
Coordinating mobile agents in interaction spaces
Frédéric Peschanski*, Alexis Darrasse, Nataliya Guts, Jérémy Bobbio

Université Pierre et Marie Curie - Paris 6 (LIP6), Paris, France

Received 1 February 2006; received in revised form 1 May 2006; accepted 1 January 2007
Available online 25 April 2007

Abstract

We present a kernel coordination language for mobile agent systems that considers as first-class citizens both the agents and the channels they use to interact with each other. Channels implement distributed, asynchronous communications with FIFO ordering and multicast routing. Features related to mobility include agent migration as well as remote cloning. Most importantly, a virtual form of channel mobility inspired by the π-calculus is also supported. This expressive feature allows mobile agents to adapt dynamically to their changing environment. The language semantics, presented formally, is based on a geometrical model named the Interaction Spaces. This provides an intuitive interpretation of the agent features and capabilities in terms of combined spatial projections and transformations. Through spatial composition, we show that standard labeled transition systems and bisimulation-based semantics may be defined above the geometry, enabling reasoning and formal verification. Finally, we describe prototype implementations of the proposed model and language.

Keywords: Mobile agents; Process algebras; Coordination languages; Geometry

1. Introduction

In this paper, we present a kernel language for specifying and reasoning about the coordination aspects of mobile agent systems. It is designed as an intermediate between a process algebra [5,7,16] and a process-centric, endogenous coordination language [3,8,19]. The coordinating behaviors of agents are modeled as concurrent, distributed and mobile processes. Given their geographic distribution, agents have to communicate in order to interact with each other. This is done through communication channels considered as first-class citizens and equipped with rich communication semantics. In this paper, we describe a versatile model of asynchronous communications with FIFO ordering and multicast routing. Most importantly, agents can migrate and thus “hop” from host to host in order to perform distributed computations. They may also be replicated as remote clones. During such operations, the surrounding environment of agents is likely to change, which may require them to adapt to new situations. This is modeled through the π-calculus notion of channel mobility, the possibility to exchange communication channels among agents at runtime.

Our goal is to provide both a specification language to model and support reasoning about the coordination of mobile agent systems, and an executable language to control their deployment and operation at runtime. Developing
a practical language and sound semantic model expressive enough to encompass all the proposed agent features is a challenging goal. On the one hand, the language semantics must be simple enough to remain tractable. This implies major constraints such as the absence of redundant constructs and, if possible, the avoidance of second-order principles. On the other hand, the practical realizations of the language should be made as efficient as possible, which also imply constraints on both the language and the underlying support mechanisms such as the avoidance of centralization, and the minimization of distributed synchronizations.

The coordination model we propose is original in that it employs a metaphor of geometrical spaces – the Interaction Spaces – in which agents interact through simple transformations: expansions, translations, etc. This geometrical metaphor provides an intuitive interpretation of the agent features. Beyond the intuition, this makes the model to relying on a reduced set of basic concepts and axioms that we may illustrate and justify in geometrical terms. A technical contribution that we also believe original is the use of a versatile spatial composition operator that allows the immersion of the geometry within standard structural operational semantics based on labeled transition systems (LTS) and bisimulation. This is a preliminary but important step for the formal reasoning about systems expressed in the proposed language.

The outline of the paper is as follows. In Section 2 we illustrate the language constructs and their expressiveness on motivating examples. We then present the geometry of interactions spaces in Section 3. The operational semantics for the language are detailed in Section 4. Finally, in Section 5 we overview the libispace library and server as well as the hspace frontend that implement the proposed model and language. A panorama of related work, conclusion and references follow. This paper is a largely revised and extended version of [20], with less emphasis on the technical aspects of the formal foundations. The adjunct of Sections 2 and 5 shed more lights on the practicability of the proposed approach.

2. Kernel coordination language

We illustrate in this section the expressiveness of the proposed kernel coordination language. We first develop a simple transactional system to explain the basic language constructs. We show how the structure of the system may be changed at runtime through channel mobility [16]. The second example describes a graphical application dealing with mobile users. Finally, the formal syntax of the language is described.

2.1. Communicating systems

Our first example illustrates the expressiveness of the language to describe the coordination behavior of geographically distributed communicating systems. It encodes the voting phase of the 2-Phase Commit (2PC) protocol in a transaction system. We separate the coordinator agent and the voter agents. The system is deployed using the following script:

```plaintext
new(ask:<<bool>>), new(reply:<bool>), new(commit:<bool>),
spawn(coord@is://site1){dockn(ask,3), link(reply), dock(commit),
Coordinator(ask,reply,commit,3)},
spawn(voter1@is://site2){link(ask), Voter(ask)},
spawn(voter2@is://site3){link(ask), Voter(ask)},
spawn(voter3@is://site4){link(ask), Voter(ask)}
```

The `new` prefix is used to create channels identified by a name and a type. The first channel we create is named `ask`. It is used by the coordinator to ask the voters to take their decision. The type expression `<<bool>>` indicates that the channel `ask` is used to carry other channels carrying booleans. It is thus a channel carrying channels. We also create a channel `reply` carrying booleans used by the voters to indicate their choice. The channel `commit` is used by the coordinator to send its final decision.

The deployment of an agent is performed by the `spawn` prefix. It is parameterized by the name of the created agent, which corresponds to its virtual location. The coordinator location is named `coord`. We also indicate the URL of the physical, implementation-dependent location, which we call a `site` (of basic type `url`). The coordination behavior of the spawned agent is enclosed between curly brackets. The goal of the coordinator is to ask the voters to vote, and then collect their decisions. The `dock` (and its counted variant `dockn`) and `link` prefixes are used to acquire respectively
the write and read capacities on channels. An agent may not communicate before acquiring such a capacity. Here, the coordinator agent is docked to channel ask, which enables emissions, and it is linked to channel reply enabling receptions. The integer parameter for the dockn variant (here 3) represents the minimum number of agents that must be linked to the channel so that the prefix is consumed. In this example, the coordinator will wait for at least 3 voters to acquire the read capacity on ask. It then calls the Coordinator definition. In the continuation, we spawn three voters that link to the ask channel and call the Voter definition.

Flowgraphs are commonly used to describe the structure of interactive systems [16]. The flowgraph of the transaction system is shown on Fig. 1. End-points of channels are represented by dots (dock capacity for emitting) and arrowheads (link capacity for receiving). We can see that the ask channel is a multicast channel with one emitter and three receivers. Note also that there is no way (yet) for the voters to send information back to the coordinator.

The Coordinator definition is specified as follows:

```plaintext
def Coordinator(ask:bool, reply:bool, commit:bool, nb:int) =
    ask!(reply), CoordWait(reply, nb, commit, true)

def CoordWait(reply: bool, nb: int, commit: bool, decision: bool) =
    if nb > 0 then reply?(vote), CoordWait(reply, nb - 1, commit, vote & decision)
    else commit!(decision)
```

The definition is parameterized by the three communication channels ask, reply and commit. The last parameter, an integer, represents the number of votes to wait for. To emit a value v on a channel c, we write c!(v). Conversely, a process performing the c?(x) prefix waits for a value to be received on c, and then binds this value to variable x. The coordinator first sends the reply channel on ask. It then calls the CoordWait definition which is used to wait for the vote decisions.

The voter code is as follows:

```plaintext
def Voter(ask:bool) =
    ask?(reply), dock(reply),
    + [τ.reply!(true), undock(reply)]
    + [τ.reply!(false), undock(reply)]
```

Each voter receives on ask the reply channel. They then dock to this channel and perform a non-deterministic choice between either a positive or negative vote. From a coordination point of view, the + operator with τ prefixes tells that the choice of either branch of execution may be decided internally by the agent. Fig. 2 describes the structure of the 2PC system at this point of execution. After the emission of the decision, the voter must release the write capacity (undock) so that another voter may write on the channel. The uniqueness of the write capacity is the primary mean of synchronization in the language. As the semantics will make clear, any code enclosed between a dock and an undock prefix is a critical section. The atomic emission on a channel c may be written dock(c),c!(v),undock(c) or, equivalently, c!!(v). The expression c??(x) is equivalent to link(c),c?(x),unlink(c) but it is not a synchronization barrier since the read capacity is not unique (multicast semantics).

