Multi-Agent Systems as Concurrent Constraint Processes*

Luboš Brim[1], David Gilbert[1],Jean-Marie Jacquet[2] and Mojmír Křetínský[3]

[1] Dept.of Comp.Sci., City University, London, U.K.
drg@soi.city.ac.uk
[2] Dept.of Comp.Sci., University of Namur, Namur, Belgium
jmj@info.fundp.ac.be
[3] Faculty of Informatics, Masaryk University Brno, Brno, Czech Republic
{brim,mojmir}@fi.muni.cz

Abstract. We present a language `Scc` for a specification of the direct exchange and/or the global sharing of information in multi-agent systems. `Scc` is based on concurrent constraint programming paradigm which we modify in such a way that agents can (i) maintain its local private store, (ii) share (read/write) the information in the global store and (iii) communicate with other agents (via multi-party or hand-shake). To justify our proposal we compare `Scc` to a recently proposed language for the exchange of information in multi-agent systems. Also we provide an operational semantics of `Scc`. The full semantic treatment is sketched only and done elsewhere.

1 Introduction

Multi-agent system is a system composed of several autonomous agents that operate in a distributed environment which they can perceive, reason about as well as can affect by performing actions. In the current research of multi-agent systems, a major topic is the development of a standardised agent communication language for the exchange of information. Several languages have been proposed, e.g. [7, 9, 12, 4]. Recently de Boer et al.([4]) have also (for the first time) introduced a formal semantic theory for the exchange of information in the multi-agent systems. Their approach uses principles of concurrent constraint programming (CCP) to model the local behaviour of agents while the communication is modelled by a standard process algebraic hand-shake approach.

Our proposal is based on CCP paradigm only, however semantics of mechanism for updating and testing the (local/global) store(s) are changed.

CCP has inherited two of main features of concurrent logic programming, namely the asynchronous character of the communication and the monotonic update of the store. During last years some work have been done to lift both features. On the one hand, de Boer et al. [5] has proposed non-monotonic updates

* This work has been partially supported by the Grant Agency of Czech Republic grant No. 201/00/0400.

of the store and have studied compositional and fully abstract semantics for them. On the other hand, Saraswat proposed a synchronisation mechanism in ([10]). Briefly, his proposal and many related ones (e.g. [6]) are based on a coding of an explicit operator to achieve synchrony. This should be contrasted with the classical concurrent constraint framework in which asynchronous communication is simply obtained by the blocking of ask primitives when information on the store is not complete enough to entail the asked constraints. Following these lines, a natural alternative to obtain synchronous communication in CCP is to force ask and tell primitives to synchronise in some way. We elaborate further on this idea by proposing new versions of these primitives.

As in the classical CCP framework, our proposal makes use of tell and ask primitives. However, a new perspective is taken in that, to be reduced, any $tell(c)$ operation needs an $ask(c')$ partner. Restated in other terms, if the tell primitives are seen as producers of new information and ask primitives as consumers, the new primitives consist of lazy tell (or "just-in-time") producers forced to synchronise on their consumer asks. Stress is put on the novelty of information, i.e. on the fact that the told information should not be entailed by the current store. Consequently, any $tell(c)$ and $ask(c)$ operations whose constraint argument c is entailed by the current store can proceed autonomously.

The general scheme is enriched by permitting the synchronisation of more than two partners. Futher we allow some of the tell primitives *not to update* the store. These primitives are subsequently called fictitious and are denoted as $ftell$. They can be used to transmit the information which is not (yet) entailed by the (global/local) store – quite important possibility in distributed systems.

Comparing to the works cited above for synchronisation, we notice the advantage of our approach is that it permits the specification of on *what* information the synchronisation should be made, rather than with *whom*. Our synchronisation is thus more data-oriented as opposed to process-oriented (however still keeping possibility to specify the latter approach as a derived operator). An interesting consequence from a software engineering point of view is that in a specification of an agent (process) it is not necessary to know in advance with which other agents synchronisation should take place. Modularity is thus gained.

