Quiet and Bouncing Objects: Two Migration Abstractions in a Simple Distributed Blue Calculus

Silvano Dal-Zilio[⋆]

INRIA Sophia-Antipolis

Abstract. In this paper, we study a model of *migrating objects* based on the *blue calculus* extended with a very simple system of localities and we show how two migration behaviors can be defined, namely those of bouncing and quiet objects. These "*migration control abstractions*" are defined separately from other aspects of the object definition and can be easily reused, thus providing more flexibility in the definition of "migration constraints".

1 Introduction

The purpose of this extended abstract is to study how the behavior of concurrent objects with respect to migration, can be defined orthogonally from other aspects of the object definition, such as synchronization constraint for example. To give the reader an intuition: many researches have been conducted on the problem of defining *synchronization abstractions* for concurrent objects [10, 7], likewise, we are interested here in the definition of migration abstractions that can be reused to implement distributed objects. To this end, we give two examples of objects that act, respectively, according to the well-known client/server and agent paradigms.

We first present the calculus used to define objects, namely a version of the blue calculus [4] enriched with a simple model of localities so that we can deal with migration. In this distributed blue-calculus ($\mathbf{D}\boldsymbol{\pi}^{\star}$), objects can be represented "as processes" [8]. In particular, we study in Sect. 3, the canonical example of the mutable cell. Then we show, using a slight modification in the process definition, how one can mimic two migration behaviors: the object that always resides at the same location and the object that migrates to the location of its clients.

2 The Calculus

The blue calculus ($\boldsymbol{\pi}^{\star}$) is a direct extension of both the λ and the π calculi. In this paper, we consider a very simple distributed version of the original calculus introduced in [4] (see Table 1) obtained by adding locations, located processes ($[a :: P]$) and a primitive for code transfer ($\mathbf{go}\ a.P$).

In $\mathbf{D}\boldsymbol{\pi}^{\star}$, terms are defined using three disjoint kinds of names: references: $u, v, w \cdots \in \mathcal{R}$, labels: $k, l, m \cdots \in \mathcal{L}$ and localities (or places) $a, b, c \cdots \in \mathcal{P}$. The definition $\mathbf{def}\, D\, \mathbf{in}\, P$ (where D is a sequence of mutually recursive definitions $u_1 = P_1, \ldots, u_n = P_n$, with the u_i's pairwise distinct) is a restricted an replicated version of the declaration $\langle u \Leftarrow P \rangle$, that can be understood as a resource, located at u, accessible only once[1]. An important restriction

[⋆] Email: Silvano.Dal_Zilio@sophia.inria.fr. Address: INRIA Sophia-Antipolis, BP 93, 2004 route des Lucioles. F-06902 Sophia Antipolis cedex. Fax: (+33) 492.38.79.98

[1] the declaration $\langle u \Leftarrow (\boldsymbol{\lambda} x)P \rangle$ is the equivalent of the π-calculus input guard $u(x).P$. This construct is useful to model processes with a mutable state.

imposed over terms is that no declaration can be defined on an abstracted reference (e.g., $(\boldsymbol{\lambda} u)\langle u \Leftarrow P\rangle$ is not a valid process). This restriction, equivalent in $\boldsymbol{\pi}$ to the one that forbids reception on received names, ensures that no new declaration on a given reference can be dynamically created.

Table 1 Syntax of the Blue Calculus with Simple Location System: $\mathbf{D}\boldsymbol{\pi}^{\star}$

x	$::=$	$u \mid a$	values
P	$::=$	$0 \mid (P \mid P) \mid (\boldsymbol{\nu} u)P \mid \mathbf{go}\ a.P \mid$	processes
		$u \mid (\boldsymbol{\lambda} x)P \mid (P\ x) \mid$	agents
		$\langle u \Leftarrow P\rangle \mid \mathbf{def}\ u_1 = P_1, \ldots, u_n = P_n\ \mathbf{in}\ P \mid$	declarations
		$[\,l_i = P_i{}^{1 \leq i \leq n}\,] \mid (P \cdot l)$	records
S	$::=$	$[a :: P] \mid (S \mid S) \mid (\boldsymbol{\nu} u)S$	locations

