Concurrent Constraint Programming with Process Mobility

David Gilbert[1] and Catuscia Palamidessi[2]

[1] Department of Computing, City University drg@soi.city.ac.uk
[2] Department of Computer Science and Engineering, Penn State University catuscia@cse.psu.edu

Abstract. We propose an extension of concurrent constraint programming with primitives for process migration within a hierarchical network, and we study its semantics.
To this purpose, we first investigate a "pure" paradigm for process migration, namely a paradigm where the only actions are those dealing with transmissions of processes. Our goal is to give a structural definition of the semantics of migration; namely, we want to describe the behaviour of the system, during the transmission of a process, in terms of the behaviour of the components. We achieve this goal by using a labeled transition system where the effects of sending a process, and requesting a process, are modeled by symmetric rules (similar to handshaking-rules for synchronous communication) between the two partner nodes in the network.
Next, we extend our paradigm with the primitives of concurrent constraint programming, and we show how to enrich the semantics to cope with the notions of environment and constraint store.
Finally, we show how the operational semantics can be used to define an interpreter for the basic calculus.

1 Introduction

Concurrent constraint programming (ccp) [16] is a computational paradigm which combines the notions of concurrency and constraints. Classical ccp is based on a shared (constraint) store and, as such, it implies a centralized computational model.

In this work, we aim at enriching the ccp paradigm with the notion of localities, local stores and environments, and process migration. More precisely, we consider a distributed version of ccp where processes (or agents) run at specific sites, and have associated a local environment of procedure declarations, and a local store of constraints. The sites are organized hierarchically, and therefore an agent may contain sub-agents. The computation of a process only depends on its local code and data; however, a crucial characteristic that we wish to describe is the ability of an agent to *move* from site to site in the network, and bring along its environment and store.

Our main goal is to provide a Structural Operational Semantics for such an extension of ccp, namely a semantics in which the behaviour of complex processes is defined in terms of the behaviour of their components. This results in the usual advantages for reasoning and for the definition of formal tools. In the long-term our motivation is to be able to describe and reason about the migration of *software agents* in a distributed system.

1.1 Process migration versus link mobility

The term "mobility" has become associated with two meanings – firstly that of reconfiguring a network by changing the links or connections between nodes, and secondly the ability of a node within a network to migrate its position, thus also reconfiguring the topology of the network. In order to avoid confusion, we use *link mobility* to describe the former, and *process mobility*, or *migration*, for the latter. In this work, we are concerned with process mobility.

The classical work on link mobility is Milner's π-*calculus* [12]. Migration has been described by Cardelli [3, 7], and formalized in work on *agent-passing* calculi, for example *Plain CHOCS*[17] and *Strictly-Higher-Order* π-*calculus* [14]. For a study of the correspondence between the two concepts, see Sangiorgi [15].

An important consideration in migration is that of locality, namely the explicit association between agents and specific sites. Several calculi supporting this notion have been presented recently; see for instance [5, 6, 10]. Of these, however, only Fournet et al's Distributed Join Calculus [10] treats locality in combination with migration. This is done in style of the Chemical Abstract Machine, by creating a flat model of local solutions with associated local names, and organising them as an implicit tree of nested locations. In contrast with [10], we describe migration in the SOS style, maintaining the network structure explicitly as it is done in [5]. Another difference with [10] is that we are able to describe migration to a sublocation, while this is not possible in [10].

1.2 Models of mobile computation

One can distinguish various types of mobile computation, which depend on the way the environment is treated under migration.

Following Cardelli [7], we regard a *closure* as the run-time description of a running procedure, i.e. the code plus the context of its execution. In general this context may include data, active network connections which are preserved on transmission, and new connections that are created to keep the closure in touch with the site that it has left behind.

With respect to the notion of closure, we can distinguish three increasingly richer models of mobility:

1. Code mobility only.
2. Mobility of *agents*, which are closures with contexts which lack link information. These agents do not communicate remotely with other agents, but move to some location and communicate locally there.
3. Mobility of general closures which include network connections (links), like in Obliq [8].

In this paper we focus on the agent mobility only. At the end of Section 3 we discuss possible extensions towards the last, most general model.

1.3 Distributed concurrent constraint programming

To our knowledge, there have been only two previous proposals for distributed extensions of ccp: Distributed Oz [18] and Distributed ccp [13].