---

1 The dockn primitive is needed to ensure a correct initial communication. It may not be encoded using the other constructs (even with parametric recursion) because of an atomicity constraint that requires the introduction of a dedicated rule in the operational semantics (cf. Section 4).
2.2. Mobile systems

The second example describes a computational environment in which users move from devices to devices. On each device a dedicated graphical user interface (GUI) agent is running. We suppose the following abstract specifications for GUI agents on respectively personal digital assistants (PDA) and desktop computers:

\[
\text{def PdaGUI}(\text{togui} : \langle \text{ViewEvent} \rangle, \text{fromgui} : \langle \text{ControlEvent} \rangle) = \\
\text{togui}?(\text{req}), \ldots , \text{fromgui}!(\text{ctl}), \ldots 
\]

\[
\text{def DesktopGUI}(\text{togui} : \langle \text{ViewEvent} \rangle, \text{fromgui} : \langle \text{ControlEvent} \rangle) = \\
\text{togui}?(\text{req}), \ldots , \text{fromgui}!(\text{ctl}), \ldots 
\]

The channels request and reply are respectively used to send and receive information to and from the GUIs. An example of behavior for a GUI client\(^2\) is as follows:

\[
\text{def Client1}(\text{togui} : \langle \text{ViewEvent} \rangle, \text{fromgui} : \langle \text{ControlEvent} \rangle, \text{move} : \langle \text{url} \rangle) = \\
[ \text{fromgui}?(\text{ctl}), \ldots , \text{togui?(view)}, \ldots ] \\
+ [ \text{move?(dest)}, \text{go}(\text{@}@\text{dest}), \text{Client1}(\text{togui}, \text{fromgui}, \text{move}) ]
\]

This definition is comprised of a GUI interaction part (first branch of the + choice) and a mobile part (second branch). A user movement is triggered by the channel move that receives the URL where the software agent should move. The go prefix realizes the movement, binding to name l the new (virtual) location of the agent. On Fig. 3 we describe the effect of the go prefix on the physical environment. The dashed arrows and boxes correspond to the agents and channels before the described movement. We can see that the environment of the client agent, as captured by its dock and link capacities on channels, is invariant through migration. In this example, the user changed its device (from a desktop computer to a PDA), and the client agent followed this move, but the user interface is still incorrectly bound to the desktop computer. The solution we propose is to modify the dock and link capacities of the GUI-related channels, as follows:

\(^2\)In this section, we only describe the coordination patterns between client and GUI agents, and abstract away from implementation-dependent details.
Fig. 4. Agent mobility with channel mobility.

\[
\text{def Client2(togui:<ViewEvent>,fromgui:<ControlEvent>,}
\]
\[
\text{move:<url>*<ViewEvent>*<ControlEvent>>)} = 
\]
\[
[ \text{fromgui?(ctl), ... , togui!(view), ... ]} 
\]
\[
+ \[ \text{move?(dest,view,ctl),go(l@dest),undock(togui),dock(view),}
\]
\[
\text{unlink(fromgui),link(ctl),Client2(view,ctl,move) } ] 
\]

The move channel now transports tuples of channels\(^3\) corresponding to the whole agent interface. The new definition thus combines agent and channel mobility, as illustrated by Fig. 4. Now, the GUI channels are correctly rebound to the PdaGUI agent, as expected. Finally, we may describe the effect of performing remote cloning instead of migration, as specified by the following definition:

\[
\text{def Client3(togui:<ViewEvent>,fromgui:<ControlEvent>,}
\]
\[
\text{move:<url>*<ViewEvent>*<ControlEvent>>)} = 
\]
\[
[ \text{fromgui?(ctl), ... , togui!(view), ... ]} 
\]
\[
+ \[ \text{move?(dest,view,ctl),clone(l@dest),Client3(view,ctl,move) } ] 
\]

In this variant, depicted on Fig. 5, a clone for the client agent is instantiated, and thus two instances are running in parallel. Since the dock capacity on a given channel is unique, it is retained by the source agent. Further coordination is needed to enable dock capacities on the clone’s side.

2.3. Formal syntax

The proposed syntactic constructs are shown in Table 1. We define type expressions, process definitions and expressions. Types fall in the categories of either basic types\(^4\) or channel types. Parameterized and potentially recursive

\(^3\) The symbol * denotes the cross-product in type expressions.

\(^4\) We only use basic types that are common in almost every programming languages. Higher-level types are being considered for future work. We do not investigate the type system furthermore in this paper and assume a simple \(\pi\)-calculus type system [22]. An implementation of these simple typing rules is provided in the hspace frontend (cf. Section 5).
Table 1
The syntax of agents

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t$</td>
<td>::= [ \text{int}</td>
</tr>
</tbody>
</table>

The condition $v$ in an alternative $\text{if } v \text{ then } P \text{ else } Q$ should be evaluated to a boolean $\text{true}$ or $\text{false}$. We do not describe the evaluation process in the paper, a simple call-by-value evaluator is assumed.

3. Interaction Spaces

In this section, we develop the model of Interaction Spaces, a geometrical characterization of the coordination primitives presented in the previous section.

3.1. The geometry of Interaction Spaces

An example of an Interaction Space is represented on Fig. 6. It is a geometrical space consisting of a dimension $\mathcal{L}$ of locations (or agent identities), a dimension $\mathcal{C}$ of channels (or channel identities), and for every pair $(l, c)$ of location and channel, a dimension $\mathcal{A}$ of acts (or channel state). The content of an Interaction Space is made of two kinds of information:

- values in set $\mathcal{V}$ that populate dimensions of acts $\mathcal{A}$, reflecting the contents of transmitted data among agents.
- colors in set $\mathcal{C} \equiv \{ \text{read}, \text{write}, \text{know}, \text{lfail}, \text{cfail}, \text{kfail} \}$ that are properties decorating channel or location references to maintain control-level information such as ownership of channels. The colors $\text{lfail}, \text{cfail}$ and $\text{kfail}$ respectively deal with location, channel and link failures.\(^6\)

---

\(^5\) We do not present the complete failure semantics in the paper. However, we take care to avoid communication on failed channels and mobility with failed locations.
The formal definition is as follows:

**Definition 1.** The denotation of an Interaction Space $\Delta$ is a partial function element of set $\mathbb{E} \cong \mathbb{N}^2 \rightarrow \wp^C \times (\mathbb{Z} \rightarrow \mathbb{V})$. In this definition, $\wp^C$ is the powerset of $C$, $\mathbb{N}$ and $\mathbb{Z}$ are the usual natural and integer sets.

An interaction space thus maps a set of pairs $(l, c)$ of location and channel identities (positive integer coordinates) to a pair $(C, A)$ where $C$ is a set of colors and $A$ is an act dimension, modeled as a partial function from positive integers (message ordering) to values (message contents). Let $\Delta$ be the interaction space represented in Fig. 6. We have, for example, $\Delta(l_1, c) = ((\text{know, read}), \emptyset)$ and $\Delta(l_2, c) = ((\text{know, write}), \emptyset)$. If we suppose that agent $l_1$ is able to receive a message $m$ on channel $c$, then we would have $\Delta(l_1, c) = ((\text{know, read}), [0 \mapsto m])$ and thus $\text{snd}(\Delta(l_1, c))(0) = m$.

### 3.1.1. Transformations and projections

Interaction Spaces and their contents are manipulated through expansions $\mathcal{E}_D$ (in each possible dimension $D$) and elementary transformations categorized as unit translations $\mathcal{T}_D$ (in dimension $D$), carbon copies $\mathcal{K}_D$, paints $\mathcal{P}_c$ and inversions $\mathcal{P}_C$ (of color $c$) as well as fills $\mathcal{F}_v$ (of value $v$). Expansions and transformations may be composed using the notation $\theta \theta' \Delta$ in which $\Delta$ is first transformed by $\theta'$ and then by $\theta$.