The model chosen to deal with distribution is very simple. A *site* is a named box, $[a :: P]$, containing a running threads P. For the sake of simplicity, we consider a flat system of locations (that is located processes are not nested) as in [9, 2]. We consider also the operation: $\mathbf{go}\ a.P$, that spawns a thread P in the location a and we denote $u@a$ the process $\mathbf{go}\ a.u$ that sends a message at location a. The reduction semantics of $\mathbf{D}\boldsymbol{\pi}^{\star}$ is given in a chemical style [3] and uses a structural equivalence ($\equiv$). The definition of the reduction relation ($\rightarrow$) uses the standard notion of *evaluation context* ($\mathrm{E}\,[]$), i.e. contexts such that the hole does not occur under a guard[2]. The definition includes three general rules:

$$\frac{Q \equiv P \quad P \rightarrow P'}{Q \rightarrow P'} \qquad \frac{P \rightarrow P'}{\mathrm{E}\,[P] \rightarrow \mathrm{E}\,[P']} \qquad \frac{P \rightarrow P'}{[a :: P] \rightarrow [a :: P']}$$

We refer the reader to [4, 8] for a full presentation of the reduction semantics for the calculus without localities. Axioms for structural equivalences are the usual axioms for the π-calculus (including scope extrusion) extended with rules to manage application. We have omitted the rules for record selection, $(P \cdot l)$, that acts like application and we refer the reader to [5] for details. We also add two new axioms in the distributed calculus to allow spawning of restricted names over locations: $(\imath)$ to distribute references restricted by a $(\boldsymbol{\nu} u)P$ statement, and $(\jmath)$ to distribute definition over parallel composition. Note that the equivalent of relation $(\jmath)$ in the π-calculus, is obtained by using a behavioral equivalence. This is the the well-known replication theorem of [11, 12] in the case of channels with output-only capabilities.

$$\begin{array}{ll}
\mathbf{def}\ D\ \mathbf{in}\,(\mathbf{def}\ D'\ \mathbf{in}\ P) \equiv \mathbf{def}\ D, D'\ \mathbf{in}\ P & (\mathbf{def}\ D\ \mathbf{in}\ P)\ x \equiv \mathbf{def}\ D\ \mathbf{in}\,(P\ x) \\
(P \mid Q)\ x \equiv (P\ x) \mid (Q\ x) & (\langle v \Leftarrow P\rangle\ x) \equiv \langle u \Leftarrow P\rangle \\
(\mathbf{go}\ a.P)\ x \equiv \mathbf{go}\ a.(P\ x) & \mathbf{def}\ D\ \mathbf{in}\,(\mathbf{go}\ a.P) \equiv \mathbf{go}\ a.(\mathbf{def}\ D\ \mathbf{in}\ P) \\
(\imath) \quad [a :: (\boldsymbol{\nu} u)P] \equiv (\boldsymbol{\nu} u)([a :: P]) & (\jmath) \quad \mathbf{def}\ D\ \mathbf{in}\,(P \mid Q) \equiv (\mathbf{def}\ D\ \mathbf{in}\ P \mid \mathbf{def}\ D\ \mathbf{in}\ Q)
\end{array}$$

[2] $\mathrm{E}\,[] ::= [.] \mid (\mathrm{E}\,[]\ x) \mid (\mathrm{E}\,[] \mid P) \mid (P \mid \mathrm{E}\,[]) \mid (\boldsymbol{\nu} u)(\mathrm{E}\,[]) \mid \mathbf{def}\ D\ \mathbf{in}\ \mathrm{E}\,[] \mid (\mathrm{E}\,[] \cdot l)$

The reduction relation embeds different mechanisms. "Small" β reduction (1), definition folding (2), resource fetching (3) and record selection (4):

$$\begin{array}{ll}
(1)\ (\boldsymbol{\lambda} x)P\ y \ \rightarrow\ P\{y/x\} & \\
(2)\ \textbf{def}\, D, u = R, D' \,\textbf{in}\, \mathrm{E}\,[u\ x_1 \ldots x_n] \ \rightarrow\ \textbf{def}\, D, u = R, D' \,\textbf{in}\, \mathrm{E}\,[R\ x_1 \ldots x_n] & (u \notin \mathrm{bn}(\mathrm{E})) \\
(3)\ \langle u \Leftarrow P\rangle \mid u\ x_1 \ldots x_n \ \rightarrow\ P\ x_1 \ldots x_n & \\
(4)\ [\, l = P\,,\, Q\,] \cdot l \ \rightarrow\ P \quad \text{and} \quad [\, l = P\,,\, Q\,] \cdot k \ \rightarrow\ Q \cdot k & (k \neq l)
\end{array}$$