The proposal in [13] is based on the notion of agents computing within their local stores of constraints, and exchanging constraint abstractions through channels. A process receiving an abstraction applies it to its local variables, thus making a sort of local version of the received constraint. The dependency on global information is avoided by a static analysis of the program, giving the sufficient conditions under which the store of two agents can be divided in two local (independent) stores.

In [18] the notion of global and local information coexist: the computation of an agent mainly depend on local data, but the bindings on the shared logical variables are global and require handling by a distributed constraint solving algorithm. The main kind of mobility is cell mobility, namely the information content of a cell (a sort of imperative variable) can be exchanged between agents.

Neither [18] nor [13] deal with distribution and agent migration in our sense, i.e. by using an explicit notion of site, in a network organized hierarchically, and by transferring environment and store along with the code.

1.4 Structure of the paper

The next section presents an abstract paradigm for the description of process migration between any two sites within a hierarchical network. Section 3 shows how the paradigm can be enriched to cope with the concepts of environment and constraint store, thus laying the foundations of concurrent constraint programming with process migration. Section 4 presents a simple (centralized) interpreter for the paradigm described in Section 3, and Section 5 discusses future work.

2 The basic paradigm for migration

In this section we present our methodology for describing migrating agents within a hierarchically organized network. Our basic assumption is that the topology of such a network can be described as a tree, where each node is associated with a name n and contains an agent A. Names are unique only amongst peer nodes (sharing the same parent), and the unique address (*location*) of a node is given by the string π formed by concatenating the names of the nodes on the direct path from the root to that node. Thus we permit the same name to be used more than once in a system, and our calculus ensures that no ambiguity concerning addresses can raise when an agent migrates within the network.

An agent A in a node n can migrate to any other node m in the network. In this migration A is relocated together with all its subnodes and is inserted in m together with the agent B of m. The structure of the network can change as a result of this migration, for instance when a process which contains nested nodes migrates to a leaf node.

We assume two basic actions for migration: *go* and *fetch*. The first sends an agent to a node n at a specified location; the second gets a copy of an agent from a node n at a specified location, leaving the agent available for another request. We think that this naturally formulates "go" instructions and "fetch" requests. In both cases we specify the location by giving the path to n starting from the first (i.e. lowest in the tree) common ancestor of n and the node m which is performing the action. We will call this path the *relative address* from the point of view of m, and the *sub-address* from the point of view of the ancestor.

The syntax of our basic calculus is specified by the following grammar, where the symbol $\|$ represents the usual parallel operator and $\mathbf{0}$ represents inaction:

$$\mathit{Agents}\ A ::= \mathbf{0} \mid \mathit{node}(n, A) \mid \mathit{go}(\pi, A) \mid \mathit{fetch}(\pi) \mid A \parallel A$$

We assume the usual structural equivalences for the parallel operator:

$$\begin{gathered} A \parallel \mathbf{0} \equiv A \\ A_1 \parallel A_2 \equiv A_2 \parallel A_1 \\ (A_1 \parallel A_2) \parallel A_3 \equiv A_1 \parallel (A_2 \parallel A_3) \end{gathered}$$

The operational semantics is defined by a labeled transition systems whose configurations are agents and labels have the following form, where A is an agent:

- $s(\pi_f, \pi_t, A)$: send A from sub-address π_f to relative address π_t
- $r(\pi_f, \pi_t, A)$: receive A from relative address π_f to sub-address π_t
- $vs(\pi_f, \pi_t, A)$: virtual send A from sub-address π_f to relative address π_t
- $vr(\pi_f, \pi_t, A)$: virtual receive A from relative address π_f to sub-address π_t
- $as(\pi_f, \pi_t, A)$: actual send A from sub-address π_f to sub-address π_t
- $ar(\pi_f, \pi_t, A)$: actual receive A from sub-address π_f to sub-address π_t
- $\mathit{migrate}(\pi_f, \pi_t, A)$: relocate A from sub-address π_f to sub-address π_t

The last three kinds of labels correspond to transitions that can be performed only by the first common ancestor of the nodes m and n between which the migration takes place. Basically, the idea is the following: when a node m executes an action $go(\pi_t, A)$, it performs a send transition $s(m, \pi_t, A)$. Correspondingly, the node n at the relative address π_t performs a virtual receive transition $vr(\pi_f, n, A)$, where π_f is the relative address of m from the point of view of n. This virtual transition is a "spontaneous initiative", i.e. it is generated by the agent $\mathbf{0}$ (always present in a node because of the equivalence $A \equiv A \parallel \mathbf{0}$).

