the first attempt at modelling distributed (local stores), spatial and epistemic ccp neatly combine the idea of distributed stores with the declarative nature of ccp. It means that we can take advantage of techniques in the logical theory to reason about knowledge and distributed information.

There are several interpreters and other tools for concurrent constraint programming. Different implementation for different flavours of the ccp model are found in the literature. Some of these implementations are for very popular constraint systems among scientists; CLP SWI-Prolog, Gecode library, Chocho library, Mozart-Oz and others. Common characteristics among them include the implementation of finite domain constraints and full store representation.

We contribute to the field of concurrency by introducing a novel tool called k-stores; a spatial and epistemic concurrent constraint interpreter[1]. We use the SWI-Prolog system to build a simple yet powerful interpreter modeling the operational semantics of the languages. The main feature is an implementation of a spatial (distributed) store enhanced with an epistemic constraint system. The system supports the specification of (named) processes along with the ccp classic primitives, namely, ask and tell operations. As it models the constraint programming paradigm, it makes it possible to add partial information to agents' stores by means of atoms and first-order formulas. The declarative view of processes is inherited from the calculus and necessary functionality (e.g. interleaving, constraint posting and information access) are added. Orthogonal implementation of the local space abstraction and the epistemic constraint system make further extensions possible.

The paper is structured as follows: in the next section we will present a brief background of spatial ccp and epistemic ccp. In particular we are interested in presenting the operational semantics of the calculus. Section 2 shows the design of the interpreter and builds the notion of a distributed implementation of the store. Constructors for programs and the general mechanism of the interpreter are explained in section 3. Finally some conclusions and future work are discussed.

1 Preliminaries

Concurrent constraint programming (ccp) is a computational model where agents interact with each other by means of a shared medium [20]. The fundamental aspect of ccp is the view of the store as a conjunction of partial information instead of the store as valuation, a central notion of the Von Neumann computing paradigm. In this sense, communication of processes (agents) is made by the shared monotonic store. The notion of read is changed for the notion of ask, which takes a constraint c and tests whether it can be inferred from the store. The write operation is replaced by tell, which adds partial information, represented as a constraint, to the store. The formalism takes a declarative view of processes that enables the implementation of different reactive systems.

Yet, this ccp viewpoint is not appropriate for the modelling of distributed information given the local computing constraints agents may have, i.e., both local processing and storage. Spatial and Epistemic ccp extend concurrent constraint programming calculus in order to capture some notions of spatiality and knowledge that have not been addressed by other extensions. In particular, the epistemic structure S4 [7] is used as the underlying logic. In this section we present some notions of these calculi. It is worth noting that both calculi, whose mathematical-theoretical foundations are currently under development [13], are not part of the contribution of

[1] Do not use copy/paste on the pdf to avoid mistakes: `http://cic.javerianacali.edu.co/~anfelbar/Sitio_web/teccpf/`

the present work. Rather, we use the underlying model to build the programming environment. We shall begin with the definition of the process construction terms.

Definition 1. *Construction terms for processes.*
Let P and Q be two spatial-epistemic ccp processes. We define the language of construction terms as:

$P, Q... :=$	0	*skip*
	$tell(c)$	*adds partial information*
	$ask(c) \rightarrow P$	*asks whether c, then executes P*
	$P \parallel Q$	*parallel execution*
	$[P]_i$	*local scope*
	X	*process invocation*

We assume that for each process variable X there exists a process definition, possibly recursive, of the form $X \stackrel{def}{=} P$.

These processes are defined over an underlying spatial or epistemic constraint system. This is much like a standard constraint system, a set Con with partial order $\sqsubseteq$ with a few particular properties. A spatial or epistemic constraint system has, in addition to the properties of standard constraint systems, a special function $\mathfrak{s}_i : Con \rightarrow Con$ for each agent i. In the constraint system, $\mathfrak{s}_i(c)$ represents c holding in agent i's space or store. The details of these constraint systems can be found in [13].

Intuitively, each agent i has his own local store $\mathfrak{s}_i(\cdot)$ where processes and other agents' stores may reside and execute. Thus, $\mathfrak{s}_i(c)$ means that in the store attributed to agent i the constraint c holds. Along the same lines, $\mathfrak{s}_i(\mathfrak{s}_j(c))$ should be viewed as a hierarchical specification meaning that c holds in agent j's store, which is a store that agent i is attributing to j. The spatial construct $[P]_i$ represents the process P executing within the store of agent i.

Additionally, the semantics of tell play a two-fold role. Executing a spatial operation, the process $[tell(c)]_i$ only adds partial information to the agent's store[2], without adding information to the global store. This is akin to the notion of belief; agent i may believe that it is raining while agent j believes it is not. Also, it reflects the distributed nature of agents as they may have different information about the true state of affairs. This implies that no spatial tell operation can cause the overall computation to fail but rather only can fail the computation of the executing agent, i.e., $[tell(false)]_i$ does not produce $false$ and $[tell(c)]_i \sqcup [tell(d)]_j$ does not produce $false$ even when $c \sqcup d = false$.