We also add the reduction rule for the **go** statement (note that process P cannot execute under the guard **go** $a.P$)

$$(5) \quad [b :: (\textbf{go}\ a.P \mid Q)] \mid [a :: R] \ \rightarrow\ [b :: Q] \mid [a :: (P \mid R)]$$

Example 1. A typical reduction sequence in $\mathbf{D}\boldsymbol{\pi}^\star$ is the one such that a message carrying a private reference is send remotely.

$$\begin{array}{rl}
\begin{pmatrix} [b :: \textbf{def}\, u = R \,\textbf{in}\, (v@a\ u \mid P)] \\ \mid [a :: \langle v \Leftarrow Q\rangle] \end{pmatrix} & \equiv \begin{pmatrix} [b :: \textbf{go}\ a.(\textbf{def}\, u = R \,\textbf{in}\, v\ u) \mid \textbf{def}\, u = R \,\textbf{in}\, P] \\ \mid [a :: \langle v \Leftarrow Q\rangle] \end{pmatrix} \\
& \rightarrow^* \begin{pmatrix} [b :: \textbf{def}\, u = R \,\textbf{in}\, P] \\ \mid [a :: \textbf{def}\, u = R \,\textbf{in}\, (Q\ u)] \end{pmatrix} \qquad (\textbf{if}\ u \notin \mathrm{fn}(Q))
\end{array}$$

To conclude, let us just state that, while the blue calculus is a *name passing* calculus (that is a process can only be applied to a name and not to another process), the "high-order" λ-calculus application can be recovered using the definition:

$$(P\ Q) =_{\mathrm{def}} \textbf{def}\, u = Q \,\textbf{in}\, (P\ u) \qquad (u \notin \mathrm{fn}(P) \cup \mathrm{fn}(Q))$$

Moreover this definition of application is coherent with our model of distribution since we can prove that $(\textbf{go}\ a.P)\ Q \equiv \textbf{go}\ a.(P\ Q)$.

3 Modeling Objects in the Blue Calculus

In this extended abstract, we will concentrate on a single example of object, namely the "mutable cell". Although it is only an example, it is a representative one, since in [8] we show how to derived a "complete" calculus of concurrent objects using cells and extensible records. Thus, the result given for the cell example can be derived for more general objects. The constructs of this object calculus, together with its derived operational rules, are given for information in Sect. 3 (consideration on types are omitted). Let $\mathrm{R}_o(b)$ denotes the record:

$$\mathrm{R}_o(b) \ =_{\mathrm{def}} \ \big[\, get = (\boldsymbol{\lambda} x)(o\ b \mid x\ b),\ put = (\boldsymbol{\lambda} x)(o\ x) \big]$$

The cell process with "name " O is defined by:

$$\textsc{Cell}(O) \ =_{\mathrm{def}}\ \textbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\textbf{in}\, o$$

and the application: $(\textsc{Cell}(O)\ a_0)$, initializes the cell to the value a_0. It is easy to see that[3] $(\textsc{Cell}\ a_0 \mid O \cdot get\ r)$ and $(\textsc{Cell}\ a_0 \mid O \cdot put\ a)$ evaluate in a deterministic way:

$$\begin{array}{rl}
(\textsc{Cell}\ a_0) \mid (O \cdot get\ r) & \rightarrow\ \textbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\textbf{in}\, (\langle O \Leftarrow \mathrm{R}_o(a_0)\rangle \mid O \cdot get\ r) \\
& \rightarrow\ \textbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\textbf{in}\, (\mathrm{R}_o(a_0) \cdot get\ r) \\
& \rightarrow^*\ \textbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\textbf{in}\, (o\ a_0 \mid r\ a_0)\ \equiv\ (\textsc{Cell}\ a_0) \mid r\ a_0
\end{array}$$

$$(\textsc{Cell}\ a_0) \mid (O \cdot put\ a) \ \rightarrow^*\ \textbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle \,\textbf{in}\, (o\ a)\ \equiv\ (\textsc{Cell}\ a)$$

[3] to simplify the examples, we use $\textsc{Cell}$ to denote $\textsc{Cell}(O)$

It is interesting to notice the linear use of the reference O in $(\textsc{Cell}(O)\ a)$. If the cell is invoked, we consume the unique declaration $\langle O \Leftarrow \mathrm{R}_o(a)\rangle$. Thus, a unique message $(o\ a')$, acting like a lock, is freed in the evaluation process, which, in turn, frees a single declaration $\langle O \Leftarrow \mathrm{R}_o(a')\rangle$. Thus, we have the invariant that there is exactly one resource available at address O, and that this resource keeps the last value passed in a $(O \cdot put)$ call.

