Some results in the Join-Calculus

Jean-Jacques Lévy *

INRIA Rocquencourt

Abstract. The join-calculus is a model for distributed programming languages with migratory features. It is an asynchronous process calculus based on static scope and an explicit notion of locality and failures. It allows standard polymorphic ML-like typing and thus to be integrated with a realistic programming language. It has a distributed implementation on top of the Caml language. We review here some of the results recently obtained in the join-calculus.

1 Introduction

The join-calculus, defined by Fournet and Gonthier[11], is a core calculus for the design of a functional language which allows to program mobile applications. In another paper [12], we presented a version of the calculus with explicit locations and a failure model, which is the basis of an on-going distributed implementation. (The release, together with a tutorial, is available at [14]). Moreover, an ML-style typing theory may be developed for the join-calculus [13] with polymorphism and distinction between synchronous and asynchronous names.

Currently, there are several languages for programming mobile applications: Emerald [6], Obliq [8], Facile [15], Telescript [27], Java. Only Facile [17] has a formal semantics based on the pi-calculus and a chemical machine. Emerald and Obliq are object-oriented, we think simpler to address the design of a language without the burden of objects. Facile and Telescript are similar to our model, but our claim is that the join-calculus is a simpler basis for distributed implementations.

As a general observation, it is very difficult to implement in a distributed fashion languages based on CCS or the pi-calculus, even in their asynchronous versions. The main obstacle is that a channel resides in the ether, and therefore is not located. If a message is sent to an unlocated port, one comes very quickly to solve a distributed consensus for nearly every communication, since two receptors on a same channel have to agree in order to take the value and thus to prevent the other from getting the same message. With the join-calculus, we avoid this problem by forcing the receptors of a given channel to reside at a same location.

Before introducing localizations of channels, we need to expose the core join-calculus, without any notion of physical locations or of failures. The idea is that it is important to first depart from the pi-calculus in order to be able to make the second step of our presentation with locations. Along the discussion of the join-calculus we will have to keep in mind that some of the restrictions or grouping of

* This work is partly supported by the ESPRIT CONFER-2 WG-21836

operations are motivated by this concern of having a feasible and direct distributed implementation.

The join-calculus can be seen as a variant of the asynchronous pi-calculus [19, 18, 7, 2], which is at the same time more atomic and more broad for some operations. It is also suited to the integration into a functional language such as ML. The main differences with the pi-calculus come from three points: the uniqueness of the localization of receptors, the permanent state of receptors, the definition of receptors. We now review this three differences, which a priori may seem minor, but which have a strong effect.

In the pi-calculus, or in the Pict language [22, 23], where there is no notion of localization, it is possible to write

$$x(y).P \mid x(z).Q \mid \overline{x}a$$

(in the pi-calculus) or

$$x?y.P \mid x?z.Q \mid x!a$$

(in Pict whose syntax is more à la CCS with value-passing) without having in mind where the channel x resides. Therefore, it could be intricate to make the communication in the previous example, since the two receptors $x(y).P$ and $x(z).Q$ may be far from one another in a pi-term. In the join-calculus, the syntax forces the receptors of a given channel to be at the same "place" in an expression. One writes:

$$\texttt{def}(x\langle y\rangle \triangleright P) \wedge (x\langle z\rangle \triangleright Q) \texttt{ in } x\langle a\rangle$$

In fact, as `def` opens a new scope for x, the analogous in the pi-calculus is:

$$(\nu x)(x(y).P \mid x(z).Q \mid \overline{x}a)$$

The scope discipline forces all the receptors on a given channel to be defined on the left hand side of a unique `def` expression. This is a first step towards the uniqueness of the location on which a channel may reside. In this example, one should also notice the overloading of the notation $x\langle y\rangle$ in a term $\texttt{def}\, D \texttt{ in } R$. It means reception of y on x in the D-part (definition part), and emission of y on x in the R-part (body part). This will be convenient when defining the operational semantics, and is reminiscent of functional definitions and pattern matching.

As for functions in a normal programming language, channel receptors are permanently defined. There are no single-shot reception processes such as $x(y).P$ in the pi-calculus. Join-calculus channels are permanent as a pi-term $!x(y).P$. Thus, the correct pi-term corresponding to our former join-term should really be:

$$(\nu x)(!(x(y).P \mid x(z).Q) \mid \overline{x}a)$$

Another important remark is that in the join-calculus every channel has to be defined. There are no pending channels without a corresponding fixed set of guards in reception. The syntax forces each channel to be statically defined. This makes

the syntax heavier, but it simplifies the behavior of expressions. For instance, in the pi-calculus, it is possible to identify two receptors, as in

$$(\nu z)z(y).(x(u).P \mid y(v).Q) \mid \overline{z}x$$

In the join calculus, two different receptors remain different, since it is impossible to redefine an input guard. This allows a simpler compilation.