Our aim is not to present a new programming language but rather to introduce new variants of tell and ask primitives, to justify our proposal via a comparison with [4] (demonstrating expressiveness), and to present a semantics for them. To sum up our approach allows each agent (i) to maintain its local private store, (ii) to share (read/write) global information in its global stores hierarchy and (iii) to communicate (via multi-party or hand-shakes) with other agents. To achieve this we largely employ standard CCP constructs for hiding of local variables ($\exists_X$) and parameter passing (d_{xy}) as well – see Definition 1.

The rest of this paper is organised as follows. In Section 2 we present the syntax and an informal semantics of Scc. To justify our proposal we compare it with the language of [4] in Section 3, while in the Section 4 we give an operational (SOS rules and final result) semantics. We conclude by summarising full semantic treatment of Scc and by suggesting some future research directions.

2 Language Scc

This section presents the syntax and the informal semantics of the language underlying the Scc paradigm, also called Scc. As in [11], the constraint system underlying Scc language consists of any system of partial information that supports the entailment relation. We assume a given cylindric constraint system $\langle C, \vdash \rangle$ over a set of variables *Svar*, defined as usual from a simple constraint system $\langle D, \vdash \rangle$ as follows.

Definition 1. *Let Svar be a denumerable set of variables (denoted by $x, y, \ldots$) and let $\langle D, \vdash \rangle$ be a simple constraint system. Let $\mathcal{P}_F(D)$ denote the set of finite subsets of D. For each variable $x \in$ Svar a function $\exists_x : \mathcal{P}_F(D) \rightarrow \mathcal{P}_F(D)$ is defined such that for any $c, d \in \mathcal{P}_F(D)$ the conditions (E_1) to (E_4) are satified. Moreover, for each $x, y \in$ Svar the elements $d_{xy} \in D$ are* diagonal elements *iff they satisfy the conditions (E_5) to (E_7).*

(E_1) $c \vdash \exists_x(c)$	(E_5) $\emptyset \vdash d_{xx}$
(E_2) $c \vdash d$ *implies* $\exists_x(c) \vdash \exists_x(d)$	(E_6) $\{d_{xy}\} \sim \exists z(\{d_{xz}, d_{zy}\})$ *whenever* $x \not\equiv y$
(E_3) $\exists_x(c \wedge \exists_x(d)) \sim \exists_x(c) \wedge \exists_x(d)$	(E_7) $\{d_{xy}\} \wedge \exists x(c \wedge \{d_{xy}\}) \vdash c$
(E_4) $\exists_x(\exists_y(c)) \sim \exists_y(\exists_x(c))$	

Then $\langle \mathcal{P}(D)_{/\sim}, \vdash \rangle$ is a cylindric constraint system *(over Svar). We denote $\exists_x(c)$ by $\exists_x c$, and for a set $X = \{x_1, \ldots, x_n\}$, we denote $\exists_{x_1} \ldots \exists_{x_n} c$ by $\exists_X c$.*

The language description is parametric with respect to $\langle C, \vdash \rangle$, and so are the semantic constructions presented.

We use G possibly subscripted to range over the set *Sgoal* (processes), $c, d, \ldots$ to range over basic constraints (i.e. constraints which are equivalent to a finite set of primitive constraints), and $X, Y, \ldots$ to range over subsets of *Svar*.

Processes $G \in$ *Sgoal* are defined by the following grammar

$$G ::= \Delta \mid \texttt{ask}(c) \mid \texttt{tell}(c) \mid \texttt{ftell}(c) \mid G ; G \mid G + G \mid G \parallel G \mid \exists_X G \mid p(\boldsymbol{t})$$

Let us briefly discuss an informal meaning of our language constructs. A constant Δ denotes a successfully terminated process. The atomic constructs `ask(c)` and `tell(c)` act on a given store in the following way: as usual, given a constraint `c`, the process `ask(c)` succeeds if `c` is entailed by the store, otherwise it is suspended until it can succeed. However, the process `tell(c)`, of a more lazy nature than the classical one, succeeds only if `c` is (already) entailed by the store and in this case it does not modify the store, and suspends otherwise. It is resumed by a concurrently suspended `ask(d)` operation provided that the conjunction of `c` and of the store entails `d`. In that case, both the `tell` and the `ask` are *synchronously* resumed and the store is atomically augmented with the constraint `c` at the same time. The atomic construct `ftell`(c) behaves as `tell` with the exception that the store is not augmented with the constraint `c`.