These transitions propagate upwards in the tree until they hit a common ancestor. At this point the send becomes an actual send, matches with the virtual receive, and the migration takes place.

During the upward propagation of $vr(\pi_f, \pi, A)$ the sub-address π of the virtual receiver is incrementally constructed, until it becomes π_t. Analogously, during the upward propagation of $s(\pi', \pi_t, A)$, the sub-address π' of the sender is constructed, until it becomes π_f. The actual send and the virtual receive can match only if the sub-addresses correspond, i.e. only if they are of the form

ℓ	$Cond$	ℓ'
$s(\pi_f, \pi_t, B)$	$hd(\pi_t) \neq n$	$s(n\pi_f, \pi_t, B)$
$s(\pi_f, \pi_t, B)$	$hd(\pi_t) = n$	$as(n\pi_f, \pi_t, B)$
$r(\pi_f, \pi_t, B)$	$hd(\pi_f) \neq n$	$r(\pi_f, n\pi_t, B)$
$r(\pi_f, \pi_t, B)$	$hd(\pi_f) = n$	$ar(\pi_f, n\pi_t, B)$
$vs(\pi_f, \pi_t, B)$	$-$	$vs(n\pi_f, \pi_t, B)$
$vr(\pi_f, \pi_t, B)$	$-$	$vr(\pi_f, n\pi_t, B)$
$migrate(\pi_f, \pi_t, B)$	$-$	$migrate(n\pi_f, n\pi_t, B)$

Table 1. Specification of labels and conditions for the propagation rule. The function hd gives the first element of a string.

$as(\pi_f, \pi_t, A)$ and $vr(\pi_f, \pi_t, A)$ respectively. Note that, strictly speaking, only one of these constructed address is necessary to test if the two actions match; we do it this way just for the sake of symmetry.

The mechanism for the $fetch(\pi)$ action is analogous: in this case its node will perform a receive transition and the node at the relative address π will perform a corresponding virtual send transition. Note however that *fetch* and *go* are not symmetric to each other: *go* does not cause a duplication of the agent, while *fetch* does.

The above ideas are formalized by the following rules, which specify the transition relation. λ represents the empty string.

The following four axioms introduce the send and receive, and their virtual counterparts.

$$\begin{array}{rrcl}
(send) & go(\pi_t, A) & \stackrel{s(\lambda,\pi_t,A)}{\longrightarrow} & \mathbf{0} \\
(receive) & fetch(\pi_f) & \stackrel{r(\pi_f,\lambda,A)}{\longrightarrow} & A \\
(virtual\ send) & A & \stackrel{vs(\lambda,\pi_t,A)}{\longrightarrow} & A \\
(virtual\ receive) & \mathbf{0} & \stackrel{vr(\pi_f,\lambda,A)}{\longrightarrow} & A
\end{array}$$

The following rule specifies the upwards propagation of transitions in the tree structure:

$$(propagation) \quad \frac{A \stackrel{\ell}{\longrightarrow} A'}{node(n, A) \stackrel{\ell'}{\longrightarrow} node(n, A')} \quad Cond$$

In this rule, ℓ' and the side condition $Cond$ depend on ℓ as specified in Table 1.

The following two symmetric rules describe the actual migration:

$$(migrate_{go}) \; \frac{A_1 \stackrel{as(\pi_f,\pi_t,B)}{\longrightarrow} A_2 \quad A_2 \stackrel{vr(\pi_f,\pi_t,B)}{\longrightarrow} A_3}{A_1 \stackrel{migrate(\pi_f,\pi_t,B)}{\longrightarrow} A_3}$$

$$(migrate_{fetch}) \; \frac{A_1 \stackrel{ar(\pi_f,\pi_t,B)}{\longrightarrow} A_2 \quad A_2 \stackrel{vs(\pi_f,\pi_t,B)}{\longrightarrow} A_3}{A_1 \stackrel{migrate(\pi_f,\pi_t,B)}{\longrightarrow} A_3}$$

Note that, if the two nodes between which the relocation takes place are *not* along the same branch, then one can use more elegant rules for migration, modeling it as *handshaking* between the real and the virtual actions. More formally, the $migrate_{go}$ could be replaced by the following:

$$(migrate'_{go})\ \frac{A_1 \xrightarrow{s(\pi_f, n\pi_t, B)} A'_1 \qquad A_2 \xrightarrow{vr(n\pi_f, \pi_t, B)} A'_2}{node(n, A_1 \parallel A_2) \xrightarrow{migrate(n\pi_f, n\pi_t, B)} node(n, A'_1 \parallel A'_2)}$$

and analogously for the $migrate_{fetch}$.