The second role for the tell operator is as an epistemic view[3]. Essentially, it is used as an epistemic modal operator to represent whatever is true in the system. But also, $[\cdot]_i$ makes it possible to model the viewpoint that a given agent may have over the constraint store, i.e., capturing the knowledge (rather than belief) of a given agent. Intuitively, the process $[tell(c)]_i$ causes c to be added to the knowledge of agent i. Some interesting properties of the epistemic tell operator are:

- After $[tell(c)]_i$ is executed, the store entails c, meaning that if an agent knows something then it is true. However, this does not mean that the other agents know c.
- $[tell(c)]_i$ is idempotent; meaning that agent i knows that he knows c.
- After $[[tell(c)]_j]_i$ is executed, c will be in i's store. This is because if i knows that j knows c, he can conclude c from this fact.

[2] In what follows, we refer to this spatial operation as *spatial tell*.
[3] In the remaining of the paper, we shall refer to the epistemic operation as *epistemic tell*.

1.1 Spatial and epistemic ccp operational semantics

The operational semantics defines the transformation of spatial-epistemic ccp processes, i.e., rules for the reduction of processes specified in the language (there is also a declarative denotational semantics for this calculus available in [13]). The parallel rule, for instance, allows two processes to be executed concurrently (at the same logical time). In what follows, $\mathfrak{s}_i(c)$ can be thought of as representing agent i's belief or knowledge of c, and c^i represents what agent i believes or knows when c holds: $c^i = \bigsqcup \{d \mid \mathfrak{s}_i(d) \sqsubseteq c\}$ (see [13] for further details). The operational semantic rules common to both languages are shown in table 1.

$$tell : \frac{}{\langle \mathbf{tell}(c), d\rangle \rightarrow \langle \mathbf{0}, d \sqcup c\rangle} \qquad ask : \frac{c \sqsubseteq d}{\langle \mathbf{ask}(c) \rightarrow P, d\rangle \rightarrow \langle P, d\rangle}$$

$$spatial : \frac{\langle P, c^i\rangle \longrightarrow \langle P', c'\rangle}{\langle [P]_i, c\rangle \longrightarrow \langle [P']_i, c \sqcup s_i(c')\rangle} \qquad parallel : \frac{\langle P, d\rangle \longrightarrow \langle P', d'\rangle}{\langle P \parallel Q, d\rangle \longrightarrow \langle P' \parallel Q, d'\rangle}$$

$$recursive : \frac{\langle P, d\rangle \rightarrow \gamma}{\langle A, d\rangle \rightarrow \gamma} \text{ where } A \stackrel{def}{=} P \text{ is a process definition}$$

Table 1: Spatial and epistemic ccp operational semantics. The symmetric rule for parallel evolution of Q, is omitted.

The spatial rule says that if P could evolve to P' based on what i believes when c is actually true, then when c holds, P can evolve to P' inside i's store, adding whatever information is computed inside of i's store. The other operational semantic rules behave as expected. An additional rule for the epistemic operator, in table 2, states that in the epistemic case, information (facts) are propagated upwards.

$$epistemic : \frac{\langle [P]_i \parallel P, c\rangle \rightarrow \langle [P]_i \parallel P', c'\rangle}{\langle [P]_i, c\rangle \rightarrow \langle [P]_i \parallel P', c'\rangle}$$

Table 2: Aditional rule for epistemic ccp operational semantics.

Common knowledge. Individual knowledge is also referred to as *first order knowledge*, i.e., atomic and monotonic knowledge that is not part of an inference nor interactive process. *Higher order knowledge* refers to the knowledge that is not explicit in the predicates but that is somehow built from them. For example, if we have a set of agents $G = \{a_1, ..., a_n\}$ and each of them knows some predicate ϕ, then we can denote by $E_G\phi$ the event that "everybody in group G knows ϕ". This can be expressed using the individual knowledge operator from epistemic logic [7], see fig 1 equation a.

$$E_G(\phi) = \bigwedge_{i \in G} K_i(\phi) \qquad CK_G(\phi) = E_G(\phi \wedge CK_G(\phi))$$

a) Everybody in group G knows ϕ. *b*) Event E is common knowledge for group G.

Figure 1: Mutual knowledge modelling.

Intuitively, the expression can be extended to model "everybody knows that everybody knows ϕ" by means of the conjunction of all individuals' knowledge and nesting of the knowledge operators, denoted as $E^2_G(\phi) = E_G(\phi \wedge E_G(\phi))$.

Common knowledge can be defined as the greatest fixed point of equation b. This can be thought of as "everybody knows that [k times] the event ϕ is the case", i.e., $E^k_G(\phi))$ for arbitrary k.

This knowledge notion, necessary for any agreement on coordination or synchronization, is captured nicely by the epistemic ccp calculus. It does so by means of a nested store abstraction, i.e., allowing agent stores to have other local stores. For in a spatial-epistemic specification a group G of agents may have group knowledge of c when all agents in G know c. Consequently, a fact c may be common knowledge among G if all agents knows c, they all know they know c, they all know that they all know that they know c, and so on ad infinitum.