Finally, since the join-calculus is asynchronous and since we have no single-shot channels, it is necessary to have a new operation to provide synchronization. This is done through join-patterns which must receive *several* messages to trigger a new computation. This is reminiscent idea of multi-functions in [4]. For instance, in

$$\texttt{def}\,(x\langle y\rangle \mid t\langle u\rangle \triangleright P) \wedge (x\langle z\rangle \mid t\langle v\rangle \triangleright Q)\ \texttt{in}\ x\langle a\rangle \mid t\langle c\rangle \mid x\langle b\rangle$$

one needs to have two messages on x and t to pass one of the two guards. Therefore the execution of P and Q are mutually exclusive.

We now proceed with a precise definition of the join-calculus.

2 The join-calculus

Assume we have an infinite set of channel names $\mathcal{N} = \{x, y, z, u, v, \kappa \ldots\}$. Let us write $\tilde{x}$, $\tilde{y}$, $\tilde{z}$ ... for tuples of names. Then the syntax of the join-calculus is defined as follows. There are three kinds of expressions: processes, definitions and join-patterns. The syntax of processes is:

$P, Q ::==$		processes
	$x\langle\tilde{v}\rangle$	emission of $\tilde{v}$ on x
|	$\texttt{def}\ D\ \texttt{in}\ P$	definition of D in P
|	$P \mid Q$	parallel composition
|	$\mathbf{0}$	empty process

Interprocess communication is achieved through definitions:

$D, E ::==$		definitions
	$J \triangleright P$	elementary clause
|	$D \wedge E$	simultaneous definitions
|	$\mathbf{T}$	empty definition

An elementary clause in a definition consists of an input guard J and a guarded process P. The guard is a join pattern:

$J, J' ::==$		join-patterns
	$x\langle\tilde{v}\rangle$	reception of $\tilde{v}$ on x
|	$J \mid J'$	a composed join-pattern

An elementary join-pattern is the reception of a tuple of names on a newly defined channel. A composed pattern is a guard imposing the presence of several tuples of messages on different channels (which are not forced to be all distinct — In the

implementation of the join-calculus, there is a linearity condition on these names, but we do not need it here). But each elementary pattern is linear and the received names in its definitions are all different.

Scopes are defined by expressions of the form $\texttt{def}\, D\, \texttt{in}\, P$ which consist in a (possibly recursive) definition of channel names and a body P in which they are used. The set of defined channels in D is written $dv(D)$. Any join-pattern J also binds a set $rv(J)$ of received values. It is finally possible to define the set $fv(P)$ of free variables in the process P:

$$
\begin{array}{lll@{\qquad}lll}
rv(x\langle\tilde{v}\rangle) & = \{u \in \tilde{v}\} & & dv(x\langle\tilde{v}\rangle) & = \{x\} \\
rv(J|J') & = rv(J) \uplus rv(J') & & dv(J|J') & = dv(J) \cup dv(J') \\
\\
fv(J \triangleright P) & = dv(J) \cup (fv(P) - rv(J)) & & dv(J \triangleright P) & = dv(J) \\
fv(D \wedge D') & = fv(D) \cup fv(D') & & dv(D \wedge D') & = dv(D) \cup dv(D') \\
fv(\mathbf{T}) & = \emptyset & & dv(\mathbf{T}) & = \emptyset \\
\\
fv(x\langle v\rangle) & = \{x\} \cup \{u \in \tilde{v}\} & & & \\
fv(\texttt{def}\, D\, \texttt{in}\, P) & = (fv(P) \cup fv(D)) - dv(D) & & & \\
fv(P|P') & = fv(P) \cup fv(P') & & & \\
fv(\mathbf{0}) & = \emptyset & & &
\end{array}
$$

The operational semantics of the join-calculus is given in a CHAM style [5], with multisets $\mathcal{D}$ of definitions and $\mathcal{P}$ of processes. A configuration $\mathcal{D} \vdash \mathcal{P}$ represents a multiset $\mathcal{P}$ of processes obeying a multiset $\mathcal{D}$ of definitions. A structural equivalence $\rightleftharpoons$ describes equivalent configurations. A single reduction rule gives the derivation $\longrightarrow$ between configurations. By using the chemical semantics terminology, a configuration is a solution, processes are molecules, definitions are reactions, and $\rightleftharpoons$ is the heating/cooling reversible rule. The following six rules are enough to fully define the semantics.

$$
\begin{array}{lrcl}
\text{(str-join)} & \mathcal{D} \vdash \mathcal{P},\, P|Q & \rightleftharpoons & \mathcal{D} \vdash \mathcal{P}, P, Q \\
\text{(str-null)} & \mathcal{D} \vdash \mathcal{P}, \mathbf{0} & \rightleftharpoons & \mathcal{D} \vdash \mathcal{P} \\
\text{(str-and)} & \mathcal{D},\, D \wedge E \vdash \mathcal{P} & \rightleftharpoons & \mathcal{D}, D, E \vdash \mathcal{P} \\
\text{(str-nodef)} & \mathcal{D}, \mathbf{T} \vdash \mathcal{P} & \rightleftharpoons & \mathcal{D} \vdash \mathcal{P} \\
\text{(str-def)} & \mathcal{D} \vdash \mathcal{P},\, \texttt{def}\, D\, \texttt{in}\, P & \rightleftharpoons & \mathcal{D}, D\sigma_{\text{dv}} \vdash \mathcal{P}, P\sigma_{\text{dv}} \\
\\
\text{(red)} & \mathcal{D},\, J \triangleright P \vdash \mathcal{P},\, J\sigma_{\text{rv}} & \longrightarrow & \mathcal{D},\, J \triangleright P \vdash \mathcal{P},\, P\sigma_{\text{rv}}
\end{array}
$$