The sequential composition $G_1 ; G_2$ and the nondeterministic choice $G_1 + G_2$ have standard meanings (the latter being a *global* as the selection of a component can be influenced by the store and by the environment of the process as well).

The parallel composition $G_1 \parallel G_2$ represents both the interleaving (merge) of the computation steps of the components involved (provided they can perform these steps independently of each other) and also synchronisation: this is the case of the `tell`, `ftell` and `ask` described above. Note that in the general case there can be a parallel composition of a finite sets of `tell`'s and `ftell`'s and a finite set of `ask`'s such that store and a conjunction of `tell` and `ftell` constraints entails `ask` constraints. In this case all the components synchronise. Sometimes this is called a *multi-party* synchronous communication.

The block construct $\exists_X G$ behaves like a process G with the variables in X considered as local. It hides the information about variables from X within the process G. Finally, $p(\boldsymbol{t})$ is a procedure call, where p is the name of a procedure and $\boldsymbol{t}$ is a list of actual parameters. Its meaning is given w.r.t. a set of procedure declarations or *program*; each such a declaration is a construct of the form $p(x_1, \dots, x_n)\ :-\ G$, where $x_1, \dots, x_n$ are distinct variables and G is a goal.

Finally, we note it is quite easy to recover the traditional concurrent constraint paradigm within our framework by the introduction of an asynchronous tell. This can be specified by providing, for each constraint to be told, a concurrent corresponding ask operation. Hence this derived operator atell (standing for an asynchronous tell) can be defined as

$$\texttt{atell}(c) : -\texttt{tell}(c) \parallel \texttt{ask}(c).$$

Note the simulation of our primitives by the old ones is not so straightforward and involve auxiliary tells and asks as well as the coding of a manager.

3 Specification of Multi-Agent Systems in Scc

In [4] a multi-agent programming language (we will refer to it in this paper as MAL) has been introduced. In this section we show how the exchange of information in multi-agent systems can be defined in Scc. To this end we represent expressions from MAL in the framework of Scc. We want to justify that our language Scc can be seen as a formal multi-agent programming language as well. Furthermore, we show that some aspects of behavior of multi-agent systems which cannot be covered by MAL have their (simple) specifications in Scc. The definition of the language MAL is taken from [4].

In the following definitions we assume a given set *Chan* of communication channels, with typical elements α, and a set *Proc* of procedure identifiers, with typical elements p. We also suppose that the set of variables *Svar* is divided into two disjoint subsets $\mathit{Svar}=\mathit{ChanVar}\cup\mathit{AgentVar}$. Typical elements of *ChanVar* are w, typical elements of *AgentVar* are x, y. The variables from *ChanVar* will be used to model communication via channels, while *AgentVar* is the set of agent's variables. We also suppose that the agent's variables are split into *local* and *global* ones. This is because in *MAL* there is a *global* store that is distributed over the agents. Each agent has direct access only to its *private* store. Information in the private store about the *global* variables can be communicated to the other agents. The *local* variables of an agent cannot be referred to in communications.

Definition 2 (Basic actions). *Given a cylindrical constraint system $\langle C, \vdash \rangle$ the basic actions of the programming language MAL are defined as follows:*

$$a ::= \alpha!c \mid \alpha?c \mid ask(c) \mid tell(c)$$

The execution of the output action $\alpha!c$ consists of sending the information c along the channel α, which has to synchronize with a corresponding input $\alpha?d$, for some d with $c \vdash d$. In other words, the information c can be sent along a channel α only if some information entailed by c is requested. The execution of an input action $\alpha?d$, which consists of receiving the information c along the channel α, also has to synchronize with a corresponding output $\alpha!c$, for some c with $c \vdash d$. The execution of a basic action $ask(c)$ by an agent consists of checking whether the private store of the agent entails c. On the other hand, the execution of $tell(c)$ consist of adding c to the private store.