1.2 Distributed store

A constraint system is essentially an arbitrary set containing partial information (variable relations) along with the relation $\sqsubseteq$, called reverse entailment, that specifies when a given formula is implied by other formulas (e.g. $x+y > 0 \sqsubseteq x+y > 42$). Different constraint systems allow different relations on the variables, that is, the formula constructions are constrained by the underlying model. Typical constraint systems are finite domains, finite sets and boolean domains.

Relations can be extended to model a particular logic such as linear logic, temporal logic or epistemic logic. The underlying logic makes it possible to focus the simulation tools and proof systems on particular properties such as timeouts, linearity properties or knowledge interaction.

Despite the great expressive power of ccp, it does not allow an easy representation of agents with local processing and storage. The novelty of spatial ccp and epistemic ccp lies in the viewpoint change of the store. In this formalism the store is not a unique shared medium for the agents. Instead, each agent has its own projection of the store, and hence, of variable domains and constraints. Furthermore, agents are allowed to have beliefs about the beliefs that any other agent may have. This translates into a hierarchy of nested stores and distributed information.

The distributed store abstraction is quite similar to that for *distributed constraint satisfaction* problems. The distributed constraint satisfaction view regards solving problems as being done by different independent agents, as we do here. However, any given problem variable is controlled by a different agent, i.e., the agent is in charge of setting the variable value [9]. In constrast, the spatial-epistemic view holds that any agent has his own local store with his own representation of a subset of variables and a subset of constraints.

2 Implementing epistemic systems

In this section we show in detail the interpreter and its primary components. First, we address the design of the overall system explaining the way agents add and ask for partial information and how the constraint system and axiomatic epistemic system are related. Then we go deeper

into the issues involving the distributed store. Some useful graphics are presented to show the architecture.

2.1 Architecture

The core characteristic of the epistemic ccp calculus is the distributed (possibly nested) store. As the store is the shared communication medium of ccp it allows different agents to interact by telling partial information. Now, with a distributed store agents are only aware of some partial information within the store, we say that agents have a limited knowledge of the state of affairs, i.e., variables, domains, and constraints.

Our programming interpreter implements different abstractions needed for the proper simulation of the operational semantics of spatial and epistemic ccp: distributed store, constraint imposition and domain pruning, epistemic axiomatic rule consistency, along with the tell and ask operations. The basic design of the epistemic interpreter divides the different abstractions in order to a) make any component orthogonal to any other and b) allow the interface with different constraint systems and axiomatic system implementations. Figure 2 shows the design of the system at first glance.

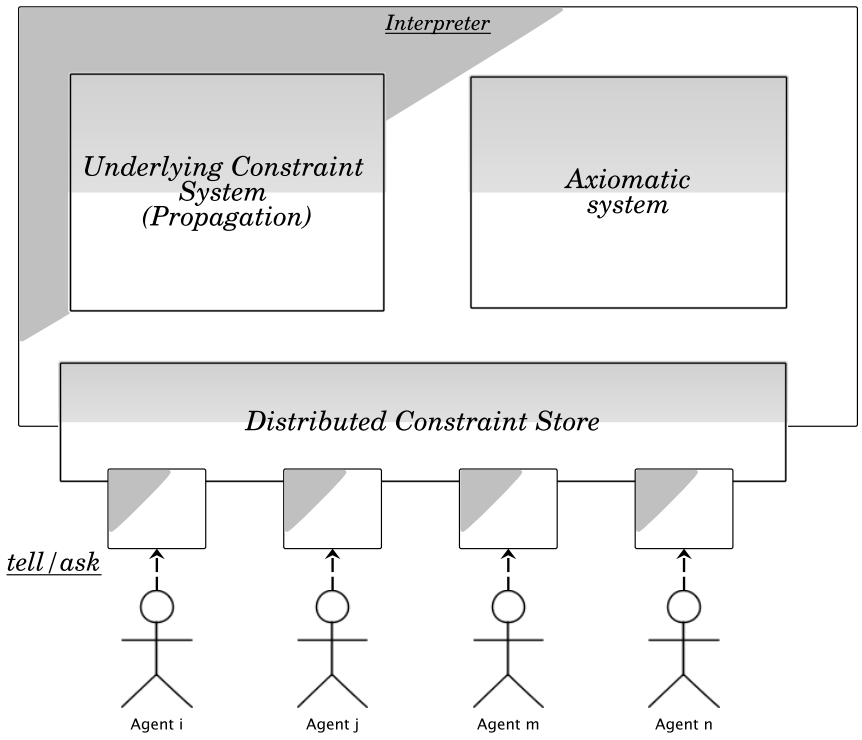

Figure 2: General architecture of the interpreter.

Spatial tell operation. This tell operation allows a given agent to add knowledge to his current knowledge partition, i.e., to add constraints to his own constraint store. Given that in an n-agent spatial constraint system there are n different local stores, any given agent is allowed to add information to any store referencing the agent the store belongs to. The purpose

of this primitive, on the one hand, is to add enough information to the agent's store so that an assignment for the variables may be inferred. An assignment θ is a mapping from variables to integers, i.e., $\{x_1 \mapsto val_1, x_2 \mapsto val_2, ..., x_n \mapsto val_n\}$. However, as any given agent has its own variable and domain view, then there must be an assignment for each agent. Let Θ represent a set of assignments such that $\Theta(i)$ gives the assignment θ for agent i. On the other hand, the spatial tell is a primitive that allows agents to communicate by means of opened channels and announcements.