4 A Concurrent Calculus of Objects

More elaborate objects than the (canonical) example of the mutable cell can be defined. In this section, we introduce a calculus of concurrent objects by specifying a set of operators and their operational semantics, and we define these operators (and their associated reduction rules) with an encoding in $\boldsymbol{\pi}^\star$. In the specification of this calculus (Table 2), we distinguish a subset $\mathcal{O}$ of references (which we call objects names, $O, A, B, \dots \in \mathcal{O}$) and we use L to denote an "object body": $L = l_1 = \varsigma(x_1)P_1, \dots, l_n = \varsigma(x_n)P_n$.

Table 2 Specification of Operators and Reduction Rules for Objects in $\boldsymbol{\pi}^\star$

$\varsigma(x)P$	method with self parameter x and body P
$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i)P_i^{\,1\le i\le n} \right\} \mathbf{in}\ P$	object with n methods $l_1, \dots, l_n$
$P \Leftarrow l$	invocation of method l
$O \leftarrow l = \varsigma(x)Q$	update of method l with body $\varsigma(x)P$
$\mathbf{clone}(O)$	cloning of object O

Let $L =_{\mathrm{def}}\ l_1 = \varsigma(x_1)P_1, \dots, l_n = \varsigma(x_n)P_n$

$$\mathbf{obj}\, O = \{L\}\mathbf{in}\ \mathrm{E}\,[O \Leftarrow l_j] \ \rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{L\}\mathbf{in}\ \mathrm{E}\,[P_j\{O/x_j\}]$$

$$\mathbf{obj}\, O = \{L\}\mathbf{in}\ \mathrm{E}\,[O \leftarrow l_j = \varsigma(x)P] \ \rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{l_j = \varsigma(x)P, l_i = \varsigma(x_i)P_i^{\,i\neq j}\}\mathbf{in}\ \mathrm{E}\,[O]$$

$$\mathbf{obj}\, O = \{L\}\ \mathbf{in}\ \mathrm{E}\,[\mathbf{clone}(O)] \ \rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{L\}\ \mathbf{in}\,(\mathbf{obj}\, A = \{L\}\ \mathbf{in}\ \mathrm{E}\,[A]) \quad (A \notin \mathrm{bn}(\mathrm{E}))$$

An example of object is the one that produces an infinite copy of itself. Let L be the body: $l = \varsigma(x)(\mathbf{clone}(x) \mid x \Leftarrow l)$, then:

$$\mathbf{obj}\, O = \{L\}\,\mathbf{in}\ O \Leftarrow l \ \rightarrow^*_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{L\}\,\mathbf{in}\,(\mathbf{obj}\, A = \{L\}\,\mathbf{in}\,(A \mid O \Leftarrow l)) \ \rightarrow^*_{\pi^\star_\varsigma}\ \dots$$

another example, using method overriding, is:

$$\begin{aligned}\mathbf{obj}\, O = \{L\}\,\mathbf{in}\,(O \leftarrow l = \varsigma(x)x) \Leftarrow l \ &\rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{l = \varsigma(x)x\}\,\mathbf{in}\ O \Leftarrow l \\ &\rightarrow_{\pi^\star_\varsigma}\ \mathbf{obj}\, O = \{l = \varsigma(x)x\}\,\mathbf{in}\ O\end{aligned}$$

4.1 Interpretation of the Derived Object Constructs

Processes modeling objects are inspired from the encoding of the mutable cell. In the definition given in Table 3, an object $(\mathbf{obj}\, O = \{l_i = \varsigma(x_i)P_i^{\,1\le i\le n}\}\mathbf{in}\ P)$, is a cell with an additional field $clone$ to allow object cloning. In this definition, a method $\varsigma(x)P$ is an abstraction $(\boldsymbol{\lambda}x)P$. This function, also called premethod, has one argument: the name of the current object (also called the self-parameter). The cell "memorizes" a record of premethods:

$[\, l_i = (\boldsymbol{\lambda} x_i) P_i{}^{1 \le i \le n}\,]$, as in the classical recursive records semantics [6]. Note that we restrict the scope of an object name to the definition of the object it refers to. Thus we have the invariant that there is a unique declaration for each object names. Moreover, method update returns "a reference" to the modified object. This is almost the behaviour of (sequential) objects in the ς calculus of Abadi and Cardelli [1].

$$\mathsf{S}_o(b) \;=_{\text{def}}\; \begin{bmatrix} \mathit{get} = (\boldsymbol{\lambda} x)(o\, b \mid x\, b\, O), \\ \mathit{modify} = (\boldsymbol{\lambda} x)(x\, o\, b \mid O), \\ \mathit{clone} = (o\, b \mid x_{\mathit{clone}}\, b) \end{bmatrix}$$

$$\textsc{obj}(O) \;=_{\text{def}}\; \mathbf{def}\, o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathsf{S}_o(b)\rangle\, \mathbf{in}\, o$$

Table 3 Definition of the Derived Operators for Objects

$$P \Leftarrow\!\!\!\dashv l =_{\text{def}} (P \cdot \mathit{get}\ (\boldsymbol{\lambda} x)(x \cdot l)) \qquad\qquad \mathbf{clone}(O) =_{\text{def}} (O \cdot \mathit{clone})$$

$$O \leftarrow l = \varsigma(x)Q =_{\text{def}} (O \cdot \mathit{modify}\ (\boldsymbol{\lambda} ob)(o\,[\, l = (\boldsymbol{\lambda} x)Q\,,\, b\,]))$$

$$\mathbf{obj}\, O = \left\{ l_i = \varsigma(x_i) P_i{}^{1 \le i \le n} \right\} \mathbf{in}\ P =_{\text{def}} \begin{cases} \mathbf{def}\ x_{\mathit{clone}} = (\boldsymbol{\lambda} b)(\boldsymbol{\nu} A)(\textsc{obj}(A)\ b \mid A) \\ \mathbf{in}\ (\boldsymbol{\nu} O)((\textsc{obj}(O)\ [\, l_i = (\boldsymbol{\lambda} x_i) P_i{}^{1 \le i \le n}\,]) \mid P) \end{cases}$$

Another remark is that we use only field selection and application in the definition of cloning, method invocation and method update. Thus the definition of structural equivalence allows us, for example, to derive the following laws, showing that these (derived) operators acts like application:

$$(\mathbf{def}\, D\, \mathbf{in}\, P \mid Q) \Leftarrow\!\!\!\dashv l \;\equiv\; \mathbf{def}\, D\, \mathbf{in}\, (P \Leftarrow\!\!\!\dashv l \mid Q \Leftarrow\!\!\!\dashv l) \qquad (\langle u \Leftarrow P\rangle \Leftarrow\!\!\!\dashv l) \;\equiv\; \langle u \Leftarrow P\rangle$$

The next result states that there is an operational correspondence between $\rightarrow_{\pi^\star_\varsigma}$ (defined in Table 2) and $\rightarrow$. These properties are proved using a simple induction.

Theorem 2 (Operational Equivalence). *The specification of the object reduction rules is* complete *with respect to the encoding of objects in* $\boldsymbol{\pi}^\star$*:* $P \rightarrow_{\pi^\star_\varsigma} P' \Rightarrow P \rightarrow^* P'$*. It is also* sound*:* $P \rightarrow Q$ *implies that* $Q \rightarrow^* Q'$ *with* $P \rightarrow_{\pi^\star_\varsigma} Q'$*.*

5 Quiet Versus Bouncing Cells

In our model of distribution, synchronization (rule (3)) does not extend over location boundaries. Thus communication is local, and, to interact with a remote declaration at location a, one has to first spawn a message at a. For example, the process $([b :: u] \mid [a :: \langle u \Leftarrow P\rangle])$ is inert while $([b :: u@a] \mid [a :: \langle u \Leftarrow P\rangle])$ reduces to $([b :: \mathtt{0}] \mid [a :: P])$. Another remark is that the result of the communication is always executed at the location of the declaration. This is reminiscent of the client-server paradigm of computation such that clients "controls" the computation, that takes place at the server location, by sending remote messages.