where σ_{dv} instantiates the variables $dv(D)$ to distinct and fresh new names in rule (str-def), and σ_{rv} is a plain substitution of the received variables $rv(J)$ in rule (red). By fresh variables in (str-def), we mean that the range of σ_{dv} does not intersect $fv(\mathcal{P}) \cup fv(\mathcal{D}) \cup fv(\texttt{def}\, D\, \texttt{in}\, P)$. In rule (red), in order to be able to match a pattern, we use the similarity of the syntax for join-patterns and for parallel composition of processes. Now, the traditional presentation of the chemical semantics of the join-calculus eliminates the unchanged context in each rule of the former presentation. Thus we get the following more compact version of the operational semantics:

$$
\begin{array}{lrcl}
\text{(str-join)} & \vdash P|Q & \rightleftharpoons & \vdash P, Q \\
\text{(str-null)} & \vdash \mathbf{0} & \rightleftharpoons & \vdash \\
\text{(str-and)} & D \wedge E \vdash & \rightleftharpoons & D, E \vdash \\
\text{(str-nodef)} & \mathbf{T} \vdash & \rightleftharpoons & \vdash \\
\text{(str-def)} & \vdash \texttt{def}\ D\ \texttt{in}\ P & \rightleftharpoons & D\sigma_{\mathrm{dv}} \vdash P\sigma_{\mathrm{dv}} \\
 & & & \\
\text{(red)} & J \triangleright P \vdash J\sigma_{\mathrm{rv}} & \longrightarrow & J \triangleright P \vdash P\sigma_{\mathrm{rv}}
\end{array}
$$

An alternative solution is to present the operational semantics in the universe of terms. The rules are not far from the chemical system, but are more in the style of the operational semantics of many process calculi. Firstly, there is a set of monoidal laws for the two parallel operators $|$ and $\wedge$, which correspond to the built-in cooling and heating rules (str-and), (str-join), (str-null) and (str-nodef) of the chemical presentation. Secondly, there are several scope extrusion rules:

$$
\begin{array}{rcl}
P \mid Q & \equiv & Q \mid P \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) \\
P \mid \mathbf{0} & \equiv & P \\
D \wedge D' & \equiv & D' \wedge D \\
(D \wedge D') \wedge D'' & \equiv & D \wedge (D' \wedge D'') \\
D \wedge \mathbf{T} & \equiv & D \\
 & & \\
P \mid \texttt{def}\ D\ \texttt{in}\ Q & \equiv & \texttt{def}\ D\ \texttt{in}\ P \mid Q \\
\texttt{def}\ D\ \texttt{in}\ \texttt{def}\ D'\ \texttt{in}\ P & \equiv & \texttt{def}\ D \wedge D'\ \texttt{in}\ P \\
\texttt{def}\ \mathbf{T}\ \texttt{in}\ P & \equiv & P
\end{array}
$$

In the extrusion rules above, we need as for the pi-calculus several side conditions. In the first rule commuting `def` and $|$, the extruded `def` should not bind free variables of P. This means $fv(P) \cap dv(D) = \emptyset$. In the second rule merging two `def`, one needs to avoid clashes between the bound variables of D and D', so we should satisfy $dv(D') \cap (fv(D) \cup dv(D)) = \emptyset$. Now the structural equivalence holds when two terms are alpha convertible (in the sense of the lambda calculus). Moreover, it is a congruence on terms as follows (Note that it is not necessary to define it in join-patterns):

$$
\begin{array}{rcl}
P =_\alpha Q & \Longrightarrow & P \equiv Q \\
P \equiv Q & \Longrightarrow & P \mid R \equiv Q \mid R \\
P \equiv Q & \Longrightarrow & J \triangleright P \equiv J \triangleright Q \\
D \equiv D', P \equiv Q & \Longrightarrow & \texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D'\ \texttt{in}\ Q
\end{array}
$$

As the structural rules allow to push upwards every `def` in any term, one is now able to define the reduction, up to the structural equivalence, and this reduction can be shown to be equivalent to the reduction defined by the chemical machine.

$$
\begin{array}{rcl}
\texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ J\sigma \mid Q & \rightarrow & \texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ P\sigma \mid Q \\
P \equiv R \rightarrow S \equiv Q & \Longrightarrow & P \rightarrow Q
\end{array}
$$

Proposition 1. $P \to Q$ *iff* $\emptyset \vdash \{P\}$ $(\rightleftharpoons^* \longrightarrow \rightleftharpoons^*)$ $\emptyset \vdash \{Q'\}$ *and* $Q' \equiv Q$

Since guarded terms are kept unchanged in the chemical machine, it is necessary to add the structural equivalence on the right hand side, but this not essential. The two reductions are basically the same. Remember that we dropped the structural equivalence in join-patterns, which could have been kept in order to give a more intuitive meaning to the | operator.