Epistemic tell operation. This tell operation allows a given agent to add facts (knowledge) to his current knowledge partition. However, any epistemic tell is propagated upwards through all agents' stores in the hierarchy. For in an n-agent epistemic constraint system knowledge is indeed true information, i.e., facts. Thus, there is no place for false statements. So, if any agent knows a fact, it must be true. Consequently, facts in a local nested store must be propagated to any agent in the hierarchy up to the global store. Any agent outside the hierarchy, however, is not necessarily aware of such an epistemic tell. This models the notion that agents may have independent knowledge, and may be unaware of one anothers' knowledge. Epistemic tell in local stores may generate inconsistency in higher stores, and consequently, all computation fails.

Ask operation. Ask operations in the interpreter have two components and are expressed as $ask(s_1, c) \rightarrow (s_2, P)$. First, we have the constraint c and store s_1 in which we want to make the query. Second, we have the process P and the store s_2 in which we want the partial information to be added. This is akin to the notion of *post* in social networks. Semantics remains as in classic ccp, i.e., if the constraint c in the first part is not entailed in the store s_1, then the consequence is discarded. Moreover, if the first constraint c is entailed by the store, then the interpreter checks whether the consequence P can be applied or not in s_2. The consequence may not be applied if it results in a false domain. Additionally, given that an ask operation includes a consequence P, then we make a division about how to handle such a consequence. Basically, agents are allowed to make spatial consequences or epistemic ones. Similar to the semantics of tell, a system may be either spatial or epistemic. In the spatial setting, the results of an ask operation remain in the local store. In the epistemic setting, the ask operator propagates up through the hierarchy of agents' views of the variables.

Constraint system. Once an agent executes a tell operation on a particular store, the constraint system ensures domain pruning. For in a constraint system, relations over variables are imposed by constraints which implies narrowing variable domains whenever a change occurs. Our underlying constraint system, CLP SWI-Prolog, allows some constraint imposition which behaves concurrently removing values from domains. It implements several constraints over finite-domain variables (e.g. $<$, $\leq$, $>$, $\geq$, all_distincts, sum). In our interpreter the narrowing process taking place in a particular store only uses information in that store. It means that any change in the store of agent i affects the computation of that agent. Any other store and computation in the remaining $n-1$ processes remains intact.

Axiomatic system. The axiomatic system of epistemic logic is implemented as a set of rules. Essentially, it ensures that the rules are applied when an agent is asking whether a first order formula is entailed by the store. In particular it ensures that the following rules apply (typical names for each axiom are provided, see [12, 14] for a detailed explanation). Both ϕ and ψ are constraints (i.e. first order formulaes) and $\top$ represents true.

$K_i(\top)$	: Self evidence axiom.
$K_i(\phi \wedge \psi) \equiv K_i(\phi) \wedge K_i(\psi)$	: Deducible closure or distribution axiom.
$\phi \sqsubseteq K_i(\phi)$	: Truth axiom.
$K_i(K_i(\phi)) \sqsubseteq K_i(\phi)$	: Positive introspection axiom.

In order to simplify the implementation, we add a primitive which is a combination of tell operations. The primitive is *everyone(c)* tell; whatever the constraint c is, it must be added to all individual stores. This combinator is an abstraction for the programmer, without it the programmer could execute several tell operations to make a constraint known in all local stores.

2.2 Distributed store design

A distributed store is just a projection of the knowledge of all agents over the variable domains. Any given agent is aware of a subset of all variables V of the overall store. Furthermore, agents are allowed to know a limited set of constraints over the variables of which they are aware. This projection is the realization of the K_i modal operator of the logic.

In several applications some variables are publicly accessible and some other variables are only known by a subset of agents. This is the case for emails and passwords. Furthermore, in many real-life scenarios information may come from different sources, in such a way that centralization can never be achieved[9]. These are properties that we need to model.

So we want a distributed store that maintains different properties; agents should be allowed to know a subset of variables and a subset of domains; the asking and telling operations are local to the owned store; inconsistency in any local store does not affect other stores and some public annoucements may be allowed in the specification of programs.

All these features are implemented by K-stores in a simplistic way. A reflection mechanism is in charge of the communication among agents. Note that a given agent can allow another agent to access its store and post constraints over a subset of variables, opening a communication channel. This means that one agent can ask for a particular property to hold on another local store. This resembles the *friend* access modifier in object oriented programming; any friend class is able to interfere with and modify private data.

Global store. Since we want to represent the ccp and spatial-epistemic ccp views, we define a global shared store where all tell operations are made. This is useful if we want to know whether the intersection of all agents' local stores are consistent or not. We may think about this global store as an implementation of the traditional ccp store. Essentially, the global store is the least upper bound ($\sqcup$) of all individual agents' stores and, of course, may include false, meaning there is a contradiction and the computation has failed. For example, agent i has $x \in \{1..5\}$ and agent j has $x \in \{3..7\}$ then an epistemic tell of $x := 2$ by agent i causes the computation of agent j to fail. Epistemic tell for an agent is only propagated to the global store and to those other agents' stores that are aware of the agent's knowledge. For in an epistemic system agents are not allowed to know false statements.