The "migration behavior" of the mutable cell object and of the declaration $\langle O \Leftarrow \mathrm{R}_o(b)\rangle$ are equal. For example, to read the content of the mutable cell from a remote location, one has to (1) move to the location of the cell, then (2) invoke the method read and finally (3) fetch the result back. After completion, the cell is still at the same location.

$$\begin{pmatrix} [a :: (\textsc{Cell}\ a_0) \mid P] \mid \\ [b :: (O@a \cdot get\ r@b) \mid Q] \end{pmatrix} \begin{array}{l} \to^* [a :: (\textsc{Cell}\ a_0) \mid (r@b\ a_0) \mid P] \mid [b :: Q] \\ \to^* [a :: (\textsc{Cell}\ a_0) \mid P] \mid [b :: (r\ a_0) \mid Q] \end{array}$$

Likewise, objects created using the cell also share the same "migration behavior", i.e. they are *quiet* objects since they never leave the location of their creation.

In this section, we define a new migration abstraction, namely the *agent* behavior, based on a new declaration statement: $\langle u \Leftarrow P\rangle_{\mathrm{agent}}$ (see Tab. 5), that can be derived in $\mathbf{D}\boldsymbol{\pi}^\star$. Roughly speaking, the result of a remote communication involving $\langle u \Leftarrow P\rangle_{\mathrm{agent}}$ takes place at the client location instead of the declaration one.

Table 4 Two Distributed Cells

$$\textsc{Cell}(O) =_{\mathrm{def}} \mathbf{def}\ o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle_{\mathrm{c/s}}\ \mathbf{in}\ o$$

$$\textsc{AgtCell}(O) =_{\mathrm{def}} \mathbf{def}\ o = (\boldsymbol{\lambda} b)\langle O \Leftarrow \mathrm{R}_o(b)\rangle_{\mathrm{agent}}\ \mathbf{in}\ o$$

The operational semantics of an agent declaration strongly depends on the distribution of the processes. Indeed in $\boldsymbol{\pi}^\star$ (and in $\boldsymbol{\pi}$) there are no explicit locations and (therefore) *where* a communication comes from is not observable. But this information does count in a distributed setting. If we redefine the cell object using $\langle O \Leftarrow [\ldots]\rangle_{\mathrm{agent}}$ instead of $\langle O \Leftarrow [\ldots]\rangle$, denoting it $\textsc{AgtCell}$, we obtain a mutable cell bouncing from locations to locations according to communications with the client.

$$\begin{pmatrix} [a :: (\textsc{AgtCell}\ a_0) \mid P] \mid \\ [b :: (O@a \cdot get\ r) \mid Q] \end{pmatrix} \to^* \begin{pmatrix} [a :: P] \mid \\ [b :: (\textsc{AgtCell}\ a_0) \mid (r\ a_0) \mid Q] \end{pmatrix}$$

to sketch the derivation of this new construct, let us just say that remote messages $[b :: u@a]$ are translated to $[b :: \mathbf{go}\ a.(u\ b)]$ (that is the same message with, as extra argument, the "departure location"), and that the declaration $\langle u \Leftarrow P\rangle_{\mathrm{agent}}$ and $\langle u \Leftarrow P\rangle_{\mathrm{c/s}}$ are defined by:

$$\begin{array}{lll} \langle u \Leftarrow P\rangle_{\mathrm{agent}} & =_{\mathrm{def}} \langle u \Leftarrow (\boldsymbol{\lambda} a)(\mathbf{go}\ a.P)\rangle & \\ \langle u \Leftarrow P\rangle_{\mathrm{c/s}} & =_{\mathrm{def}} \langle u \Leftarrow (\boldsymbol{\lambda} a)P\rangle & (\text{with } a \notin \mathrm{fn}(P)) \end{array}$$

Other object migration abstractions can be uniformly defined by first defining a new kind of declaration. For example, an *applet* object can be interpreted as an object that does not change location but that spawn a copy (or clone) of itself at the location of its client. We can give this behavior to an object using, for its definition, a new declaration construct,

$\langle u \Leftarrow P \rangle_{\text{applet}}$ (see Tab. 5), that can also be derived in $\mathbf{D}\boldsymbol{\pi}^{\star}$. For example $\langle u \Leftarrow P \rangle_{\text{applet}}$ can be translated to:

$$\langle u \Leftarrow P \rangle_{\text{applet}} =_{\text{def}} \mathbf{def}\, x = \langle u \Leftarrow (\boldsymbol{\lambda} a)(\mathbf{go}\ a.P \mid x)\rangle \,\mathbf{in}\, x$$