**Definition 2.** We define various projections of an Interaction Space $\Delta$:

- $\Delta(L \subseteq L_\Delta, C \subseteq C_\Delta) \in \wp^C \times (\mathbb{Z} \rightarrow \mathbb{V})$
- $\Delta(l \in L_\Delta, c \in C_\Delta) \in \wp^C \times (\mathbb{Z} \rightarrow \mathbb{V})$
- $\text{fst}(\Delta(l \in L_\Delta, c \in C_\Delta)) \in \wp^C$
- $\text{snd}(\Delta(l \in L_\Delta, c \in C_\Delta))(a \in A_\Delta(l, c)) \subseteq \mathbb{V}$.

The simplest projection is on a given location $l$ and channel $c$, written $\Delta(l, c)$. The colors and dimension of acts of a particular intersection are respectively noted $\text{fst}(\Delta(l, c))$ and $\text{snd}(\Delta(l, c))$. The domain of a dimension of acts is noted $A_\Delta(l, c)$. Projected subspaces may serve as restriction for transformations, with the notation $(T, \delta)\Delta$ in which the space $\Delta$ is transformed by $T$ only in its projected subspace $\delta$. One may also project on whole dimensions as follows:

**Definition 3.** The dimensions of an Interaction Space $\Delta \in \mathbb{E}$ are:

- $L_\Delta \cong \text{fst}(\text{dom}(\Delta))$ the dimension of locations
- $C_\Delta \cong \text{snd}(\text{dom}(\Delta))$ the dimension of channels
- $A_\Delta(l \in L_\Delta, c \in C_\Delta) \cong \text{dom}(\text{snd}(\Delta(l, c)))$ a dimension of acts.

### 3.1.2. Freshness

The notion of freshness is of central importance in process calculi to discuss about the creation and handling of private resources. There are two kinds of resources to consider here: channels and locations. To create a new location (resp. a new channel), we have to expand a given space in the dimension $L$ of locations (resp. $C$ of channels).

The concerned transformations and projections are as follows:

- $\text{newloc}(\Delta) \equiv \mathcal{E}_L \Delta$
- $\text{locref}(\Delta) \equiv \text{card}(L_\Delta) - 1$
- $\text{newchan}(\Delta) \equiv \mathcal{E}_C \Delta$
- $\text{chanref}(\Delta) \equiv \text{card}(C_\Delta) - 1$.

The freshness properties that must be established are:

(fresh-location) $\text{locref}(\text{newloc}(\Delta)) \notin L_\Delta$

(fresh-channel) $\text{chanref}(\text{newchan}(\Delta)) \notin C_\Delta$. 
3.1.3. Dynamic flowgraphs

Fig. 7(a) shows a flowgraph and its corresponding Interaction Space. In this system, the channel $c$ is *docked* to agent $l_2$, which may thus perform emissions. The channel $c$ is also *linked* to agents $l_1$ and $l_3$. The intersection of a channel $c$ docked to a location $l$ is painted using color *write*. This is shown as a black pavement in the figures. Linked locations are decorated by color *read*, which is represented by a blue pavement. On Fig. 7(b), the dock and link capacities of $c$ are modified. To model such modification, we define functions to get and set colors at specific intersections of channels and locations:

\[
\begin{align*}
\text{cget}(\Delta, l, c) & \equiv \text{fst}(\Delta(l, c)) \\
\text{cset}(\Delta, l, c, \text{color}) & \equiv (\text{P}_{\text{color}}, \Delta(l, c)) \Delta \\
\text{cunset}(\Delta, l, c, \text{color}) & \equiv (\text{P}_{\text{color}}, \Delta(l, c)) \Delta.
\end{align*}
\]

The expected properties are:

(cset-safety) $\text{color} \in \text{cget(cset}(\Delta, l, c, \text{color}), l, c)$

(cunset-safety) $\text{color} \notin \text{cget(cunset}(\Delta, l, c, \text{color}), l, c)$.

An Interaction Space also carries a derived color, named *know*, that indicates the channel identities known by each given location (or agent). In the figures, this is represented as a dashed outline for location/channel intersections. An agent at location $l$ may only manipulate a channel $c$ if $\text{know} \in \text{cget}(\Delta, l, c)$.

3.1.4. Communication

Once a correct flowgraph has been set up using *dock* and *link* operations, communications between agents may be performed. We illustrate the emission of an information represented as a white cube\(^7\) on Fig. 8(b). The cube is emitted by agent $l_2$ and transits on channel $c_3$ (docked to $l_2$). A copy of the cube is put in every locations the channel $c_3$ is

\(^7\)The colors of the cubes used in the figures are arbitrary. We use different colors to illustrate the ordering property of channel semantics.
linked to (i.e. \( l_1 \) and \( l_3 \)). For this we need first to expand the dimension of acts for linked agents with:

\[
\text{linked}(\Delta, c) \ni \{ l \in L_\Delta \mid \text{read} \in \text{cget}(\Delta, l, c) \}
\]

\[
\text{newact}(\Delta, c) \equiv \bigcup_{l \in \text{linked}(\Delta, c)} \mathcal{E}_\Delta(\Delta, l, c).
\]

Once the space has been expanded, we may multicast the emitted value by filling the subspace of linked agents for the channel. For this we write:

\[
\text{mcast}(\Delta, c, v) \equiv \bigcup_{l \in \text{linked}(\Delta, c)} (\mathcal{F}_v, \text{snd}(\Delta(l, c))(\max(\mathcal{A}_\Delta(l, c)))) \Delta.
\]

We then combine the previous two transformations to perform the emission as follows:

\[
\text{send}(\Delta, c, v) \equiv \text{mcast} \circ \text{newact}(\Delta, c, c, v).
\]

On Fig. 8(c) we depict the effect of sending another information represented as a black cube by the same agent \( l_2 \). It is as if the cube is stacked over the previous one, introducing an order relation.

The reception process is depicted on Fig. 8(d). Here, agent \( l_3 \) is performing the reception. The value to receive is a dot in the space at coordinates \( \text{snd}(\Delta(l_3, c_3))(0) \), which corresponds to the oldest emitted value on channel \( c_3 \) for location \( l_3 \). The following projection gets this value:

\[
\text{fetch}(\Delta, l, c) \equiv \text{snd}(\Delta(l, c))(0).
\]

To perform the reception in geometrical terms, we have to translate a unit in the dimension of acts, which we write:

\[
\text{pull}(\Delta, l, c) \equiv (\mathcal{T}_\Delta, \text{snd}(\Delta(l, c))) \Delta.
\]

It is possible that a new channel identity is received. In this case, we have to complete the definition in order to update the \texttt{know} color:

\[
\text{receive}(\Delta, l, c) \equiv \text{cset}(\text{pull}(\Delta, l, c), l, \text{fetch}(\Delta, l, c), \texttt{know}).
\]

Note that values, once fetched, do not disappear from the Interaction Space. The subspace of negative coordinates in the dimension of acts characterizes precisely the history of interactions. The main requirement is the implementation of FIFO semantics:

\[
\text{(send-fifo)} \forall l \in \text{linked}(\Delta, c), \Delta \text{ be such that } \mathcal{A}_\Delta(l, c) = \emptyset \text{ and let } \Delta' = \text{send}(\text{send}(\Delta, c, v_1), c, v_2), \text{ then } \text{fetch}(\Delta', l, c) = v_1 \text{ and } \text{fetch}(\text{receive}(\Delta', l, c), l, c) = v_2.
\]

3.1.5. Mobility

The model of Interaction Spaces support the two most common forms of mobility: agent mobility (so-called migration) and channel mobility. We already discussed the implementation of dynamic flowgraphs, similar to the kind of mobility implemented in the \( \pi \)-calculus. The isolated movement of channel states is a prerequisite to agent mobility, as we discuss in [14]. In the semantics, this corresponds to a simple translation defined as follows:

\[
\text{xfer}(\Delta, c, l_1, l_2) \equiv (\mathcal{T}_\Delta, \Delta([l_1, l_2], c)) \Delta \text{ with } l_1 < l_2.
\]