Reflection mechanism. This mechanism is in charge of locking and unlocking certain variables in the local store of any given agent. It keeps track of opened channels among agents so that ask and tell operations can be applied among stores. If an agent wants a variable to be seen by another agent, he opens the channel by declaring the variable to be global. Without such an opened channel other agents are not allowed to ask nor to tell partial information to that agent's store. Nevertheless, in the spatial (non-epistemic) case, whatever an agent believes about other agents does not necessarily have to relate to what the other agent really knows, i.e., agents may think anything about the state of variables of other agents even when they are absolutely wrong.

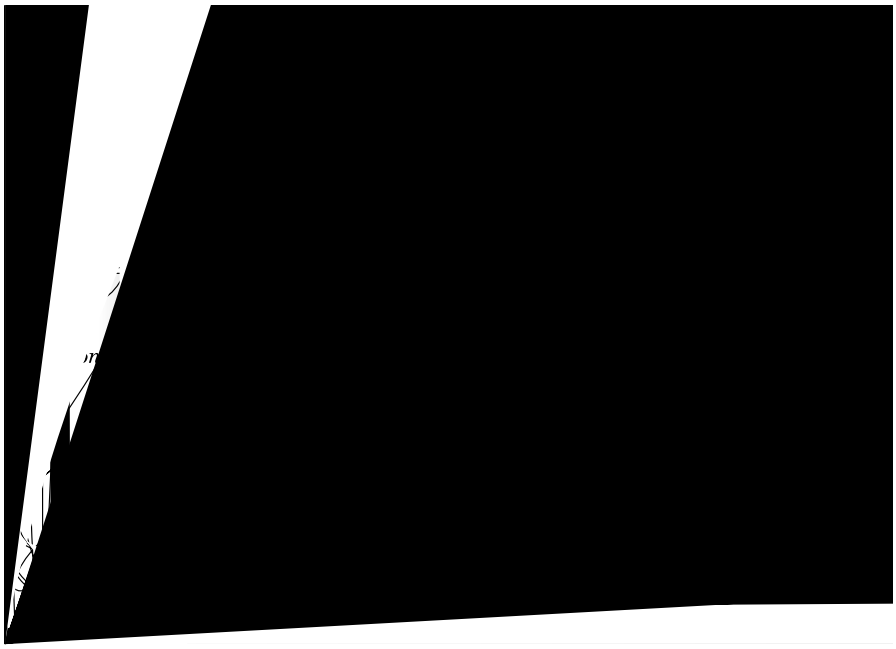

Figure 3: Each store may be accessed by means of a reflection mechanism.

Local constraint imposition. Constraint posting in local stores is the same as in classic ccp. Yet, we have a fundamental difference: any given agent may have some beliefs (or knowledge) about other agents' beliefs (or knowledge). The core design is nested stores. Any given agent i is waiting for its variable assignment θ_i and he may use whatever beliefs he has about other agents to determine the variable assignment. If the agent has beliefs about another agent, say j, then i has in his store another store representing the beliefs of agent j. Then, agent i may ask and tell partial information to their own local representation of j's beliefs. Moreover, agent j's local store may have some beliefs about another agent, say l, in which case we have nested stores working together to find the assignment θ_i for agent i.

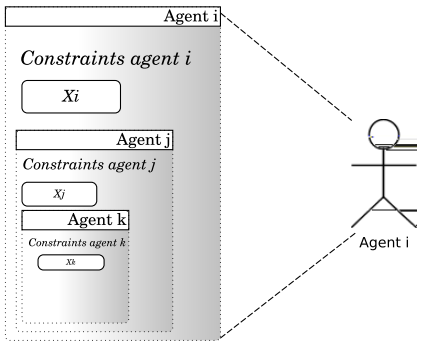

Figure 4: Agent i may have nested stores (agent i's belief).

3 A simple interpreter

According to Borshchev et al. [3] a simulation model may be seen as a set of rules that explains how a system passes from its initial state to future states. A simulation is, then, the process of model execution that leads the system through state changes over time [3].

The K-stores interpreter is an implementation of the spatial and epistemic ccp formalism, i.e., its operational semantics, along with an underlying constraint system, specifically, the SWI-Prolog constraint logic programming. The basic construction in the interpreter is specifications of spatial-epistemic ccp agents that interact with each other simulating the semantics of the language. Programs' inputs (processes) are executed in an interleaved fashion and outputs (store information) are shown in the standard Prolog output.

3.1 General mechanism

Constraint programming is concurrent in the sense that processes are executed in an interleaved fashion. Execution of processes implies ask and tell operations that possibly will narrow variable domains which in turn may unblock some other ask operations. The general mechanism is shown in fig 5.