This rule however does not cover the case of relocation between ancestor and descendant, because that situation cannot be described, in our paradigm, by using the parallel operator.

Finally, the rule for the parallel operator is the standard interleaving rule, refined by a condition intended to maintain the uniqueness of names among sibling nodes:

$$(parallel)\ \frac{A_1 \xrightarrow{\ell} A'_1}{A_1 \parallel A_2 \xrightarrow{\ell} A'_1 \parallel A_2} \qquad names A'_1 \cap names A_2 = \emptyset$$

where the function $names(A)$ gives all the names of top-level nodes in A. Formally:

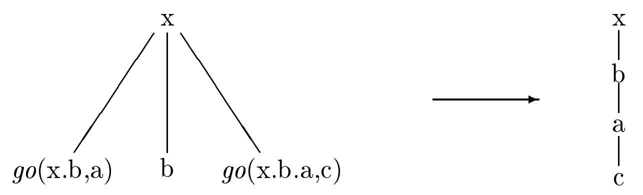
$$\begin{array}{ll} names(\mathbf{0}) & = \emptyset \\ names(node(n, A)) & = \{n\} \\ names(go(\pi, A)) & = \emptyset \\ names(fetch(\pi)) & = \emptyset \\ names(A_1 \parallel A_2) & = names(A_1) \cup names(A_2) \end{array}$$

We conclude this section with some examples illustrating how our model works.

Examples

In the following examples, for the sake of simplicity we omit null agents and represent the agent $node(n, 0)$ by $node(n)$, or (in the figures) by n:

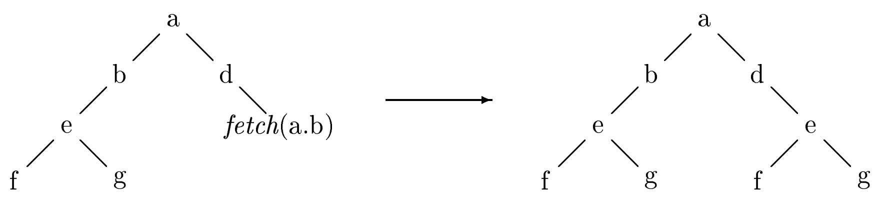
(1) Reorganising a branched network to a linear network



There is only one (strict) order of migrations:

$$node(x, node(x, go(x.b, node(a)) \parallel node(b) \parallel go(x.b.a, node(c))))$$
$$\xrightarrow{migrate(x,x.b,node(a))}$$
$$node(x, node(b, node(a) \parallel go(x.b.a, node(c))))$$
$$\xrightarrow{migrate(x,x.b.a,node(c))}$$
$$node(x, node(b, node(a, node(c))))$$

(2) Using fetch

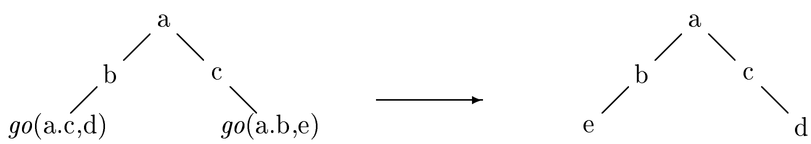


Again the reader can verify that there is only one order for migration commands to be executed

(3) Swapping children nodes using two agents

In this case two different migration histories are possible:

1. $\xrightarrow{migrate(a.b,a.c,node(d))} \ldots \xrightarrow{migrate(a.c,a.b,node(e))}$
2. $\xrightarrow{migrate(a.c,a.b,node(e))} \ldots \xrightarrow{migrate(a.b,a.c,node(d))}$



3 Enhancing ccp with migration

In the previous section we have dealt with the simple case of agents without environment or store. Of course, this is a very simplistic assumption. One of the main issues about migration is the formalization of the way a migrating process is inserted in to the environment of the host, how it interacts with the resources of the host, what are the scoping rules, etc.

In this section we investigate how the basic calculus for migration can be enriched with the notions of environment and constraint store, laying the foundations for concurrent constraint programming with process mobility.