The important properties of \texttt{xfer} are as follows:

\[
\text{(xfer-cleanup)} \text{xfer}(\Delta, c, l_1, l_2)(l_1, c) = \emptyset \quad \text{xfer-safety} \text{xfer}(\Delta, c, l_1, l_2)(l_2, c) = \Delta(l_1, c).
\]

The cloning of a channel, named \texttt{xcpy}, is similar but the source location is not modified during the movement. At the destination, everything is similar except for the dock capacities that are empty. The rationale is that the channel clone may not acquire any dock capacity because such a capacity is unique and the starting location has to undock in order to loose the capacity. So we define:

\[
\text{xcpy}(\Delta, c, l_1, l_2) \equiv (\mathcal{K}_\Delta, \Delta([l_1, l_2], c)) \Delta \text{ with } l_1 < l_2.
\]
The important properties of \textit{xcpy} are as follows:

\begin{align*}
(xcpy\text{-}preserve) \quad & xcpy(\Delta, c, l_1, l_2)(l_1, c) = \Delta(l_1, c) \\
(xcpy\text{-}safety) \quad & xcpy(\Delta, c, l_1, l_2)(l_2, c) = \Delta'(l_1, c) \text{ with } \Delta' = \Delta \text{ excepted for } write \notin cget(\Delta', l_2, c).
\end{align*}

Fig. 9 describes the movement of a whole agent. As seen on the figure, this comprises the movement of all channels ends and channels states bound to the mobile agent. The transition from Fig. 9(a)–(b) may be decomposed in two steps. First, we have to create the target location $l_3$. The rationale is that we do not allow race conditions on locations at the semantic level.\footnote{Of course, implementations should map locations to physical, distributed sites that permit sharing.} The migration phase itself consists in translating the subspace of all channels at the source location to the newly created one, which we formalize as follows:

\[
migrate(\Delta, l) \triangleq xfer(\Delta, l, l', C) \text{ with } l' = \text{locref}(\Delta).
\]

The whole movement is a combination of creating a location and moving:

\[
move(\Delta, l) = migrate(\text{newloc}(\Delta), l).
\]

The properties we expect from this operation are:

\begin{align*}
\text{(move\text{-}cleanup)} \quad & move(\Delta, l)(l, C_\Delta) = \emptyset \\
\text{(move\text{-}safety)} \quad & move(\Delta, l)(\text{locref}(\Delta), C_\Delta) = \Delta(l, C_\Delta).
\end{align*}

For \textit{remote cloning}, the main difference is that the source location is not cleaned up, and of course dock capacities are preserved and absent at the clone location. This is described by Fig. 10(a) and (b). The formal definitions are similar except that \textit{xcpy} is used instead of \textit{xfer}:

\[
copy(\Delta, l) \triangleq xcpy(\Delta, l, l', C) \text{ with } l' = \text{locref}(\Delta).
\]

The whole cloning is a combination of creating a location and copying:

\[
clone(\Delta, l) = copy(\text{newloc}(\Delta), l).
\]

The properties of cloning are:

\begin{align*}
\text{(clone\text{-}preserve)} \quad & clone(\Delta, l)(l, C_\Delta) = \Delta(l, C_\Delta) \\
\text{(clone\text{-}safety)} \quad & clone(\Delta, l)(\text{locref}(\Delta), C_\Delta) = \Delta'(l, C_\Delta) \text{ with } \Delta' = \Delta \text{ except for } write \notin cget(\Delta', \text{locref}(\Delta), C_\Delta).
\end{align*}
3.2. Spatial composition

We now describe the aspect of Interaction Space composition, an important prerequisite for the composition of agent behaviors. For this purpose we introduce the spatial composition operator $\otimes$. In $\Delta_1 \otimes_{\lambda,\gamma} \Delta_2$ we combine two spaces $\Delta_1$ and $\Delta_2$. The operator is parameterized by a map $\lambda$ of locations and $\gamma$ of channels from the left-hand space to the right-hand space. All mappings must be consistent with respect to the contents of the composed spaces. This means that the subspaces denoted by the mapped locations and/or channels must be equal. For example, in $\Delta_1 \otimes_{l_1 \mapsto l_2', c_1 \mapsto c_2'} \Delta_2$, we must ensure for example that $\Delta_1(l_1, c_1) = \Delta_2(l_2', c_2')$.

**Definition 4.** The spatial composition operator is a function of $\mathbb{E} \times \mathbb{E} \times (\mathbb{N} \rightarrow \mathbb{N}) \times (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{E}$, noted $\Delta_1 \otimes_{\lambda,\gamma} \Delta_2$ such that $\Delta_1, \Delta_2 \in \mathbb{E}$, $\lambda \in \lambda_{\Delta_1} \rightarrow \lambda_{\Delta_2}$, $\gamma \in \gamma_{\Delta_1} \rightarrow \gamma_{\Delta_2}$ and $\forall l \in \lambda_{\Delta_1} \cup \lambda_{\Delta_2}$, $\forall c \in \gamma_{\Delta_1} \cup \gamma_{\Delta_2}$:

\[
\begin{align*}
(1) & \quad (\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)(l, c) = \Delta_1(l, c) \text{ if } l \in \lambda_{\Delta_1}, \ c \in \gamma_{\Delta_1} \\
(2) & \quad (\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)(l, c) = \Delta_2(\lambda(l), \gamma(c)) \text{ if } l \in \lambda_{\Delta_1}, \ c \in \gamma_{\Delta_2} \\
(3) & \quad (\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)(l, c) = \Delta_1(l, c) \text{ if } l \in \lambda_{\Delta_2}, \ c \not\in \gamma_{\Delta_2} \\
(4) & \quad (\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)(l, c) = \Delta_2(l, \gamma(c)) \text{ if } l \not\in \lambda_{\Delta_1}, \ c \in \gamma_{\Delta_2} \\
(5) & \quad (\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)(l', c') = \Delta_2(l, c) \text{ if } l \in \lambda_{\Delta_2} \setminus \gamma_{\Delta_2} \text{ and } c \in \gamma_{\Delta_2} \setminus \gamma_{\Delta_2} \text{ with } l \mapsto l' \in \sigma_{\lambda \otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2) \text{ and } c \mapsto c' \in \sigma_{\gamma \otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2).
\end{align*}
\]

The result of spatial composition is a combined space in which, by default, locations and channels from both the operand spaces are distinguished. We may use $\emptyset$ or simply omit the $\lambda$ and $\gamma$ parameters in this case. This means that locations and/or channels from the right-hand side space can be updated through spatial composition. We define the substitutions $\sigma_{\lambda \otimes}$ and $\sigma_{\gamma \otimes}$ to record all updates:

**Definition 5.** The spatial updates substitutions are as follows:

- $\sigma_{\lambda \otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2) = \bigcup\{l \mapsto l'\}$ with $l \in \lambda_{\Delta_2} \setminus \gamma_{\Delta_2}$ and $l' \not\in \lambda_{\Delta_1} \cup \lambda_{\Delta_2}$
- $\sigma_{\gamma \otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2) = \bigcup\{c \mapsto c'\}$ with $c \in \gamma_{\Delta_2} \setminus \gamma_{\Delta_2}$ and $c' \not\in \gamma_{\Delta_1} \cup \gamma_{\Delta_2}$
- $\Sigma_{\lambda,\gamma}(\Delta_1, \Delta_2) = \lambda^{-1} \circ \lambda_{\otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2) \circ \gamma^{-1} \circ \sigma_{\gamma \otimes}(\Delta_1 \otimes_{\lambda,\gamma} \Delta_2)$.