Interleaved process execution is a basic strategy for multiplexing a processor unit in a typical operating system. There exist many different multiplexing algorithms in the literature (e.g. round robin, priority scheduling, first-come first-served). Each one of those algorithms defines certain criteria to assign the processor to a given process. Our implementation is based on an early implementation of the Erlang language as described by Armstrong et al. [2] that uses a round robin strategy. We choose the round robin algorithm for the system because:

- It has a standard sense of fairness, i.e., any given process will be executed eventually.
- It uses the execution steps (e.g. reductions, tell operations, clock ticks) as a criteria to know how many times the process will be executed.
- As the spatial-epistemic processes do not have priorities we only create a circular structure for iteration.
- As spatial-epistemic ccp processes do not synchronize, we can execute any process independently from any other.

Of course the round robin algorithm is not useful for the execution when the processes are executed in different processor units. However, the present work does not address execution in different nodes nor processor cores. That functionality is postponed for further research.

3.2 Construction of programs

The K-stores tool allows parallel execution of (named) spatial-epistemic ccp agents. A program consists of a list of agent specifications. Any specification must be a spatial or an epistemic one; we do not allow the two types to be mixed. All specifications are run in parallel with the following notation

```
<program> ::=  parallel(<agent_list_spec>)
```

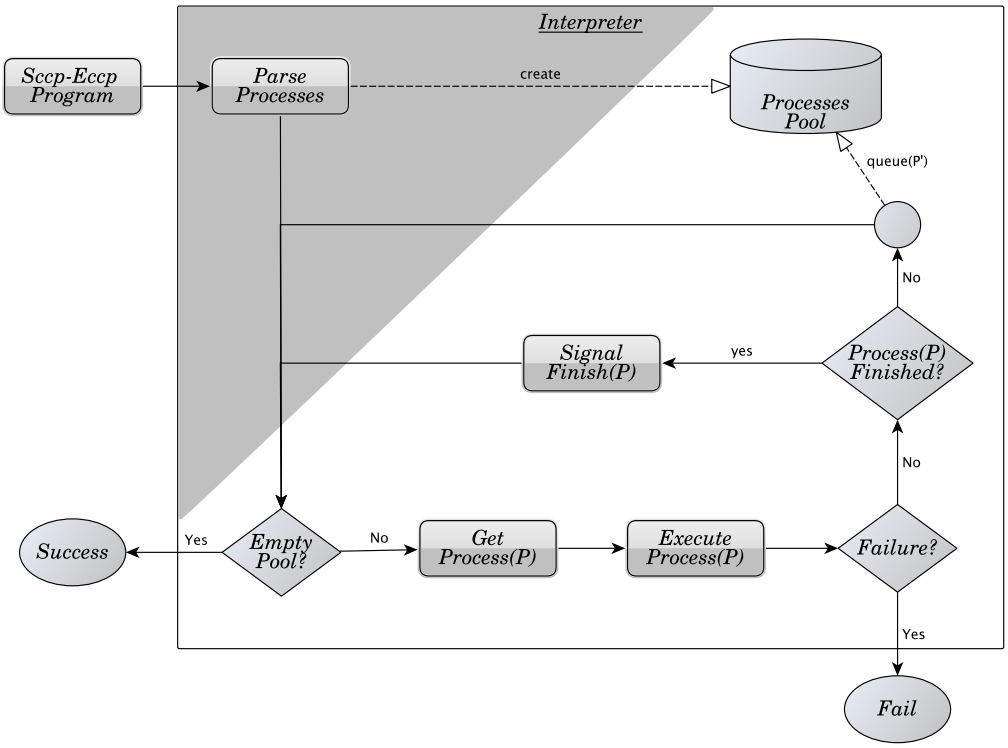

Figure 5: General mechanism for program execution.

An agent consists of an identifier (agent name) and a list of operations which may be tell and ask operations and another agent's specifications (fresh nested stores). Two atoms are avaliable for the creation of a new named agent. The first atom is *spatial* which creates a spatial ccp agent. The second atom is *epistemic*, and defines a new epistemic ccp agent in the program. These agent definitions are exclusive, i.e., combining them in a program specification is not permitted. All *Ids* should be pairwise distinct in order to avoid ambiguity. Additionally, we include a construction for making knowledge known to every agent. We call this construction *everyone*. The syntax of these constructions is

```
<agent_list_spec> ::=  <agent_spec> {,<agent_list_spec> *} |
                       everyone(<constraint_list>) {,<agent_list_spec> *} |
<agent_spec>      ::= {spatial | epistemic} (<id>, <operation_list>)
```

If an agent knows a constraint variable he must declare it to be within a given domain. He may do so by declaring the variable global or local to his own store. Declaring a global variable represents opening a public channel where other agents may ask and tell partial information. To declare a global variable the agent uses the *declare* atom along with the variable and its domain. If the variable is supposed to be local, then the atom *tell* is enough to make the declaration.