Let us first recall the definition of ccp [16]:

$$\mathit{Agents}\ A ::= \mathbf{0} \mid \mathit{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid p(x) \mid \exists_x A$$

The c and c_i's are *constraints*, i.e. elements of a given constraint system $(C, \vdash)$. We recall that $\vdash$ represents a relation of entailment between elements of C, that C is closed under logical conjunction $\wedge$, and that a cylindrification operator $\exists_x : C \rightarrow C$ is defined for any variable x.

Briefly, the computational meaning of this paradigm is the following: the agents interact via a common *store* which ranges over C. The execution of $\mathit{tell}(c)$ adds c to the current store, i.e. if the current store is s then the resulting store is $s \wedge c$. The *guarded choice* agent $\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i$ selects nondeterministically one j such that $\mathit{ask}(c_j)$ is enabled in the current store s, i.e. $s \vdash c_j$, and then behaves like A_j. The agent $\exists_x A$ behaves like A, with x considered *local* to A. Finally, the agent $p(x)$ is a procedure call. Its meaning is given by a declaration of the form $p(y)$:- A.

In this presentation, taken from [16], there is a unique global set of declarations. Furthermore, although in the course of the computation some agents might obtain a local store, initially there is only a unique global store (this assumption makes it easier to describe the semantics). Since our purpose here is to study agent migration in the presence of a structure of environments and stores, we will enrich this paradigm with the possibility of associating local declarations and a local store with an agent (besides a local variable). More precisely, we will substitute the hiding construct $\exists_x A$ with the more general block construct:

$$\mathit{block}(D, X, s, A)$$

where D is a (possibly empty) set of local procedure declarations, X is a (possibly empty) set of local variables, and s is the initial (possibly empty) local store.

Thus the syntax of this extended ccp, enhanced with the migration constructs, will be:

$$\begin{aligned}\mathit{Agents}\ A ::= {} & \mathbf{0} \mid \mathit{tell}(c) \mid \textstyle\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i \mid A \parallel A \mid p(x) \mid \\ & \mathit{block}(D, X, s, A) \mid \mathit{node}(n, A) \mid \mathit{go}(\pi, A) \mid \mathit{fetch}(\pi)\end{aligned}$$

The operational semantics is defined via a labeled transition system as follows: the basic configurations are the blocks, the labels are only those introduced in Section 2, plus τ, which will label the transitions corresponding to the standard (unlabeled) ccp transitions. The transition rule for tell is similar to the one for standard ccp:

$$\mathit{block}(D, X, s, \mathit{tell}(c)) \xrightarrow{\tau} \mathit{block}(D, X, s \sqcup c, \mathbf{0})$$

The symbol $\sqcup$ here represents concatenation, and will be interpreted as logical conjunction when the store is checked for entailment. In [16] the corresponding rule uses logical conjunction directly. We need to distinguish the contribution made by an agent essentially to deal with the presence of an initial local store. This will become apparent in the rule for nested blocks.

The guarded choice rule is just the same as in standard ccp.

$$\mathit{block}(D, X, s, \textstyle\sum_{i=1}^{n} \mathit{ask}(c_i) \rightarrow A_i) \xrightarrow{\tau} \mathit{block}(D, X, s, A_j) \quad s \vdash c_j$$

For the parallel operator, we have to add the condition on uniqueness of sibling names. The function *names* extends to ccp in the obvious way (for the procedure call it gives the empty set and for the choice it gives the union of the names of all branches).

$$\frac{\mathit{block}(D, X, s, A_1) \xrightarrow{\ell} \mathit{block}(D, X, s', A_1')}{\mathit{block}(D, X, s, A_1 \parallel A_2) \xrightarrow{\ell} \mathit{block}(D, X, s', A_1' \parallel A_2)} \quad \mathit{names}(A_1') \cap \mathit{names}(A_2) = \emptyset$$

The procedure call is just the same as in standard ccp. In this rule, Δ_y^x is an elegant mechanism which links the formal and the actual parameter, and avoids clashes with other variable names in the network. See [16] for details. In our case, we will have to enrich it so that it also avoids clashes with sibling node names

$$\mathit{block}(D, X, s, p(x)) \xrightarrow{\tau} \mathit{block}(D, X, s, \Delta_y^x(A)) \quad p(y) \text{ :- } A \in D$$

The rule for the block construct enriches the rule for hiding in [16] with the treatment of definitions in nested blocks, and with the distinction of the agent's contribution to the store, which is necessary for coping with the possibility of an initial (non empty) local store.