In the definition above, the cosupport members (e.g. $l'$ and $c'$) must be chosen fresh. For a given couple $(\Delta, \Gamma)$ of Interaction Spaces, we note $\Sigma_{\lambda,\gamma}(\Delta_1, \Delta_2)$ the set of all substitutions that must be taken into account in processes coming from the separate spaces $\Delta_1$ and $\Delta_2$, and merged within the same resulting composition $\Delta_1 \otimes_{\lambda,\gamma} \Delta_2$. As an illustration consider the case, say $\Delta_1$, represented on the left of Fig. 11 with locations $l_1, l_2$ and channels $c_1, c_2$. The figure shows the composition $\Delta_1 \otimes_{l_1 \rightarrow l_1', c_1 \rightarrow c_1'} \Delta_2$ in which $\Delta_2$ has locations $l_1', l_2'$ and channels $c_1', c_2'$. The result is a space $\Delta_3$ with three locations $l_1, l_2, l_3$ and channels $c_1, c_2, c_3$ such that $\sigma_{\lambda \otimes}(\Delta_1 \otimes_{l_1 \rightarrow l_1', c_1 \rightarrow c_1'} \Delta_2) = \{l_2' \mapsto l_3\}$ and $\sigma_{\gamma \otimes}(\Delta_1 \otimes_{l_1 \rightarrow l_1', c_1 \rightarrow c_1'} \Delta_2) = \{c_2' \mapsto c_3\}$.

Interestingly, the spatial composition operator denotes a commutative space model for which the natural equality is a lot more flexible than the set-theoretic equality identifying Interaction Spaces with identical dimensions, contents and colors.

**Definition 6.** Two Interaction Spaces $\Delta_1, \Delta_2 \in \mathbb{E}$ are spatial permutations if, and only if, $\exists \lambda, \gamma \in \mathbb{N} \rightarrow \mathbb{N}$ such that $\lambda_{\Delta_1} = \lambda_{\Delta_2}$, $\lambda_{\Delta_2} = \lambda_{\Delta_1}$, $\gamma_{\Delta_1} = \gamma_{\Delta_2}$, $\gamma_{\Delta_2} = \gamma_{\Delta_1}$, and $\forall l \in \lambda_{\Delta_1}, \forall c \in \gamma_{\Delta_2}$, $\Delta_2(\lambda(l), \gamma(c)) = \Delta_1(l, c)$.

**Definition 7.** The spatial equivalence on Interaction Spaces is defined as follows. Let $\Delta_1 \in \mathbb{E}$ and $\Delta_2 \in \mathbb{E}$, then $\Delta_1 = \Delta_2$ if, and only if, $\Delta_1$ and $\Delta_2$ are spatial permutations.
The intuition is that two spaces are spatially equivalent if the only way they differ is obtained by permutations of dimensions of locations or channels. To ensure this, the mappings must cover the full domain of both Interaction Space, and of course they must have the same number of locations and channels.

Fig. 12 illustrates the property. It is simple enough to prove that it is, indeed, an equivalence relation. Reflexivity is obtained by considering the identity functions for \( \lambda \) and \( \gamma \). Symmetry is obtained through \( \lambda^{-1} \) and \( \gamma^{-1} \) and the composition of the two mapping functions naturally leads to the transitivity result.

4. Operational semantics

We describe in this section the formal operational semantics of the proposed coordination language. It is described using the usual presentation of *Structured Operational Semantics* for process algebras. The section concludes with a set of behavioral equivalences that enable reasoning on system specifications.

4.1. Basic operators

The semantic rules we propose in this section and the following one apply on *terms* built using the syntax of the language (see Table 1). To form *agent behaviors*, the process expressions must be adjoined with Interaction Spaces. The syntactic category of agent behaviors is formed as follows:

\[
\text{agent} \quad A, B, \ldots \quad ::= \quad [P]@l \mid A \parallel B.
\]

This says that an agent is either a basic agent running a process expression \( P \) at some location \( l \), or a set of agents running in parallel.

The relation of structural congruence (cf. Table 2) identifies “syntactically” equivalent terms. The rule SC4 describes the main connection point between the geometrical semantics of Interaction Spaces and the process-algebra semantics proposed here. It shows how two agents evolving in separate Interaction Spaces may be composed through the spatial composition of their respective spaces, and the parallel composition of the agent specifications or programs. Of course, the updates of the right-hand space must be taken into account by applying the substitutions for both locations and channels (see Section 3.2). A simpler case corresponds to agents evolving in equal spaces (cf. rule O1 in Table 3). LTS transitions are labeled by observable actions performed by the agents. Each observation \( \alpha@l \) corresponds to the performance of a prefix \( \alpha \) at a location \( l \). This concerns all the prefixes of the syntax (see Table 1) except the spawn, new and dockn prefixes that are not observable and coincide with the internal step \( \tau \). By observable
Table 3

Operational semantics: basic axioms and operator rules

(A1) \[ \Delta \vdash \text{spawn}(l_1)(Q)@l_2, P \xrightarrow{\tau@l_2} \text{newloc}(\Delta) \vdash (P || [Q]@l_1)[\text{locref}(\Delta)/l_1] \]

(A2) \[ \Delta \vdash \text{new}(c)@l, P \xrightarrow{\tau@l} \text{newchan}(\Delta) \vdash P[\text{chanref}(\Delta)/c] \]

\[ \text{card}(\text{linked}(\Delta, c)) = n \]

(A3) \[ \Delta \vdash \text{dock}(c, n)@l, P \xrightarrow{\tau@l} \Delta \vdash \text{dock}(c)@l, P \]

For any action \( \alpha \notin \{\text{spawn, new, dock}\} \)

(A4) \[ \Delta \vdash P \xrightarrow{\alpha} \Delta' \vdash P' \]

\[ \Delta \vdash P || Q \xrightarrow{\alpha} \Delta' \vdash P' || Q \quad P \equiv P' \]

\[ \Delta \vdash P \xrightarrow{\alpha} \Delta' \quad Q \]

we mean observable in the language. For instance, the \text{spawn} and \text{new} prefixes modify an Interaction Space, and as such possess, strictly speaking, an observable effect.

The axioms A1, A2, A3 and A4 describe the basic semantics for all silent and observable actions. As rule A4 states, the Interaction Spaces part of a term is notably invariant through observations. Only internal actions may involve contextual transformations. The basic axioms must be combined by the rules for the basic operators of the language: parallel (rule O1) and choice (rule O2). The structural congruence relation is implemented by rule O3. These definitions are rather classical in process algebras.

4.2. Reduction rules

The rules of Table 4 tell how silent steps or reductions can be inferred from the occurrence of observable actions in particular Interaction Spaces contexts. All the rules follow the same pattern that we may summarize as follows:

(A5) \[ \Delta \vdash P \xrightarrow{\text{obs}@l} P' \quad \text{cond}(\Delta) \] (Red)

\[ \Delta \vdash P \xrightarrow{\tau@l} \text{xform}(\Delta) \vdash P' \]

Each rule corresponds to an observation \text{obs} “made” by a given agent within an Interaction Space \( \Delta \). Under some geometrical condition \text{cond}, the rule may trigger a silent reduction (\( \tau \) step) associated to some transformation \text{xform} of the Interaction Space. An invariant condition (not shown in the rules) is that a channel \( c \) must be known at a location \( l \) in order to be manipulated. This invariant is written \( \text{known} \in \text{cget}(\Delta, l, c) \). For other conditions, consider for example rule S1 giving the semantics of the \text{dock} operations. It states that docking may only be performed by an agent at location \( l \) who knows the channel, and it may not be performed if the channel is already docked to another location, or if the agent is already linked to it. Undocking (rule S2) is only allowed at a location already docked to the channel. We must also ensure that all communications sent on the channel have been delivered. Rules S3 and S4 for linking operate in a similar way. The unlinking is only effected when no further message is to be received. An agent may perform an emission (rule C1) on a given channel only if it is docked to the location. Note that the communication is allowed even if no agent is linked to the channel, the message is of course lost in this case. Rule C2 describes the reception by an agent on a correctly linked channel. Of course it is only enabled if there is something to receive. The restrictions on communication enforce a non-interfering discipline in which agents may not meddle with other agents when communication occurs.