Representing basic actions in Scc
With each channel α we associate a variable w_α from *ChanVar*. The actions $ask(c)$ and $tell(c)$ behave equally in both languages, hence are represented by the same expressions. Sending information along a channel α is modeled by the Scc action $\texttt{ftell}(w_\alpha = \mathit{true} \wedge c)$. This action has to synchronize with the corresponding `ask` action. As `ftell` does not update information on the store, the corresponding `ask` must be sequentially followed by an asynchronous `tell` action which will actually store the information. Hence, receiving of information is modeled as a sequence $\texttt{ask}(w_\alpha = \mathit{true} \wedge c); \texttt{atell}(c)$. The representation of MAL basic actions in the `Scc` is summarized in the following table.

MAL	Scc
$ask(c)$	$\texttt{ask}(c)$
$tell(c)$	$\texttt{tell}(c)$
$\alpha!c$	$\texttt{ftell}(w_\alpha = \mathit{true} \wedge c)$
$\alpha?c$	$\texttt{ask}(w_\alpha = \mathit{true} \wedge c); \texttt{atell}(c)$

Definition 3 (Statements). *MAL agents (statement S) are defined as:*

$$S ::= a.S \mid S_1 + S_2 \mid S_1 \& S_2 \mid \exists_x S \mid p(\overline{x})$$

Statements are thus built up from the basic actions using the following standard programming constructs: action prefixing (denoted by "."), non-deterministic choice (denoted by "+"), internal parallelism (denoted by "&"), local variables (denoted by $\exists_x S$, which indicates that x is a local variable in S), and (recursive) procedure calls of the form $p(\bar{x})$, where $\bar{x}$ denotes a sequence of variables which constitute the actual parameters of the call.

Representing statements in Scc
With the exeption of prefixing, all the statements are directly represented by the corresponding Scc expressions. Prefixing is modeled by sequential composition.

MAL	Scc
$a.S$	$a;S$
$S_1 + S_2$	$S_1 + S_2$
$S_1 \& S_2$	$S_1 \parallel S_2$
$\exists_x S$	$\exists_x S$
$p(\overline{x})$	$p(\overline{x})$

Definition 4 (Multi-agent systems). *A multi-agent system A of MAL is as*

$$A ::= < D, S, \sigma > \mid A_1 \parallel A_2 \mid \delta_H(A)$$

A basic agent in a multi-agent system is represented by a tuple $< D, S, c >$ consisting of a set D of procedure declarations of the form $p(\bar{x}) :- S$, where $\bar{x}$ denote the formal parameters of p and S denotes its body. The statement S in $< D, S, c >$ describes the behavior of the agent with respect to its private store c. The threads of S, i.e. the concurrently executing sub-statements of S, interact with each other via the private store of the basic agent by means of the actions $ask(d)$ and $tell(d)$. Additionally, a multi-agent system itself consists of a collection of concurrently operating agents that interact with each other only via a synchronous information-passing mechanism by means of the communication actions $\alpha!d$ and $\alpha?d$. (In [4] authors provide the parallel composition of agent systems only; the semantic treatment of sequential and the non-deterministic composition of agent systems is standard.)

Representing multi-agent systems in Scc
The parallel operator $\parallel$ is represented as the asynchronous parallel operator $\parallel$ of Scc. The operator $\delta_H(A)$ is represented as $\exists_{w_H} A$. The encapsulation is thus achieved by making the channels from H local. The communication among concurrently operating agents is achieved by *synchronous* communication mechanism of Scc. In particular, the pair `ask`, `ftell` allows to synchronously communicate information between two agents without storing information on the global store. On the other hand the pair `ask`, `tell` allows for multi-agent communication among several process and with storing the communicated information. We summarize the translation in the following table.

MAL	Scc
$< D, S, c >$	Scc program
$A_1 \parallel A_2$	$A_1 \parallel A_2$
$\delta_H(A)$	$\exists_{w_H}(A)$

Global Multi-Agent Communication

In contrast to *MAL* our `Scc` language allows asynchronous and synchronous multi-agent communication. Besides a synchronous `ftell` action, a `Scc` agent can also perform `tell`, `ftell` and `ask` actions on the *global* store. If an agent uses `atell` then it just communicates some piece of information to all processes, i.e. it makes information generally accessible. If an agent uses a synchronous `tell` action on the global store, then there must be at least one agent waiting for this information and the communication is synchronous in this case. However, as information is stored into the global store in this case it would be accessible to any agent. This more general way of transmitting information among agents, makes `Scc` more general and more flexible language for specification and implementation of the exchange of information in multi-agent systems as is *MAL*.