Table 5 Client/Server, Agent and Applet Communications

$$\begin{pmatrix} [a :: \langle u \Leftarrow P \rangle_{\text{c/s}} \mid R] \mid \\ [b :: (\mathbf{def}\, D\, \mathbf{in}\, u@a\ v_1 \ldots v_n) \mid Q] \end{pmatrix} \to^* \begin{pmatrix} [a :: (\mathbf{def}\, D\, \mathbf{in}\, P\ v_1 \ldots v_n) \mid R] \mid \\ [b :: Q] \end{pmatrix}$$

$$\begin{pmatrix} [a :: \langle u \Leftarrow P \rangle_{\text{agent}} \mid R] \mid \\ [b :: (\mathbf{def}\, D\, \mathbf{in}\, u@a\ v_1 \ldots v_n)) \mid Q] \end{pmatrix} \to^* \begin{pmatrix} [a :: R] \mid \\ [b :: (\mathbf{def}\, D\, \mathbf{in}\, P\ v_1 \ldots v_n) \mid Q] \end{pmatrix}$$

$$\begin{pmatrix} [a :: \langle u \Leftarrow P \rangle_{\text{applet}} \mid R] \mid \\ [b :: (\mathbf{def}\, D\, \mathbf{in}\, u@a\ v_1 \ldots v_n)) \mid Q] \end{pmatrix} \to^* \begin{pmatrix} [a :: \langle u \Leftarrow P \rangle_{\text{applet}} \mid R] \mid \\ [b :: (\mathbf{def}\, D\, \mathbf{in}\, P\ v_1 \ldots v_n) \mid Q] \end{pmatrix}$$

6 Conclusions and Future Work

A challenging problem encountered in the design of programming languages with concurrent objects, is to be able to express the synchronization control of objects in a compositional and reusable way. Now that "network-oriented" languages have added functionalities to remotely download code and to migrate objects, a similar problem arises in the description of the migration behavior.

In the present paper, we have presented how two abstractions for the definition of the migration behavior can be applied to define mutable cell objects that react differently when invoked by a remote client. Using a translation of a calculus of concurrent objects in $\boldsymbol{\pi}^{\star}$ defined in [8], we claim that those abstractions can be uniformly applied to every object creation. Moreover, These abstractions can be transposed to other process calculi, and in particular to distributed versions of the π-calculus [14, 9, 2], and to other interpretation of objects [13].

The principal advantage of defining "standardized behaviors of migration", is that we can define migrating objects from objects designed in an non-distributed language without adding any explicit thread of control, thus providing a flexible and simple way to automatically add migration features to objects. But further works must be done to define more elaborated behaviors than those presented in this abstract. For example it will be interesting to give an accurate model of the mobile agents behavior as defined in TELESCRIPT [15].

References

1. Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 2(125):78–102, March 1996.
2. Roberto Amadio. An asynchronous model of locality, failure and process mobility. In *COORDINATION 97*, volume 1282 of *Lecture Notes in Computer Science*, 1997.
3. G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
4. Gérard Boudol. The π-calculus in direct style. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 228–241, Paris, France, 15–17 January 1997.
5. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *ASIAN'97, the Asian Computing Science Conference*, Lecture Notes in Computer Science, Kathmandu, December 1997.
6. L. Cardelli and J. C. Mitchell. Operations on records. *Math. Structures in Computer Science*, 1(1):3–48, 1991.
7. Denis Caromel. Programming abstractions for concurrent programming. In *Technology of Object-Oriented Languages and Systems (TOOLS Pacific'90)*, pages 245–253, Sydney, November 1990.
8. Silvano Dal-Zilio. Concurrent objects in the blue calculus. (draft) `http://www.inria.fr/meije/personnel/Silvano.Dal_Zilio/`.
9. Matthew Hennessy and James Riely. Ressource access control in systems of mobile agents. Technical Report 2/98, University of Sussex, February 1998.
10. Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, chapter 4, pages 107–150. The MIT Press, 1993.
11. Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, University of Edinburgh, 1991.
12. Benjamin Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 11, 1995.
13. Davide Sangiorgi. An interpretation of typed objects into typed π-calculus. Technical Report 3000, INRIA, 1996.
14. Peter Sewell. Global/local subtyping for a distributed pi-calculus. Technical Report 435, University of Cambridge, 1997.
15. Jim White. Mobile agent white paper. Technical report, General Magic, 1996.