$$\frac{\begin{array}{c}\mathit{block}(D_1 \lhd D_2, X_2, (\exists_{X_2} s_1) \sqcup s_2, A) \\ \xrightarrow{\ell} \\ \mathit{block}(D_1 \lhd D_2, X_2, (\exists_{X_2} s_1) \sqcup s_3, A)\end{array}}{\begin{array}{c}\mathit{block}(D_1, X_1, s_1 \sqcup \exists_{X_2} s_2, \mathit{block}(D_2, X_2, s_2, A)) \\ \xrightarrow{\ell} \\ \mathit{block}(D_1, X_1, s_1 \sqcup \exists_{X_2} s_3, \mathit{block}(D_2, X_2, s_3, A))\end{array}}$$

Here, $D_1 \lhd D_2$ represents the hierarchical union of D_1 and D_2, i.e. in case p is defined both in D_1 and in D_2, the declarations for p in D_2 override those in D_1.

The intuition behind the above rule is the following: In the internal block, the procedure declarations D_1 of the external block are visible, except for those which are "shadowed" by local declarations of the same procedure name (standard rule of scoping). The external store (s_1) is also entirely visible, except for the constraints involving variables with the same name as the local ones (X_2). The information about the shadowed external variables (X_2) is be filtered away by using the cylindrification operator $\exists_{X_2}$. Conversely, in the external block the information produced in the internal block (s_2 and s_3) is entirely visible, except for the constraints involving the local variables. Again, this information is filtered away by using $\exists_{X_2}$. This way of treating the store is inspired by [16].

The rule for the node expresses that the environment of an agent in a node is the same as the environment of the node[1]:

$$\frac{node(n, block(D, X, s, A)) \stackrel{\ell}{\longrightarrow} node(n, block(D, X, s', A'))}{block(D, X, s, node(n, A)) \stackrel{\ell}{\longrightarrow} block(D, X, s', node(n, A'))}$$

Note that the premise of this rule is a transition between node agents. These will be considered auxiliary configurations and the rules for their transitions are the rules *propagation*, $migrate_{go}$ and $migrate_{fetch}$ of Section 2. The rule *parallel* is not needed.

Finally we have to adapt the rules *send*, *receive*, and their virtual counterparts. The following definitions formalize migration with dynamic scope, i.e. when a migrating agent brings with it only its internal environment, not its external one:

$$\begin{array}{rcl} block(D, X, s, go(\pi_t, A)) & \stackrel{s(\lambda,\pi_t,A)}{\longrightarrow} & block(D, X, s, \mathbf{0}) \\ block(D, X, s, fetch(\pi)) & \stackrel{r(\pi_f,\lambda,A)}{\longrightarrow} & block(D, X, s, A) \\ block(D, X, s, A) & \stackrel{vs(\lambda,\pi_t,A)}{\longrightarrow} & block(D, X, s, A) \\ block(D, X, s, \mathbf{0}) & \stackrel{vr(\pi_f,\lambda,A)}{\longrightarrow} & block(D, X, s, A) \end{array}$$

Note that we could model a more lexical kind of scoping rule by modifying the label of the send and the receive actions. For instance, the send rule would be written as

$$block(D, X, s, go(\pi_t, A)) \stackrel{s(\lambda,\pi_t,block(D,X,s,A))}{\longrightarrow} block(D, X, s, \mathbf{0})$$

In this way we export also the local environment and the store of the father. However note that this is a mixture of dynamic and lexical scope: to represent a purely lexical scoping rule, we would need closures.

3.1 An example

We illustrate now our extension of ccp with an example. We assume dynamic scope, although in this example it does not really matter.

Assume that a seller, at address *root.a*, is willing to sell a certain good to the best offerer, by auction. Three potential buyers, at nodes *root.b*, *root.c*, and

[1] We could have simplified the syntax and the semantics by unifying the concept of node and block, i.e. we could have considered only one construct containing a node name, local declarations, local variables, local store and an agent. The reason why we did not do this is because we think of a node as a physical site which can host many parallel agents, each one with its own environment.

$root.d$ respectively, are willing to buy the product, but are too busy to participate directly in the auction process. Instead, they send an agent to the site where the auction takes place. The agent will have certain parameters specified, like the increment for raising the bidding each time, and the maximum price the buyer is willing to pay. At the end, the auctioneer will send an agent back to each buyer to tell whether he has won the bidding or not.