4 Operational semantics $\mathcal{O}$ of `Scc`

Contexts
It turns out that it is possible to treat the sequential and parallel composition operators of `Scc` in a very similar way by introducing the auxiliary notion of *context*. Basically, a context consists of a partially ordered structure where place holders (subsequently referred to by $\Box$) have been inserted at a top-level place, i.e. a place not constrained by the previous execution of other atoms. Viewing goals as partially ordered structures too, the ask and tell primitives to be reduced are those which can be substituted by a place holder $\Box$ in a context. Furthermore, the goals resulting from the reductions are essentially obtained by substituting the place holder by the corresponding clause bodies or the Δ, depending upon whether an atom or a ask/tell primitive is considered.

Definition 5. *Contexts are functions inductively defined on goals as follows:*

1. *A nullary context is associated with any goal. It is represented by the goal and is defined as the constant mapping from Sgoal to this goal with the goal as value.*
2. $\Box$ *is a unary context that maps any goal to itself. For any goal G, this application is subsequently referred to as $\Box[G]$. Thus $\Box[G] = G$ for any goal.*
3. *If tc is an n-ary context and if G is a goal, then $(tc; G)$ is an n-ary context. Its application is defined as follows : for any goals $G_1, \ldots, G_n$,*

$$(tc; G)[G_1, \cdots, G_n] = (tc[G_1, \cdots, G_n]; G)$$

4. *If tc_1 and tc_2 are m-ary and n-ary contexts then $tc_1 \parallel tc_2$ is an (m+n)-ary context. Its application is defined as follows: for any goals $G_1, \ldots, G_{m+n}$,*

$$(tc_1 \parallel tc_2)[G_1, \cdots, G_{m+n}] = (tc_1[G_1, \cdots, G_m]) \parallel (tc_2[G_{m+1}, \cdots, G_{m+n}])$$

In what follows the goals are considered modulo syntactical congruence induced by associativity of ";", "||" and "+", by commutativity of "||" and "+", and Δ as the unit element. Also we will simplify the goals resulting from the application of contexts accordingly.

Transition system
The operational semantics of `Scc` is defined in Plotkin's style ([8]) by means of a transition system, which is itself defined by rules of the form

$$\frac{Assumptions}{Conclusion} \quad if \quad Conditions$$

where *Assumptions* and *Conditions* may possibly be absent. Configurations traditionally describe the statement to be computed and a state summing up the computations made so far. Rephrased in the Scc context, the configurations to be considered here comprise a goal to be reduced together with a store. In the following definition *Sstore* denotes the set of stores.

Definition 6. *The transition relation* $\rightarrow$ *is defined as the smallest relation of* $(Sgoal \times Sstore) \times (Sgoal \times Sstore)$ *satisfying the rules*[1] *of Figure 1. We write* $<G,\sigma> \rightarrow <G',\sigma'>$ *rather than* $(<G,\sigma>,<G',\sigma'>) \in \rightarrow$.

$$\text{(T)} \quad <tc[sp_1,\cdots,sp_m],\sigma> \rightarrow <tc[\Delta,\cdots,\Delta],\tau>$$

$$\text{if} \left\{ \begin{array}{l} \{sp_1,\cdots,sp_m\} = \{\, ask(a_1),\cdots,ask(a_p), \\ \qquad\qquad\qquad\quad tell(at_1),\cdots,tell(at_q), \\ \qquad\qquad\qquad\quad tell(rt_1),\cdots,tell(rt_r), \\ \qquad\qquad\qquad\quad ftell(af_1),\cdots,ftell(af_s), \\ \qquad\qquad\qquad\quad ftell(rf_1),\cdots,ftell(rf_t) \;\; \}, \\ \sigma \cup \{rt_1,\cdots,rt_r\} \cup \{rf_1,\cdots,rf_t\} \\ \quad \vdash \{a_1,\cdots,a_p\} \cup \{at_1,\cdots,at_q\} \cup \{af_1,\cdots,af_s\}, \\ \text{there is no strict subset } S \text{ of } \{rt_1,\cdots,rt_r\} \cup \{rf_1,\cdots,rf_t\} \\ \text{ such that } \sigma \cup S \vdash \{a_1,\cdots,a_p\} \cup \{at_1,\cdots,at_q\} \cup \{af_1,\cdots,af_s\}, \\ \tau = \sigma \cup \{rt_1,\cdots,rt_r\}, \;\; m > 0 \end{array} \right\}$$

Fig. 1. Scc transition rules for new versions of aks and tells

The Operational Semantics

Rules for the sequential and parallel composition operators are tackled by means of the notion of context within the rule (T).