Agent mobility and remote cloning are implemented in the semantics by rules M1 and M2. We can see that physical locations are not taken into account. The virtual locations are chosen fresh to avoid interferences.
4.3. Behavioral equivalences

The LTS characterization of the language denotes a behavioral equivalence on terms. Such a mathematical construction is fundamental to compare and reason about agent behaviors. We first adapt the common notion of bisimulation to the particular setting of interaction spaces:

**Definition 8.** A (strong) bisimulation in Interaction Spaces is a symmetric relation $\sim$ between the LTS of two agents $\Delta \vdash A$ and $\Gamma \vdash B$ such that:

- $\Delta \vdash A \sim \Gamma \vdash B$ with $\Delta = \Gamma$, and
- if $\Delta \vdash A \xrightarrow{a} \Delta' \vdash A'$ then $\Gamma \vdash B \xrightarrow{a} \Gamma' \vdash B'$ and $\Delta' \sim \Gamma' \vdash B'$ with $\Delta' = \Gamma'$. 

**Strong bisimulation equivalence** $\Delta \vdash A \sim \Gamma \vdash B$ is the largest bisimulation between agents $\Delta \vdash A$ and $\Gamma \vdash B$.

The intuition is that two agents possess the same behavior iff they: (1) perform the same sequences of observable and silent actions, (2) possess the same branching structure, and (3) if put in the same spaces, they produce the same transformed spaces through reductions up to spatial equivalence. Bisimulation equivalences are not generally preserved by input prefixes in the case of name passing [16]. However, the language offers a clean separation between parallel composition, only available at the agent level, and sum composition at the process level. From this we may introduce separate agent and process contexts and congruences.

**Definition 9.** An agent context $C[X]$ is an agent $X$ put in parallel with some other agents. The grammar is $C[X] ::= X | X \parallel S$ where $S$ is a system of parallel agents.

**Definition 10.** A process context $C[X]$ is a context in which $X$ is part of a process. It is given by the syntax: $C[X] ::= X | \alpha@l.X + P$ where $P$ is a process expression. We write $C[X][A]$ for the context $C[X]$ within an agent $A$. 

---

Table 4

Operational semantics: reduction rules

\[
\begin{align*}
\Delta \vdash P \xrightarrow{\text{unlock}(c)@l} \Delta \vdash P' & \quad \text{write} \notin \mathcal{C}(\Delta, \mathcal{L}_\Delta, c) \quad \text{read} \notin \mathcal{C}(\Delta, l, c) \\
\Delta \vdash P \xrightarrow{\text{set}(\Delta, l, c, \text{write})} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{unset}(\Delta, l, c, \text{read})} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{set}(\Delta, l, c, \text{write})} \Delta \vdash P' & \quad \text{read} \in \mathcal{C}(\Delta, l, c) \quad \mathcal{A}_\Delta(\text{linked}(\Delta, c), c) = \emptyset \\
\Delta \vdash P \xrightarrow{\text{set}(\Delta, l, c, \text{read})} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{set}(\Delta, l, c, \text{write})} \Delta \vdash P' & \quad \text{write} \in \mathcal{C}(\Delta, l, c) \quad \mathcal{A}_\Delta(\text{linked}(\Delta, c), c) = \emptyset \\
\Delta \vdash P \xrightarrow{\text{set}(\Delta, l, c, \text{read})} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{send}(\Delta, c, v)} \Delta \vdash P' \quad \text{read} \in \mathcal{C}(\Delta, l, c) \quad \mathcal{A}_\Delta(l, c) \neq \emptyset \\
\Delta \vdash P \xrightarrow{\text{fetch}(\Delta, l, c)\{x\}} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{clone}(l_2)@l_1} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{move}(\Delta, l_1)} \Delta \vdash P' \quad \text{locref}(\Delta) = l_2 \\
\Delta \vdash P \xrightarrow{\text{clone}(l_2)@l_1} \Delta \vdash P' \\
\Delta \vdash P \xrightarrow{\text{move}(\Delta, l_1)} \Delta \vdash P' \quad \text{locref}(\Delta) = l_2 \\
\end{align*}
\]
A process congruence is an equivalence that preserves all process contexts. An agent congruence is an equivalence that preserves all agent contexts. It is relatively easy to show that strong bisimulation equivalence is both an agent congruence and a process congruence, which is detailed in [20]. We only outline here the part of the proof that specifically deals with input prefixes. In the pure $\pi$-calculus, it is easy to show the interleaving semantics property to hold:

$$c! (v) \parallel d? (x) \sim c! (v), d? (x) + d? (x), c! (v).$$

However, it does not hold anymore in the context of an input prefix that binds either $c$ or $d$, hence:

$$b? (d), [c! (v) \parallel d? (x)] \not\sim b? (d), [c! (v), d? (x) + d? (x), c! (v)].$$

In the left-hand side, a communication may occur if, for example, channel $c$ is received through $b$. This communication may not occur in the right-hand side because processes may only interact in parallel compositions in the $\pi$-calculus. Let us translate this in the framework of Interaction Spaces by considering a space $\Delta$ with, for some $l_2 \neq l_1$:

$$\Delta \vdash [\text{spawn}(d? (x), c! (v))]@l_1 \sim \Delta \vdash \tau@l_1, \left[ d? (x)@l_2, c! (v)@l_1 + c! (v)@l_1, d? (x)@l_2 \right].$$

We may ask if this result is preserved by input prefixes, for example by verifying that the following property holds for some $l', l_2 \neq l_1$:

$$\Gamma \vdash \lbrack b? (d), \lbrack \text{spawn}(d? (x)), c! (v)\rbrack\rbrack@l_1$$

$$\sim \Gamma \vdash b? (d)@l_1, \tau@l_1, \left[ d? (x)@l_2, c! (v)@l_1 + c! (v)@l_1, d? (x)@l_2 \right].$$

In fact, this is the case here because the agents do not communicate directly but through a common Interaction Spaces. For communication, the fact that we use a sum context instead of a parallel is thus not important.

The strong semantics exposes all the details of the agent behaviors. A useful variant is the weak semantics and a transitive closure of internal steps. Weak bisimulation equivalence is the largest symmetric relation $\approx$ between agents $A$ and $B$ such that:

- $\Delta \vdash A \approx \Gamma \vdash B$ with $\Delta = \emptyset$ $\Gamma$, and
- if $\Delta \vdash A \xrightarrow{\alpha} \Delta' \vdash A'$ then $\Gamma \vdash B \xrightarrow{\alpha} \Gamma' \vdash B'$ and $\Delta' \approx \Gamma' \vdash B'$ with $\Delta' = \emptyset$ $\Gamma'$
- if $\Delta \vdash A$ $\xrightarrow{\alpha} \Delta' \vdash A'$ then $\Gamma \vdash B$ $\xrightarrow{\alpha} \Gamma' \vdash B'$ and $\Delta' \approx \Gamma' \vdash B'$ with $\Delta' = \emptyset$ $\Gamma'$.

It is relatively easy to show that the weak equivalence is preserved by agent contexts. However, the process context $C[X] \cong X + c! v$ may be used to show that $\approx$ is not a process congruence. We trivially have $\Delta \vdash [c! w]@l \approx \Delta \vdash [\tau. c! w]@l$. However, we have $\Delta \vdash [C(P)]@l \equiv [c! w + c! v]@l$ and $\Delta \vdash [C(\tau. P)]@l \equiv [\tau. c! w + c! v]@l$ and as such $\Delta \vdash [C(P)]@l \not\approx \Delta \vdash [C(\tau. P)]@l$.

The weak semantics allows us to derive a weak normal form in which terms may be rewritten, simplifying the equational theory. We may for example establish the following lemma:

$$\Delta \vdash [\alpha, \text{new}(c)]@l \sim \Delta \vdash [\text{new}(c), \alpha]@l$$ for any action $\alpha$ with $c \not\in \alpha$.

The proof scheme for this lemma is as follows. Since the $\text{new}$ prefix infers silent transitions, it is not observable and thus absent from weak semantics. However, the effect of the new prefix is visible since it is recorded in the context. Since $c$ is not part of $\alpha$, the subspace in which the expansion is performed is independent of the subspace potentially affected by the performance of $\alpha$, which suffices to prove the lemma (see [20] for proof details).