The following process represents the auctioneer. For simplicity, we assume a very simple kind of auction, with only one round: all the offers are collected, compared, and the best one wins. We use $ask_X(c) \to A$ to represent the agent $ask(\exists_X c) \to tell(c) \parallel A$.

$$
\begin{array}{l}
node(root.a,\\
\quad block(\emptyset, \{pb, pc, pd\}, \emptyset,\\
\quad\quad ask(offer(b, pb) \wedge offer(c, pc) \wedge offer(d, pd)) \to\\
\quad\quad\quad ask_{pb,pc,pd}(pb \geq pc \wedge pb \geq pd) \to\\
\quad\quad\quad\quad go(root.b, tell(winner(yes)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.c, tell(winner(no)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.d, tell(winner(no)))\\
\quad\quad\quad +\\
\quad\quad\quad ask_{pb,pc,pd}(pc \geq pb \wedge pc \geq pd) \to\\
\quad\quad\quad\quad go(root.b, tell(winner(no)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.c, tell(winner(yes)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.d, tell(winner(no)))\\
\\
\quad\quad\quad +\\
\quad\quad\quad ask_{pb,pc,pd}(pd \geq pb \wedge pd \geq pc) \to\\
\quad\quad\quad\quad go(root.b, tell(winner(no)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.c, tell(winner(no)))\\
\quad\quad\quad\quad \parallel\\
\quad\quad\quad\quad go(root.d, tell(winner(yes)))\ ))
\end{array}
$$

The following process represents the potential buyer at site $root.b$. The other buyers are similar, except possibly for the price offered (100) and the continuation process (A).

$$
\begin{array}{l}
node(root.b,\\
\quad block(\emptyset, \{price, answer\}, \{price = 100\},\\
\quad\quad go(root.a, offer(b, price)) \quad \parallel \quad ask(winner(answer) \to A\ )
\end{array}
$$

Note that, thanks to the mobility of the store, the information can be transmitted from the buyer to the auctioneer and viceversa. Thanks to the locality of the stores, there is no need of distributed constraint solving, and we can also ensure a certain privacy of the information; for instance, the winner's identity will not be available to the other buyers.

4 Interpreter

We have implemented an interpreter in SICStus Prolog based on the operation semantics defined in previous sections; the interpreter can be obtained over the Web at `www.soi.city.ac.uk/~drg/migration`. The software has been used as part of an undergraduate module on Software Agents given to final year Computing and Software Engineering students at City University.

Our implementation technique involves representing a transition rule in the form:

$$\frac{A1 \xrightarrow{l_1} A'_1 \; \ldots \; A_n \xrightarrow{l_n} A'_n}{A_1 \xrightarrow{l} A'_n} \quad Condition$$

by the Prolog clause

```
trans(A1,Label,name(Label,ObsA1,...,ObsAn),An'):-
   trans(A1,L1,ObsA1,A1'), ..., trans(An,Ln,ObsAn,An'),
   Condition.
```

Thus, for instance, the axiom

$$go(\pi_t, A) \stackrel{s(\lambda,\pi_t,A)}{\longrightarrow} \mathbf{0}$$

is represented by the unit clause

```
trans(go(To,A), s([],To,A), send(s([],To,A)),0).
```

and the rule

$$\frac{A_1 \stackrel{as(\pi_f,\pi_t,B)}{\longrightarrow} A_2 \qquad A_2 \stackrel{vr(\pi_f,\pi_t,B)}{\longrightarrow} A_3}{A_1 \stackrel{migrate(\pi_f,\pi_t,B)}{\longrightarrow} A_3}$$

is represented by the clause

```
trans(A1, migrate(Fr,To,B), migrate_go(migrate(Fr,To,B),OA,OB), A3):-
        trans(A1, as(Fr,To,B), OA,A2),
        trans(A2, vr(Fr,To,B), OB,A3).
```

Users can input an agent description as a Prolog term at the prompt; the interpreter will process this term and output a trace of

$agent_0 \; migration\_action_1 \; agent_1 \; \ldots migration\_action_n \; agent_n$

and will offer to display alternative traces and final states (if these exist). The final state of the agent is also reported, which can be either *inactive* (contains no migration instructions) or *stuck* (contains migration instructions which cannot be processed, for example references to addresses which do not exist).