The rule (T) defines reductions of `tell`, `ftell` and `ask` primitives. The primitives to be reduced, referred to as $sp_1,\ldots,sp_m$, are partitioned in five categories:
(1) the `ask` primitives (the multi-set $\{ask(a_1),\cdots,ask(a_p)\}$),
the `tell` primitives split into (2) those which add information to the store (the multi-set $\{tell(rt_1),\cdots,tell(rt_r)\}$) and (3) those which do not (the multi-set $\{tell(at_1),\cdots,tell(at_q)\}$, i.e. already entailed), and
the fictitious primitives `ftell` split in a similar way into (4) the multi-sets $\{ftell(rf_1),\cdots,ftell(rf_t)\}$ and (5) $\{ftell(af_1),\cdots,ftell(af_s)\}$, respectively.

All these primitives are then simultaneously reduced to the empty goal Δ when information on the current store (σ) together with new information told ($rt_1,\ldots,rt_r,rf_1,\ldots,rf_t$) entails information of the other primitives. The new store consists in this case of the old store enriched by new information told. Note that this rule reflects the laziness feature of our tell primitives.

[1] Please note that due to lack of space we do not give the very standard rules for a nondeterministic choice, hiding and a procedure call in Figure 1

An `ask(c)` primitive for a constraint c entailed by the current store σ can be reduced alone following rule (T) by taking the unary context $\square$, $m=1$, $p=1$, $q=0$, $r=0$, $s=0$, $t=0$. The axiom

$$<\mathtt{ask}(c),\sigma> \rightarrow <\Delta,\sigma> \qquad \text{if} \quad \{\sigma \vdash c\} \tag{1}$$

results from rule (T) as the particular case.

A `tell(c)` primitive for a constraint c entailed by the current store σ can be reduced alone following rule (T) by taking the unary context $\square$, $m=1$, $p=0$, $q=1$, $r=0$, $s=0$, $t=0$. The axiom

$$<\mathtt{tell}(c),\sigma> \rightarrow <\Delta,\sigma> \qquad \text{if} \quad \{\sigma \vdash c\} \tag{2}$$

results from rule (T) as the particular case as well.

Other tell's and ask's need each other for reduction and reduce simultaneously. A minimality condition (see the side condition of (T)) is required to forbid outsider tell's to be reduced by taking advantage of a concurrent reduction.

To define the operational semantics we follow the logic programming tradition – it specifies the final store of the successful computations. It also indicates those stores corresponding to deadlock situations and distinguishes between two types: *failure* corresponding to the absence of suitable procedure declarations to reduce procedure calls and *suspension* corresponding to the absence of suitable data on the store or of concurrent processes that would allow tell and ask primitives to proceed, i.e. to suspended tell's and ask's. Note that, as illustrated by axioms (1) and (2) above, the two situations may be distinguished by a simple criterion: the existence of a store richer than the current one that would enable the computation to proceed. The following definition is based on this intuition. The symbols δ^+, δ^-, and δ^s are used to indicate the computations ending by a success, a failure, and a suspension, respectively.