An (output) barb is any free variable on which a process can immediately emit. The weak barbed bisimulation $\dot{\approx}$ and the $\approx$ observational congruence are defined in the standard way. See also [16] for may and must testing semantics in the join-calculus.

Definition 2. $P \Downarrow_x$ iff $P \to^* P' = \texttt{def}\ D\ \texttt{in}\ x\langle\tilde{v}\rangle \mid Q$ and $x \notin dv(D)$

Definition 3. $\dot{\approx}$ is the largest relation such that $P \dot{\approx} Q$ implies

1. $P \Downarrow_x$ implies $Q \Downarrow_x$
2. $\forall P'\ P \to P'\ \ \exists Q'\ Q \to^* Q'$ and $P' \dot{\approx} Q'$
3. $\forall Q'\ Q \to Q'\ \ \exists P'\ P \to^* P'$ and $P' \dot{\approx} Q'$

Definition 4. The observational congruence $\approx$ is the largest congruence contained in the weak barbed bisimulation, i.e such that

1. $P \approx Q$ implies $P \dot{\approx} Q$
2. $P \approx Q$ implies $R \mid P \approx R \mid Q$
3. $P \approx Q$ implies $\texttt{def}\ D\ \texttt{in}\ P \approx \texttt{def}\ D\ \texttt{in}\ Q$

3 Abbreviations and elementary encodings

As for the pi-calculus, one can consider a monadic version of the join-calculus to get a more elementary language. In this version, each channel may only transport single names, and all definitions contains a simple binary join-pattern. The monadic calculus is inductively defined as follows

$$P, Q ::= x\langle u\rangle \ \mid\ P|Q \ \mid\ \texttt{def}\ x\langle u\rangle|y\langle v\rangle \triangleright P\ \texttt{in}\ Q$$

Proposition 5. *There is an encoding for every P,Q of the join-calculus into a process* $[\![P]\!]_0$ *of the monadic calculus such that:*

$$P \approx Q \quad \textit{iff} \quad [\![P]\!]_0 \approx [\![Q]\!]_0$$

One can find the details of this encoding in [11]. In that paper, there is also another interesting property for expressing the relative power of the join-calculus and of the asynchronous pi-calculus [7, 2] whose definition is:

$$P, Q ::= \overline{x}\langle u\rangle \ \mid\ P|Q \ \mid\ x(u).P \ \mid\ !x(u).P \ \mid\ (\nu x)P$$

From the monadic join-calculus to the pi-calculus, the translation $[\![\ ]\!]_j$ is rather easy since it is the following simple rewriting

$$
\begin{aligned}
[\![x\langle v\rangle]\!]_j &= \overline{x}v \\
[\![P \mid Q]\!]_j &= [\![P]\!]_j \mid [\![Q]\!]_j \\
[\![\mathtt{def}\ x\langle u\rangle \mid y\langle v\rangle \triangleright P\ \mathtt{in}\ Q]\!]_j &= (\nu x)(\nu y)(!x(u).y(v).[\![P]\!]_j \mid [\![Q]\!]_j)
\end{aligned}
$$

In the reverse direction, the translation makes explicit the asymmetry between reception and emission in the join-calculus. A channel x is represented by two channels, the first one x_i for input and the second x_o for output. With join-patterns, it is possible to connect them as below.

$$
\begin{aligned}
[\![\overline{x}v]\!]_\pi &= x_o[\![v_o, v_i]\!]_\pi \\
[\![P \mid Q]\!]_\pi &= [\![P]\!]_\pi \mid [\![Q]\!]_\pi \\
[\![x(v).P]\!]_\pi &= \mathtt{def}\ \kappa\langle v_o, v_i\rangle \triangleright [\![P]\!]_\pi\ \mathtt{in}\ x_i\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &= \mathtt{def}\ \kappa\langle v_o, v_i\rangle \triangleright x_i\langle\kappa\rangle \mid [\![P]\!]_\pi\ \mathtt{in}\ x_i\langle\kappa\rangle \\
[\![(\nu x)P]\!]_\pi &= \mathtt{def}\ x_o\langle v_o, v_i\rangle \mid x_i\langle\kappa\rangle \triangleright \kappa\langle v_o, v_i\rangle\ \mathtt{in}\ [\![P]\!]_\pi
\end{aligned}
$$

In fact both encodings get more intricate in [11] , since it is possible to observe guards in the translated terms. It is necessary to add a firewall context in order to get a fully abstract translation.