As a simple generalization of the previous lemma we can prove that terms in the calculus accept a normal form which we write as follows:

$$\Delta \vdash \prod_l \lbrack \sum \text{new}(\bar{x}), P_l \rbrack@l_i$$ with $P_l$ free of $\text{new}$ actions and $\bar{x}$ fresh names.
Table 5
The libispace programming interface

<table>
<thead>
<tr>
<th>Operation</th>
<th>C function</th>
<th>Semantics</th>
<th>Projection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location creation</td>
<td>ispace_create_agent</td>
<td>newloc</td>
<td>(L_\Delta)</td>
</tr>
<tr>
<td>Channel creation</td>
<td>ispace_create_channel</td>
<td>newchan</td>
<td>(C_\Delta)</td>
</tr>
<tr>
<td>Acquisition</td>
<td>ispace_dock</td>
<td>cset</td>
<td>((l, l'), c))</td>
</tr>
<tr>
<td>Release</td>
<td>ispace_undock</td>
<td>cunset</td>
<td>((l, c))</td>
</tr>
<tr>
<td>Connection</td>
<td>ispace_link</td>
<td>cset</td>
<td>((l, l'), c))</td>
</tr>
<tr>
<td>Disconnection</td>
<td>ispace_unlink</td>
<td>cunset</td>
<td>((l, l'), c))</td>
</tr>
<tr>
<td>Emission</td>
<td>ispace_send</td>
<td>send</td>
<td>linked((\Delta, c))</td>
</tr>
<tr>
<td>Reception</td>
<td>ispace_receive</td>
<td>receive</td>
<td>snd((\Delta(l, c)))</td>
</tr>
<tr>
<td>Channel mobility</td>
<td>ispace_xfer</td>
<td>xfer</td>
<td>((l, l'), c))</td>
</tr>
<tr>
<td>Channel copy</td>
<td>ispace_xcpy</td>
<td>xcpy</td>
<td>((l, l'), c))</td>
</tr>
<tr>
<td>Migration</td>
<td>ispace_migrate</td>
<td>move</td>
<td>((l, l'), C_\Delta)</td>
</tr>
<tr>
<td>Remote cloning</td>
<td>ispace_clone</td>
<td>clone</td>
<td>((l, l'), C_\Delta)</td>
</tr>
</tbody>
</table>

Such a normal form is interesting at the formal level because it reduces the number of cases to consider in proofs based on structural induction on process terms. In practice, this lemma is also interesting since it allows pre-allocation of channels, an interesting source of optimizations at the implementation level.

5. Prototype implementations

We describe in this section an early implementation of the Interaction Spaces models and the associated coordination language. Following the decomposition of the model, the implementation is split in two basic components. The geometry of Interaction Spaces and the associated operations form the basic coordination services. They are implemented as a relatively low-level C library called the libispace. The library also integrates a server architecture and associated distributed runtime. It can be used directly in C programs as a generic and low-level coordination subsystem. The library may also be used to develop frontends for high-level programming languages. We describe the haspace Haskell frontend that implements the constructs proposed for the coordination language, and also integrates in a seamless way within the Glasgow Haskell compiler environment [9].

5.1. Libispace: A library and server architecture for Interaction Spaces

The libispace library and server architecture provides an implementation of the geometry of Interaction Spaces.

The main functions of the programming interface are summarized in Table 5. We make the connection with the geometry explicit by relating each library function to its semantic counterpart. We also describe the kind of projection that governs each function. Any projection involving only a couple \((l, c)\) of a location (agent) and channel is a local projection and may thus be performed efficiently. Since globally unique identifiers may be generated locally, the channel and agent (location) creation functions operate locally, and so do the unlinking and reception operations. Any projection or transformation involving more that one location is said distributed, which may involve more or less costly synchronizations. When two locations and a unique channel are concerned, this is a point-to-point synchronization. This concerns most of the mobility features, with always a departure and arrival location. Migration is of course most costly since it involves the movement of all attached channels. The mechanism we implement is described in [14]. The multicast semantics makes the implementation of the emission primitive quite demanding also. The current implementation only supports the TCP/IP protocol as communication backend forcing us to dispatch the emitted messages “by hand”. We plan to implement a support for a multicast backend in the near future. Finally, it is clear that no global server is needed to implement the Interaction Spaces, in that there is no projection or transformation involving the whole geometry.

In Table 6, we illustrate the use of the libispace API to implement the dynamic flowgraph example of Section 3.1.3. This is almost a direct use of the Interaction Spaces primitives except for the manual management of resources (most notably to free agents and channels). The library, also, does not support any explicit notion of scope. In order to
bootstrap the deployment\(^9\) and allow higher-level frontends to implement scoping rules, we provide minimal naming services through the `export` and `fetch_name` functions.

The library is available online [13]. The current size of the library is less that 4000 lines of C and about 150 kb when compiled statically on Linux/x86/gcc (21kb if stripped down).

### 5.2. The hspace Haskell frontend

In this section, we describe the `hspace` frontend to the `libispace` library. The library has been designed so that frontends could be implemented in any programming language environment supporting an even simple foreign function interface (FFI) for the C language. We decided to illustrate this aspect by developing a frontend in the realm of the Glasgow Haskell Compiler (GHC) architecture. This also demonstrates the separation of concerns between the coordination model (and its implementation as a C library) and the language constructs of the proposed language.

The first part of the frontend concerns the binding to the C library, for which we use the standard FFI model of the GHC environment. We obtain a set of monadic operators that allow standard Haskell programs to exploit the coordination features of the Interaction Spaces. The second part of the frontend concerns the support of the syntactic constructs of Table 1 with a dedicated parser and a simple type checker. The latter uses a simple unification algorithm. A type inference scheme is currently under investigation. The parallel and non-deterministic choice operators are most challenging. We use the STM model and library of *composable memory transactions* [11] to support “fair” non-determinism and avoid polling as well as busy waiting. The transaction mechanisms are particularly interesting to implement the semantics for the sum operator. In a process \([c?\langle\text{x}.\text{P}\rangle+\text{Q}\]`, a possible STM run will be to try to input on the channel \(c\) and backtrack to \(\text{Q}\) (or any of its sub-processes) if \(c\) is unavailable.

We now give a simple example of use for the `hspace` frontend. It is a simple distributed computation of the Ackerman function.

\[
def \text{Ack}(n:\text{int},p:\text{int},r:\text{<int>}) = 
\text{if } n=0 \text{ then } r!(p+1).\text{undock}(r)
\]

---

\(^9\) Obviously, when considering distributed systems, the first channel used by the processes may not be communicated through channel passing.
else if \( p=1 \) then \( \text{Ack}(n-1,1,r) \)
else new(r1:<int>),
    init(l'@remote),\{ dockn(r1,1),\text{Ack}(n,p-1,r1) \},
    link(r1),r1?(pp:int),unlink(r1),
    \text{Ack}(n-1,pp,r);

[ new(res:<int>),
    init(l'@remote) \{ dockn(res,1),\text{Ack}(3,7,res) \}
    link(res), res?(v:int),
    \#putStrLn("Result = "++ show $v)#,
    unlink(res) ] @ local

A syntax extension we recently added in the frontend allows us to configure a list of remote sites that can be selected randomly (and fairly) so that explicit physical locations are not manipulated directly. A limitation of the current implementation is that code migration is not implemented. We may rely on the hsplugins system to allow dynamic compilation of Haskell code from string representations. The frontend implementation, available online [13], represents about 2000 lines of Haskell code.