For instance, Example 3 of Section 2 is represented by the term

```
node(a,node(b,go([a,c],node(d,0)))//node(c,go([a,b],node(e,0))))
```

where the symbol"//" represents parallel composition.

If we give this term to the prompt, the interpreter responds in the following way:

```
History:
 Scene: 1 node(a,node(b,go([a,c],node(d,0)))//node(c,go([a,b],node(e,0))))
 Move:  2 migrate([a,b],[a,c],node(d,0))
 Scene: 3 node(a,node(b,0)//node(c,node(d,0)//go([a,b],node(e,0))))
 Move:  4 migrate([a,c],[a,b],node(e,0))
 Scene: 5 node(a,node(b,node(e,0))//node(c,node(d,0)//0))
Inactive final state
New Network=node(a,node(b,node(e,0))//node(c,node(d,0)))
More solutions? ;

History:
 Scene: 1 node(a,node(b,go([a,c],node(d,0)))//node(c,go([a,b],node(e,0))))
 Move:  2 migrate([a,c],[a,b],node(e,0))
 Scene: 3 node(a,node(b,node(e,0)//go([a,c],node(d,0)))//node(c,0))
 Move:  4 migrate([a,b],[a,c],node(d,0))
 Scene: 5 node(a,node(b,node(e,0)//0)//node(c,node(d,0)))
Inactive final state
New Network=node(a,node(b,node(e,0))//node(c,node(d,0)))
More solutions? ;

No (more) solutions
```

5 Future work

In the present proposal names are "static entities". One might want to relax the side condition of the parallel rule and provide instead a renaming mechanism that renames a migrating node when it is going to be inserted in parallel with another node having the same name.

In our approach the paths contained in an agent do not change during migration. This means that the relative address specified by a path inside an agent will refer, after migration, to a location different than the one before migration. This might be regarded as undesirable. One direction of future work is to enrich the calculus so to ensure location invariance during migration.

One of the advantages of SOS semantics is that it helps in developing an algebraic theory of the language, based on the concept of bisimulation. This task is particularly facilitated when the rules are in the so-called De-Simone format [9, 11], or similar formats [4], since such formats ensures that bisimulation is a congruence. In our case the labels of the transitions contain agents and therefore we need to consider a sort of higher-order extension of the De-Simone format along the lines of [2]. In the future we intend to check whether the format of our rules is in some sort of extended De-Simone format for which the congruence theorem holds, and then try to determine the algebraic laws of the language following similar work done in first-order process algebras [1].

References

1. Luca Aceto, Bard Bloom, and Frits Vaandrager. Turning SOS rules into equations. *Information and Computation*, 111(1):1–52, 1994.
2. Karen L. Bernstein. A congruence theorem for structured operational semantics of higher-order languages. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 153–164, 1998.
3. K. Bharat and L. Cardelli. Migratory applications. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 131–148. Springer-Verlag, 1997.
4. Bard Bloom, Sorin Istrail, and Albert R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, 1995.
5. Chiara Bodei, Pierpaolo Degano, and Corrado Priami. Names of the π-calculus agents handled locally. *Theoretical Computer Science*.
6. G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn. Observing localities. *TCS*, 114(1):31–61, June 1993.
7. L. Cardelli. Mobile computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222, pages 3–6. Springer-Verlag, 1997.
8. Luca Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, Winter 1995.
9. R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Science*, 37(3):245–267, 1985.
10. C. Fournet, G. Gonthier, JJ. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. CONCUR'97*, LNCS 1119, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag.
11. Jan Friso Groote and Frits Vaandrager. Structured operational semantics and bisimulation as a congruence. *Information and Computation*, 100(2):202–260, 1992.
12. R. Milner, J. Parrow, and D Walker. A Calculus of Mobile Processes. *Information and Control*, 100:1–77, 1992.
13. Jean-Hugues Réty. Distributed concurrent constraint programming. *Fundamenta Informaticae*, 34(3):323–346, 1998.
14. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, Department of Computer Science, Edinburgh University, 1993.
15. Davide Sangiorgi. π-Calculus, Internal Mobility and Agent-Passing Calculi. *TCS*, 167(1,2):235–274, 1996.
16. Vijay Saraswat, Martin Rinard, and Prakash Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *POPL 91*, pages 333–352. ACM Press, 1991.

17. Bent Thomsen. Plain CHOCS. A second generation calculus for higher order processes. *Acta Informatica*, 30(1):1–59, 1993.
18. Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, 1997.