To avoid an excessive use of channels corresponding to programming in a continuation passing style, we follow the approach of Pict [23, 24], and consider an important set of abbreviations to allow results of expressions to be returned on channels. We add a new set of synchronous channels, written in roman style such as x, which should not be confused with the names as x for asynchronous channels. Each synchronous name has a unique κ_x continuation channel attached to it. The new extended join-language is now:

P, Q ::==		processes
	$x\langle\tilde{v}\rangle$	asynchronous emission of $\tilde{v}$ on x
	$\mathtt{def}\ D\ \mathtt{in}\ P$	definition of D in P
	$P \mid Q$	parallel composition
	$\mathbf{0}$	empty process
	$\mathrm{x}(\tilde{V}); P$	sequential composition
	$\mathtt{let}\ \tilde{u} = \tilde{V}\ \mathtt{in}\ P$	named values
	$\mathtt{reply}\ \tilde{V}\ \mathtt{to}\ \mathrm{x}$	implicit continuation
D, E ::==		definitions
	$J \triangleright P$	P guarded by J
	$D \wedge E$	simultaneous definitions
	$\mathbf{T}$	empty definition
J, J' ::==		join-patterns
	$x\langle\tilde{v}\rangle$	asynchronous reception of $\tilde{v}$ on x
	$J \mid J'$	conjunction of guards J and J'
	$\mathrm{x}(\tilde{v})$	synchronous reception of $\tilde{v}$ on x
V ::==		values
	x	value name
	$\mathrm{x}(\tilde{V})$	synchronous call

where the new terms are defined as abbreviations in the following way:

$$\begin{aligned} \mathrm{x}(\tilde{v}) &= x\langle \tilde{v}, \kappa_x \rangle && \text{(in join-patterns)} \\ \texttt{reply}\ \tilde{V}\ \texttt{to}\ \mathrm{x} &= \kappa_x\langle \tilde{V} \rangle && \text{(in processes)} \end{aligned}$$

$$\begin{aligned} x\langle \tilde{V} \rangle &= \texttt{let}\ \tilde{v} = \tilde{V}\ \texttt{in}\ x\langle \tilde{v} \rangle \\ \texttt{let}\ \tilde{u} = \tilde{V}\ \texttt{in}\ P &= \texttt{let}\ u_1 = V_1\ \texttt{in}\ \texttt{let}\ u_2 = \cdots\ \texttt{in}\ P \\ \texttt{let}\ \tilde{u} = \mathrm{x}(\tilde{V})\ \texttt{in}\ P &= \texttt{def}\ \kappa\langle \tilde{u} \rangle \triangleright P\ \texttt{in}\ x\langle \tilde{V}, \kappa \rangle \\ \texttt{let}\ u = v\ \texttt{in}\ P &= P\{v/u\} \\ \mathrm{x}(\tilde{V}); P &= \texttt{def}\ \kappa\langle\rangle \triangleright P\ \texttt{in}\ x\langle \tilde{V}, \kappa \rangle \end{aligned}$$

For instance, the previous encoding of the pi-calculus into the join-calculus is written below. If we expand the definition of the synchronous name new_pi_channel, we exactly get $[\![(\nu x)P]\!]_\pi$. The two returned channels *send* and receive correspond to the x_o and x_i. Please remark the elegance of the synchronous notation, when we use channels receive, g_1 and print.

```
def new_pi_channel() ▷
    def send⟨x⟩ | receive() ▷ reply x to receive
    in  reply send, receive to new_pi_channel
in
let s1, r1 = new_pi_channel() in
in  s1⟨1⟩ | s1⟨2⟩ | print(r1()) ; 0
```

It is similarly possible to write examples with traditional imperative and applicative features, such as a counter with a state that is is represented as an asynchronous message *count* giving its current value. A new counter returns two methods for getting or incrementing the value. It also gives the initial value x to the counter. When one increments the value, one also has to send a null acknowledge to the synchronous get method.

```
def new_counter(x) ▷
    def count⟨n⟩ | inc() ▷ count⟨n+1⟩ | reply to inc
    ∧   count⟨n⟩ | get() ▷ count⟨n⟩ | reply n to get
    in  count⟨x⟩ | reply get, inc to new_counter
in ...
```

Now, we define a locking system, again with the same programming paradigm. A new lock consists of two synchronous methods lock and unlock, which are used in a standard imperative way, to disallow the mixture of - and + on the output.

```
def new_lock() ▷
    def lock() | free⟨⟩ ▷ reply to lock
    ∧   unlock() ▷ free⟨⟩ | reply to unlock
    in  free⟨⟩ | reply lock, unlock to new_lock in
let lock, unlock = new_lock()
in  (lock(); print("----"); unlock(); 0)
|   (lock(); print("++++"); unlock(); 0)
```

Note that we could have just written in a less traditional way by use of the join-patterns, since the *free* resource can be directly used in the definition of functions.

$$\begin{array}{ll}\texttt{def} & \mathit{do\_print}\langle\rangle \mid \mathit{free}\langle\rangle \triangleright \mathrm{print}(\texttt{"----"})\ ;\ \mathit{free}\langle\rangle \\ \wedge & \mathit{do\_print}\langle\rangle \mid \mathit{free}\langle\rangle \triangleright \mathrm{print}(\texttt{"++++"})\ ;\ \mathit{free}\langle\rangle \\ \texttt{in} & \mathit{free}\langle\rangle \mid \mathit{do\_print}\langle\rangle \mid \mathit{do\_print}\langle\rangle \end{array}$$