6. Related work

Coordination languages [3,8,19] establish the principles of (1) focusing on the interaction aspects of systems, and (2) abstract away from the their internal computations. In the recent years, coordination languages for mobile agent systems have been investigated. An important category is formed by endogenous, data-centric coordination models based on Linda-like tuple-space manipulation primitives [8] and extended for mobility. The coordination model proposed in Lime [17] makes both the virtual (i.e. dynamic binding/rebinding) and physical movement of agents appear as changes in the surround tuple-spaces. An expressive notion of reaction allows one to react when such a change occurs using a pattern matching facility. Lower-level primitives with explicit locations permit finer-grained control over the movement of agents. The latter are similar to what is provided in the Klaim language [18]. Agents may exchange locations and explicitly move to some location to interact there with the local tuple-space. Except for the tuple-space coordination primitives, the klaim primitives are more process-centric, and its underlying formal semantics are based on labeled transition systems. This makes our approach quite similar in spirit: both are kernel languages that may (must) be extended for practical use. The coordination model provided by Interaction Spaces is closer to actual distributed environments with the explicit modeling of communication channels. We thus see our approach as applying at a lower level of abstraction, and Linda-like coordination facilities could be modeled on top of Interaction Spaces. In the semantics, most approaches see mobility as a “play” with explicit locations, and very often higher-order traits are introduced to deal with agent migration (this is the case in both Lime and Klaim). Locations are purely virtual in Interaction Spaces, and we avoid their explicit use in agent behaviors, favoring a (more) virtual view of mobility from the channel perspective. We also avoid some second-order traits such as manipulating processes as data.

The explicit modeling of channels is closely related to exogenous coordination models such as Reo [4] and Mocha-\( \pi \) [10]. These can be described as “channel languages” that abstract away from the internal behavior (and internal coordination also) of agents (or components). The semantics for the (external) coordination primitives (e.g. send/receive) used by agents are defined externally. The primitive channels support various forms of communications: (a)synchronous, lossy, FIFO, etc. Complex “connectors” may be constructed by the composition of primitive channels and lower-level connectors, covering a large domain of communication semantics. In Mocha-\( \pi \), the channel semantics are expressed in a dedicated variant of the \( \pi \)-calculus, and both the physical and virtual movement of channel ends is supported. We see our approach as complementary: channels and associated semantics are considered as largely immutable and biased towards distributed environments. We focus on the coordinating behavior of agents evolving in such environments. More than opposing endogenous and exogenous coordination, we think that for fine-grained coordination behaviors, there is no such stability (or component structure) within the agents. Moreover, the distributed environment denotes physical channels with fundamental properties (e.g. asynchrony). But at a higher level of abstraction, we can consider stable organizations of agents (i.e. components) and, also, agents whose purpose
is solely to coordinate other agents (i.e. connectors). The example of the coordinator agent of section 2 is interesting here since its whole behavior is dedicated to coordination.

*Actor languages and models* [11] form another important category of approaches related to process-centric coordination with solid semantic foundations. Similarly to actors, we make a clear distinction between systems (we call them agent contexts) and programs (process contexts). From this point of view, the Interaction Spaces may also be seen as an extension of the actor configurations. The difference is that connections are implicit in actors whereas explicit channels are modeled in Interaction Spaces. We think that the latter provides an easier way to manage resources, most notably garbage collection (a topic we discuss in [21]).

From a process-algebra perspective, the agent calculus we propose may be seen as a distributed implementation of the $\pi$-calculus [16]. A refined view over channel mobility is proposed as we differentiate between communication of names (i.e. name passing), the movement of channel ends, and finally the migration of channel states. It is clear that the proposed formalization appears more complex than programming languages directly based on existing process algebras such as *nomadic-pict/nomadi-$\pi$* [23] and *Jocaml/Join-calculus* [7]. The separation between the coordination model – and its formal characterization as Interaction Spaces – and the LTS/bisimulation-based operational semantics may be seen as the principal reason for the increase in terms of complexity. However, our early meta-theoretic investigations exhibit nice *separation of concerns* properties, in that most proofs may be split in both purely geometric arguments and more traditional bisimulation-based cases. To some extents, the developed semantics provide a simpler treatment of some fundamental features of the $\pi$-calculus, most notably the issue of *freshness* that is here guaranteed by geometrical construction. Despite its asynchronous nature, our agent calculus is in its syntax closer to the standard $\pi$-calculus than its asynchronous variants [2,7]. For instance, the asynchronous calculi disallow output prefixing and most applied semantics also enforce *locality* constraints [15], which is not our case. This is discussed more thoroughly in [20]. The full externalization of the control (i.e. exogenous coordination) requires *objective* mobility features, such as moving an agent against its own will. In order to achieve this in Interaction Spaces, objective mobile behaviors must be re-encoded as subjective ones, a technique illustrated in [5]. Eventually, the non-trivial transformation makes more complex the formal characterization of behaviors and associated proof techniques.

Multicast semantics play a central place in the model, and provides in our opinion quite a large part of the language expressiveness. Very few channel-passing calculi supporting multicast have been proposed, a notable exception being the *broadcast* $\pi$-calculus [6]. Multicast interactions are generally built using a *subtractive* scheme, involving a *discard* relation for unwanted communications. Interaction Spaces, on the other hand, employ an *additive* scheme in which receptors are selected explicitly and dynamically. The advantage of the additive case is that the model is not specific for broadcast systems. Point-to-point interactions can be modeled by enforcing that a given channel may only be linked to one agent at a time. This is a simple geometrical constraint.

7. Conclusion

The model of Interaction Spaces proposes a geometrical characterization for mobile agent systems. An advantage of the geometrical point of view is that it offers an intuitive model, on which a basic denotation is easily constructed. Above the geometrical foundations, we design a coordination language that allows the description and composition of mobile agent behaviors. The model captures semantics that are of higher level than most mobile calculi. These semantics are also relatively complex but we show in the paper that the standard bisimulation techniques still apply. In a complementary way, we developed early implementations of the coordination language in a relatively short period. We thus think the model offers a good balance between *tractability* and *practicality*. Of course, we need more experiments with both the theory and practice of Interaction Spaces to strengthen this belief.

An interesting aspect of the model is its *modular* architecture. Most process algebras enforce a tight connection between the language constructs and the SOS semantics. The separate characterization of the coordination model allows various changes in the communication and mobility semantics to impact at that level. For example, a point-to-point synchronous channel can be built by constraining the size of the act dimensions (0 or 1 message only) and allowing only one link property. We plan to define a generalization of Interaction Spaces where channels (and, more generally subspaces) would be parameterized by geometrical constraints. Unlike the mobile ambients [5], we do not yet provide any hierarchical migration feature. However, we do think the model could be extended for hierarchical mobility by allowing spaces to nest within locations.
Type systems form another important aspect that we intend to investigate more thoroughly. The \textit{hspace} frontend of the language already provides a simple form of type checking based on the syntax illustrated in Section 2 and 5. The elaboration of a more expressive type system and type-checking (as well as inference) algorithm is left as a future work. We plan to build on existing typing framework for the \pi-calculus (e.g. \cite{22}), and also integrate elements of connection-time typing that we investigated in previous works \cite{14}. We also plan to integrate recent insights in type systems related to security and resource control, such as in \textit{safeDpi} \cite{12} and \textit{Klaim} \cite{18}. As a starting point, we think that the “colors” contained in Interaction Spaces could be reified as typing properties for resource control. Our current investigation is on the failure semantics with the dedicated \textit{lfail} (location failure), \textit{cfail} (channel failure) and \textit{kfail} (link failure). It is also interesting to investigate properties that may be fully captured in geometrical terms. For example, a communication deadlock may be characterized as a permanent “dot” in the geometry. Liveness issues may be also discussed in terms of geometrical invariants. Our current plan is to capture such invariants in terms of type-based analyses, from which we may derive verification mechanisms for the implementations.

References

\begin{enumerate}
\item Farhad Arbab, Coordination of mobile components, Electronic Notes in Theoretical Computer Science 54 (2001).
\item David Gelernter, Nicholas Carriero, Coordination languages and their significance, Communications of the ACM 35 (2) (1992) 96–107.
\item The Glasgow haskell compiler. \url{http://www.haskell.org/ghc}.
\item The interaction spaces project. \url{http://www-poleia.lip6.fr/~pesch/iss}.
\item Robin Milner, Communicating and Mobile Systems: The \pi-Calculus, Cambridge University Press, 1999.
\item Benjamin C. Pierce, Davide Sangiorgi, Typing and subtyping for mobile processes, Mathematical Structures in Computer Science 6 (5) (1996) 409–453.
\end{enumerate}