Definition 7. *Operational semantics* $\mathcal{O} : Sgoal \rightarrow \mathcal{P}(Sstore \times \{\delta^+, \delta^s, \delta^-\})$ *is defined as the following function: for any goal* G,

$$\begin{aligned}
\mathcal{O}(G) = &\ \{ <\tau,\delta^+> : <G,true> \rightarrow \cdots \rightarrow <\Delta,\tau> \} \\
\cup &\ \{ <\tau,\delta^s> : <G,true> \rightarrow \cdots \rightarrow <G',\tau> \not\rightarrow,\ \textit{where } G' \neq \Delta \textit{ and} \\
&\qquad \textit{there are } \sigma', G'', \sigma'' \textit{such that } <G',\sigma'> \rightarrow <G'',\sigma''>\} \\
\cup &\ \{ <\tau,\delta^-> : <G,true> \rightarrow \cdots \rightarrow <G',\tau> \not\rightarrow,\ \textit{where } G' \neq \Delta \textit{ and} \\
&\qquad \textit{for any } \sigma', <G',\sigma'> \not\rightarrow \ \}
\end{aligned}$$

5 Conclusions

We have presented a language for a *specification* of the exchange and/or the global sharing of information in multi-agent systems. It is solely based on concurrent constraint programming paradigm with slightly modified test and updates operations. We have briefly compared it to the latest proposal within this area as given in [4].

Due to a lack of space we have presented an operational semantics only. Reader can easily verify it is not compositional. We refer to our studies in [2], where a compositional semantics is given and proved to be correct with respect to the semantics $\mathcal{O}$ (it is based on 'hypothetical' steps which can be made by both concurrent agents and the state of global store rather than successive updates). In [3] an algebraic (failure) semantics is defined for a subset of `Scc` (only finite behaviours and without `ftells`). The algebraic semantics is proved to be sound and complete with respect to a compositional operational semantics. Allowing handshake communications only in the mentioned subset, we proposed [1] a denotational semantics. We employed so-called testing techniques – this semantics uses monotonic sequences of labelled pairs of input-output states, possibly containing "hypothetical" gaps, and ending with marks reporting success or failure (to follow logic programming tradition). This semantics is proved to be correct with respect to the operational semantics and fully abstract as well.

Our future work aims at designing fully abstract semantics for the full version of `Scc`. Also we study possibilities of incorporating (discrete) real-time aspects.

References

1. L. Brim, D. Gilbert, J-M. Jacquet, and M. Křetínský. A fully abstract semantics for a version of synchronous concurrent constraint programming. Technical Report FIMU-RS-99-08, Faculty of Informatice, MU Brno, 1999.
2. L. Brim, D. Gilbert, J-M. Jacquet, and M. Křetínský. New Versions of Ask and Tell for Synchronous Communication in Concurrent Constraint Programming. Technical report, TCU/CS/1996/03, City University London, ISSN1364-4009, 1996.
3. L. Brim, D. Gilbert, J-M. Jacquet, and M. Křetínský. A Process Algebra for Synchronous Concurrent Constraint Programming. In *Proceedings of ALP96*, LNCS, pages 24–37. Springer-Verlag, 1996.
4. F. de Boer, R. van Eijk, M. van der Hoek, and Ch. Meyer. Failure semantics for the exchange of information in multi-agent systems. In *CONCUR: 11th International Conference on Concurrency Theory.* LNCS, Springer-Verlag, 2000.
5. Frank S. de Boer, Joost N. Kok, Catuscia Palamidessi, and Jan J. M. M. Rutten. Non-monotonic concurrent constraint programming. In Dale Miller, editor, *Logic Programming - Proceedings of the 1993 International Symposium*, pages 315–334, Vancouver, Canada, 1993. The MIT Press.
6. M. Falaschi, G. Levi, and Catuscia Palamidessi. A Synchronization Logic: Axiomatics and Formal Semantics of Generalized Horn Clauses. *Information and Control*, 60:36–69, 1994.
7. T. Finin, D. McKay, R. Fritzson, and R. McEntire. KQML: An information and knowledge exchange protocol. In *Knowledge Building and Knowledge Sharing.* Ohmsha and IOS Press, 1994.
8. G. Plotkin. A structured approach to operational semantics. Technical report, Tech.Rep. DAIMI FN-19, Computer Science Dept., Aarhus University, 1981.
9. J-H. Rety. *Langages concurrents avec constraintes, communication par messages at distribution.* Phd thesis, University of Orleans, 1997.
10. Vijay Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
11. Vijay Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constrant programming. In *Proc. of the 18th POPL.* ACM, 1991.
12. M. Wooldridge. Verifiable semantics for agent communication languages. In *ICMAS'98.* IEEE Computer Press, 1998.