Finally, we consider the example of a register cell, with two synchronous methods get and set. The state is again stored in an asynchronous message. Remark that this example is well-typed (see the discussion in section 6).

$$\begin{array}{l}\texttt{def}\ \mathrm{ref}(v) \triangleright \\ \quad \texttt{def}\ \mathrm{get}() \mid \mathit{state}\langle x\rangle \triangleright \mathit{state}\langle x\rangle \mid \texttt{reply}\ x\ \texttt{to}\ \mathrm{get} \\ \quad\ \wedge\ \mathrm{set}(y) \mid \mathit{state}\langle x\rangle \triangleright \mathit{state}\langle y\rangle \mid \texttt{reply to}\ \mathrm{set} \\ \quad \texttt{in}\ \ \mathit{state}\langle v\rangle \mid \texttt{reply}\ \mathrm{get}, \mathrm{set}\ \texttt{to}\ \mathrm{ref} \\ \texttt{in}\ \texttt{let}\ \mathrm{g}_1, \mathrm{s}_1 = \mathrm{ref}(7) \\ \texttt{in}\ \texttt{let}\ \mathrm{g}_2, \mathrm{s}_2 = \mathrm{ref}(\texttt{"seven"}) \\ \texttt{in}\ \ldots \end{array}$$

4 The join-calculus with locations

The join-calculus has been introduced to impose locality on definitions of channels inside the syntax of terms. There are other solutions proposed by [3, 1, 21, 25] where this is achieved through (linear) typing, since as exposed in the introduction it is preferable to have the uniqueness of the location which defines a channel. Our caveat is that the syntax is enough, and that we can speak of locality in an untyped world. It is possible to add types to the join-calculus à la ML [13], but we do not want to have a logical system for handling locations. The claim of the distributed join-calculus [12] is that it is possible to speak of locations in a rather simple and concise way.

Firstly, we need to have an explicit notion of location. As we believe in static scopes and first-class objects, a location is taken in an infinite set of names $\mathcal{L} = \{a, b, c, \ldots\}$. Intuitively, a location resides on a physical site, but we want to stay at a process calculus level, and we will not specify sites. Intuitively, some of the names represent fixed addresses on a network. But locations are more logical spaces, such as Unix process spaces or smaller ones. Locations are units of migration. In a definition of a port, we want to be able to specify where this port is located. Locations are introduced in the syntax by adding a new kind of definition:

$$D, E ::== \ \ldots \mid a[D : P]$$

meaning that the set $dv(D)$ of defined channels of D is located at location a.

Secondly, locations obey a static scope discipline. In $a[D : P]$, the set of defined variables now also contains a. The scope of a is delimited by the `def` statement which contains it. In the statement $\texttt{def}\ E \wedge a[D : P]\ \texttt{in}\ Q$, the scope of a is

formed of the whole expression. More precisely, we extend the sets $dv(D)$ of defined variables and $fv(D)$ of free variables in D as follows:

$$\begin{aligned} fv(a[D:P]) &= \{a\} \cup fv(D) \cup fv(P) \\ dv(a[D:P]) &= \{a\} \uplus dv(D) \end{aligned}$$

Thirdly, as for channel names, a location and the code it contains are defined at the same time. We will see later what is the meaning of P in $a[D:P]$. Intuitively, it is the initialization of location a when it becomes active.

Fourthly, locations are first-class objects. They can be passed as value to any channel, which means that we have also to extend the syntax of $x\langle\tilde{v}\rangle$ in processes to allow $v \in \mathcal{L}$. About static scoping, we may remark that this is quite different from the approach of ambients of Cardelli and Gordon [10], where the names of locations is purely dynamic. This latter calculus is more oriented towards protection, and it may seem important to mark all the routing information giving the access to a location. Here we believe in abstraction and prefer to stay just with names that can be abstracted.

Finally, we have two important remarks on the uniqueness of locations and receptors, which add two syntactic side-conditions on the well-formness of definitions with locations. The first condition states that a location a can only be defined once in any definition D. The second necessary condition is that a defined channel x may only appear in the join-patterns of one location. For instance, the following definitions are forbidden:

$$\begin{array}{l} \mathtt{def}\ a[D:P] \wedge a[E:Q] \triangleright R\ \mathtt{in}\ S \\ \mathtt{def}\ a[x\langle u\rangle \triangleright P:Q] \wedge b[x\langle v\rangle \triangleright R:S]\ \mathtt{in}\ T \end{array}$$