A tell operation is done with the atom *tell* along with a constraint (e.g. tell(X # > 10)). Used without an identifier it means a tell in the owned store. Conversely, a tell with an identifier

adds partial information to the store of the agent represented by the identifier (provided an open channel between the two agents exists). For nested stores, there is no need for an opened channel, as expected. The operation list is too long to include, but an important part is

```
operation_list  ::=  tell(<var> in l..u) {,<operation_list> *} |
                     tell(<constraint>) {,<operation_list> *} |
                     tell(<id>, <constraint>) {,<operation_list> *} |
                     <agent_spec> {,<operation_list> *}
                     ...
```

An agent is allowed to ask whether certain formulae are entailed by the store and then apply another constraint given the response. The first way to do this is by asking in any store (with an open channel) whether a formula is entailed and then applying a constraint c in the owned store (using the atom *askI*). The second possibility is to ask in any store (with an open channel) whether a formula is entailed and then apply a constraint c in the store in which the query is made (using the atom *ask*). This abstraction enables any given agent to a) change his current store provided some property holds or change his belief about other agents in his nested stores, and b) to change the current belief of other agents by means of partial information posting (as is done in social networks and amorphous computing).

Also, another non-formal operation is allowed for agents, *solve*, which invokes search from the contraint logic programming module of Prolog. This last primitive is useful when solving puzzles. If used, it should be put in the last position of the operation list. The final part of the operation list is

```
<operation_list> ::= ...
                     ask(<id>, <constraint> → <constraint>)  {,< operation_list> *} |
                     askI(<id>, <constraint> → <constraint>)  {,< operation_list> *} |
                     solve(<var_list>)
```

Tell operations are used when an agent wants to add partial information to any given store. Partial information is represented by first order formulae. For a practical development we -30

Example 1 (Distributed send+more=money)**.** *The Send More Money Problem consists of finding distinct digits for the letters D, E, M, N, O, R, S, Y such that S and M are different from zero (no leading zeros) and the equation SEND+MORE=MONEY is satisfied. The unique solution of the problem is 9567+1085=10652.*

The first attempt to solve the problem uses a spatial division. Two agents are trying to reach a solution. Agent `katherine` ignores that all variables are pairwise disticnts, whereas agent `andres` does not know that `s` and `m` are can not be zero.

```
parallel([
  spatial(katherine, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10, E*1],
             #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell( M #\= 0), tell(S #\= 0),
    tell(solve([S,E,N,D,M,O,R,Y]))]),
  spatial(andrew, [tell([S,E,N,D,M,O,R,Y] ins 0..9),
    tell(sum([S*1000, E*100, N*10, D*1, M*1000, O*100, R*10, E*1],
             #=, [M*10000, O*1000, N*100, E*10, Y*1])),
    tell(all_different([S,E,N,D,M,O,R,Y])),
    tell(solve([S,E,N,D,M,O,R,Y]))])
]).
    Knowledge for agent katherine --> {9,0,0,0,1,0,0,0, }
    Knowledge for agent andrew --> {2,8,1,7,0,3,6,5, }
```

In this specification, agents have different information about the state of affairs, i.e., the complete set of constraints. Thus, neither agent is able to reach the valid answer[4].

Example 2 (Client discounts)**.** *A vendor sells phones, laptops, and tablets. All products prices ranges from 500 to 3000 dollars. He has three different discounts, one for each product. A given client is allowed to have only one discount.*

```
parallel([
  epistemic(vendor, [
    epistemic(client, [tell([Phones, Laptops, Tablets] ins 500..3000), tell(Discount in 1..3)]),
    tell('client:own', tell(Discount#=1))]),
  epistemic(client, [tell([Phones, Laptops, Tablets] ins 500..3000), tell(Discount in 1..3),
    askI(client, Discount#=1 -> Phones#=<1000),
    askI(client, Discount#=2 -> Laptops#=<2000),
    askI(client, Discount#=3 -> Tablets#=<2500)])
]).
```

When the vendor executes the statement `tell(Discount#=1)` he means that he knows that the clients knows the discount is equal 1. Hence, the client must actually know that he has a discount on phones. The output of the program is

```
    Knowledge for agent vendor --> {500..1000,500..3000,500..3000,1, }
    Knowledge for nested:vendor's representation of client --> {500..1000,500..3000,500..3000,1, }
    Knowledge for agent client --> {500..1000,500..3000,500..3000,1, }
```

[4]The symbol ø means that there is no more knowledge in that particular store.

4 Concluding remarks and future work

Theories for reasoning about concurrent systems and knowledge interaction are a major research field nowadays [11, 7]. As these theories evolve an implicit need must be satisfied: the use of these theories in real-life applications. The system presented in this paper is a realization of the operational semantics of the novel calculi spatial ccp and epistemic ccp. Although it is just a basic tool, it can be used for educational purposes in epistemology, logic and concurrency. Also, it can be used to simulate processes that have their own local storage and may gain information by means of a shared medium, such as amorphous computing and social networks.

Social networks, mainly web-based networks, are a growing field of research because of their complexity and ubiquity. In such multi-agent systems information flows in huge magnitudes from client to client. Two fundamental features in these networks are the private data locality and the (possibly constrained) public information posting that is allowed for agents. Furthermore, information exchange may take place within a subset of agents inside the network. Moreover, different information (knowledge) shared inside the network may become common knowledge throughout the entire network (e.g. worldwide disasters).