The recursive syntax imposes a hierarchy on locations, since we may have locations inside locations. In fact, hierarchical locations are a convenient way of programming mobile agents. At any given time, there will be a tree of locations. Each location corresponds exactly to a single node, and it will be impossible to give the same location to two different nodes. If a location a appears at a subtree of location b, it means that a resides on the same site as b. At the top of this tree, we may suppose a non located root whose principal sons are some physical addresses on a network. This tree evolves any time a process wants to migrate. A process may move to the son of a given location by the new process expression:

$$P, Q ::== \ \ldots \mid go\langle a, \kappa\rangle$$

meaning that the implicit current location of the process $go\langle a, \kappa\rangle$ will become a direct son of location a, and when this operation is terminated, there will a null message sent on channel κ. This term has no binding effect. Therefore

$$fv(go\langle a, \kappa\rangle) = \{a, \kappa\}$$

As for channel names, a process can migrate only to a location whose name is statically known to it. This may be infeasible at the initialization of the system, but as usual we may use built-in name servers. Once a location is passed to a process, it has always the same meaning.

In the join-calculus without locations, we have defined in several ways the operational semantics. We shall try to follow the same method now. It is possible to have the chemical description. We still have the five previous structural rules (str-join), (str-and), (str-def), (str-null) and (str-nodef) and the (red) reduction rule. Notice that (str-def) applies also to locations which are in the defined names of the distinguished definition and thus also produces new fresh location names. Now they are extra rules due to localities. The reflexive chemical machine is augmented with locations. In fact, we will have a chemical machine at each node of the location tree. A machine $\mathcal{D} \vdash_a \mathcal{P}$ is therefore indexed by the name of its location. For technical reasons, we keep as index the access path ϕ of location a from the root (a included). In the following, configurations of the distributed reflexive chemical machine will be a set of expressions $\mathcal{D} \vdash_\phi \mathcal{P}$ where $\phi, \psi \in \mathcal{L}^*$. We denote such configurations using the $||$ operator to separate machines. Now we add the following three rules in the concise form.

$$
\begin{array}{lrcl}
\text{(str-loc)} & a[D : P] \vdash_\phi & \rightleftharpoons & \vdash_\phi \;||\; \{D\} \vdash_{\phi a} \{P\} \qquad (a \text{ frozen}) \\
\text{(comm)} & \vdash_\phi x\langle \tilde{v} \rangle \;||\; J \triangleright P \vdash & \longrightarrow & \vdash_\phi \;||\; J \triangleright P \vdash x\langle \tilde{v} \rangle \qquad (x \in dv(J)) \\
\text{(move)} & a[D : P \mid go\langle b, \kappa \rangle] \vdash_\phi \;||\; \vdash_{\psi b} & \longrightarrow & \vdash_\phi \;||\; a[D : P \mid \kappa\langle\rangle] \vdash_{\psi b}
\end{array}
$$

The rule (str-loc) introduces a new machine. On the left hand side, we have a definition located at a sublocation a of ϕ and we create a new machine for it, with just D as initial definition and P as initial process. The machine at ϕ continues but discharged of the sublocation a, although it is still present in the location tree. This rule shows that the access path is written from left to right. But again, with our condition on the syntax of terms, the name a is enough to distinguish where it is in the location tree. The side condition "a frozen" means that there is no other chemical machine in this configuration of the form $\vdash_{\phi a \psi}$. It forces the full subtree of a to be frozen on the left hand side. By the notation $\{D\}$ and $\{P\}$, we say that there are no extra definitions or processes on the right hand side for machine $\vdash_{\phi a}$.

The second rule (comm) is reminiscent of distributed systems, where routing is a different step from actual computation. It means that in order to emit on a given remote channel, it is first necessary to ship the message to the remote machine. More precisely, on the left hand side, on machine $\vdash_\phi$, one wants to send a message on x, but x is in the defined set $dv(J)$ of a join pattern located on another machine. Notice that as the location containing the definition of x is unique, we can deterministically find the machine receiving the messages for x (it suffices to propagate enough routing information with x). On the right hand side, the first machine has no longer the message which has been delivered to the second machine. This rule has to be combined with (red) in order to perform actual remote communication.

The third rule (move) gives the semantics of the migration. A sublocation ϕa of ϕ wants to move to a sublocation ψb of ψ. On the right hand side, the machine $\vdash_\phi$ is fully discharged of the a location, but location a now on $\vdash_{\psi b}$ is notified with a continuation message, when the migration is finished. (We will use the same convention as for synchronous channel names and do not hesitate to write go(b) for $go\langle a, \kappa \rangle$ in a synchronous way).