It seems likely that simulation of social networks is a necessary condition for their understanding. An interpreter is an initial and basic tool for this task. Interpreters allow system models to be mapped and simulated, allowing the programmer to build systems with a specified behaviour and logic. These kinds of tools are an active research topic which have been demonstrating their capabilities in recent years. Results of such research include several programming environments.

One example of such results is *Timed Gentzen*. It is a framework for programming timed concurrent reactive systems [19]. This framework is an implementation of Timed ccp [16, 5] which is an extension to ccp made by Saraswat for programming and modelling timed reactive systems. *NtccRt* is another tool for analyzing systems which involve real-time interaction among agents [21]. This framework is built using the concepts and semantics for the ccp extension Ntcc [17]. Yet another tool is an interpreter in which the timed concurrent constraint programming paradigm is modelled; *Timed ccp*. Villanueva et. al [8] developed this interpreter in Maude[5] facilitating the simulation of concurrent agents and enabling some model checking involving different logics. Finally, we have the *MOLOG* implementation; an extension of Prolog using modal logics [10]. In that work Fariñas shows a method for extending Prolog with modal operators. They do so by analyzing and comparing two deduction approaches; one with modal resolution limited to modal Horn clauses and the other, called compilation, that transforms modal Horn clauses into clauses closer to classical Horn clauses. In particular, he uses epistemic logic to exemplify the deduction methods.

4.1 Relevant Perspectives

The main feature of the tool is to model a distributed and nested store. This feature allows us to think about computation made in different nodes. Furthermore, any given node may have its own constraint system (e.g. Choco, Gecode, Mozart-Oz) that helps with constraint imposition and possibly search. This means that the interpreter must be orthogonal to any platform and any constraint system. Along those line we propose certain previously unaddressed developments: a) to implement the system taking into account different nodes (a distribution layer); b) to interface the system with several constraint systems; c) to interface the system with different axiomatic system implementations in languages other than Prolog.

[5] http://maude.cs.uiuc.edu/

References

[1] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. Knight Jr., R. Nagpal, E. Rauch, G. Jay Sussman, and R. Weiss. Amorphous computing. *Commun. ACM*, 43:74–82, May 2000.

[2] J. Armstrong, S. Virding, and M. Williams. Use of Prolog for developing a new programming language. In *Proceedings of The practical Application of Prolog*, 1992.

[3] A. Borshchev and A. Filippov. From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools. In *Proceedings of the 22nd International Conference of the System Dynamics Society*, pages 24–29, 2004.

[4] E. Cody, R. Sharman, R. Rao, and S. Upadhyaya. Security in grid computing: A review and synthesis. *Decision Support Systems*, 44(4):749 – 764, 2008.

[5] F. de Boer, M. Gabbrielli, and M. Meo. A timed concurrent constraint language. *Information and Computation*, 161(1):45 – 83, 2000.

[6] F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Information and Computation*, 165(1):14 – 41, 2001.

[7] R. Fagin, J. Halpern, Y. Moses, and M. Vardi. *Reasoning About Knowledge*. MIT Press, 1995.

[8] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Computing Research Repository*, abs/cs/0505026, 2005.

[9] B. Faltings. *Distributed constraint crogramming*, pages 699–729. Foundations of Artificial Intelligence. Elsevier, 2006.

[10] L. Fariñas. Molog: A system that extends prolog with modal logic. *New Generation Computing*, 4:35–50, 1986.

[11] H. Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. Research Report RR-6368, INRIA, 2007.

[12] J. Geanakoplos. Common knowledge. In *Proceedings of the 4th conference on Theoretical aspects of reasoning about knowledge*, pages 254–315. Morgan Kaufmann Publishers Inc., 1992.

[13] S. Knight, C. Palamidessi, P. Panangaden, and F. Valencia. Spatial distribution of information in constraint-based calculi. *Lix École Polytechnique, Paris, France*, Technical Report, 2012. Available: http://www.lix.polytechnique.fr/∼fvalenci/papers/eccp-extended.pdf.

[14] F. Koessler. Common knowledge and interactive behaviors: A survey. Working Papers of BETA 2000-07, Bureau d'Economie Théorique et Appliquée, UDS, Strasbourg, 2000.

[15] C. Marshall and F. Shipman. Which semantic web? In *Proceedings of the fourteenth ACM conference on Hypertext and hypermedia*, pages 57–66. ACM, 2003.

[16] M. Nielsen and F. Valencia. Notes on timed ccp. In *In 4th Advanced Course on Petri Nets ICPN'03. LNCS*. Springer-Verlag, 2004.

[17] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, pages 302–316, London, UK, UK, 2001. Springer-Verlag.

[18] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80. IEEE Computer Press, 1994.

[19] V. Saraswat, R. Jagadeesan, and V. Gupta. Programming in timed concurrent constraint languages, 1994.

[20] V. Saraswat, X. Parc, and M. Rinard. Semantic foundations of concurrent constraint programming. pages 333–352. ACM, 1991.

[21] M. Toro, C. Rueda, G. Assayag, and C. Agón. Ntccrt: A concurrent constraint framework for real-time interaction. In *Proc. of International Computer Music Conference*, 2009.