In the expanded chemical style, which keeps all the lengthy contexts, and may thus avoid any confusion, the three new rules are written as follows:

$$\begin{array}{lll}
\text{(str-loc)} & \mathcal{M} \;||\; \mathcal{D}, a[D:P] \vdash_\phi \mathcal{P} \rightleftharpoons \mathcal{M} \;||\; \mathcal{D} \vdash_\phi \mathcal{P} \;||\; \{D\} \vdash_{\phi a} \{P\} & (a \text{ frozen}) \\
\text{(comm)} & \mathcal{M} \;||\; \mathcal{D} \vdash_\phi x\langle \tilde{v}\rangle, \mathcal{P} \;||\; \mathcal{D}', J \triangleright P \vdash_\psi \mathcal{P}' \longrightarrow & \\
& \qquad \mathcal{M} \;||\; \mathcal{D} \vdash_\phi \mathcal{P} \;||\; \mathcal{D}', J \triangleright P \vdash_\psi x\langle \tilde{v}\rangle, \mathcal{P}' & (x \in dv(J)) \\
\text{(move)} & \mathcal{M} \;||\; \mathcal{D}, a[D:P \mid go\langle b, \kappa\rangle] \vdash_\phi \mathcal{P} \;||\; \mathcal{D}' \vdash_{\psi b} \mathcal{P}' \longrightarrow & \\
& \qquad \mathcal{M} \;||\; \mathcal{D} \vdash_\phi \mathcal{P} \;||\; \mathcal{D}', a[D:P \mid \kappa\langle\rangle] \vdash_{\psi b} \mathcal{P}' &
\end{array}$$

With some care, we may define structural equivalence, syntactic reduction, barbs and observational congruence as in section 2 and definitions 2, 3 and 4.

To any join-term of the distributed calculus, we can associate a term of the plain calculus, by erasing the location information with the following rewrite rules:

$$\begin{array}{rcl}
\texttt{def}\ a[D:P] \wedge E\ \texttt{in}\ Q & \Longrightarrow & \texttt{def}\ a\langle\rangle \triangleright 0 \wedge D \wedge E\ \texttt{in}\ P \mid Q \\
go\langle a, \kappa\rangle & \Longrightarrow & \kappa\langle\rangle
\end{array}$$

Let $unloc(P)$ be the normal form that may be shown unique up to alpha conversion of this transformation. We have the following network transparency proposition.

Proposition 6. *Let P be move-lock free. Then $P \approx Q$ iff $unloc(P) \approx unloc(Q)$*

The proof of this statement needs some subtleties. Firstly, we kept addresses as inert receptors in the $unloc(P)$ translation, since this preserves all bindings. Otherwise removing location names may unveil new barbs. An alternative solution would have been to suppress the $a\langle\rangle \triangleright 0$ definition and to replace every instance of a in processes by a fixed constant, meaning that all locations coincide. Secondly, the move-lock free condition means that there is no blocking happening in the (move) rule. It may only be the case when we have a term with a circular migration such as $a[D : P \mid go\langle a, \kappa\rangle]$, ie the sublocation a wants to move to the son of itself. This is forbidden by the (move) rule since (str-loc) prevents a frozen location to be at same time somewhere else in the diluted solution of the chemical machines. Statically, one may prevent it by typing locations. A much simpler case would say that we have just two levels: fixed sites (principal sons of the root in the locations tree) and standard locations, and moves to sites are only allowed. But finer solutions could be better.

The previous proposition is the central property in the design of the distributed join-calculus. Rephrasing Cardelli's terminology [9], our calculus has the two properties of network awareness, since we manipulate explicitly the locations, and of network transparency, which states that no modification of the meaning of a term will arrive by giving explicit locations to terms of the join-calculus. Unfortunately, but as expected, this property is no longer true in the presence of location failures, which we will consider later.

We now use the extended notation with synchronous names, defined in section 3. The two following join-terms are supposed to be on two different locations. On a

server side, we have an applet which is a write-once cell with two methods get and put. (Our implementation allows to register this applet on a name server). On the client side, one may call the name server to get the pointer to the applet (this is obligatory since the two toplevel terms are supposed to be in two distinct scopes). Once the name server responds, one call the cell applet with the client location as parameter. The (red) rule copies the body of cell and creates a fresh location for applet which moves, becomes a son of *user* in the location tree and initializes the state of the cell to $none\langle\rangle$. Now the fresh copy of applet is on the user side, and can be used in a standard way. We put the string `"world"` in the write-once cell, read it (and thus empty it), then put again the concatenation of the strings `"hello, "` and `"world"`, read it, and then print it. Notice all these get/put operations are local to *user*. We have downloaded the applet cell from the server to the user.

```
def cell(a) ▷
    def applet [
        get() | some⟨x⟩ ▷ none⟨⟩ | reply x to get
     ∧  put(x) | none⟨⟩ ▷ some⟨x⟩ | reply to put
     :
        go(a);  none⟨⟩
    ] in
    reply get, put to cell
in
ns.register("cell", cell)

let cell = ns.lookup("cell") in
def user[
    :
    let get, put = cell(user) in
    put("world"");
    put("hello, " ^ get());
    print(get());
]
```

This applet is very classical, although we can appreciate the conciseness of the join-calculus. Suppose now that the server wants to keep an eye on the applet by maintaining a log file to be further exploited. It means that while the applet moves to the user side, we have to keep some remote link, in order to produce the logging information. The following small variation of the applet is enough, and we may notice how the lexical scope discipline helps. On the server side the new log definition stays on the server location, while the applet location migrates with the (move) rule reacting to the go(a) term. But inside the applet-methods get and put, we now have some synchronous calls of log indicating the status of the exported cell. Notice that the calls to log are in fact remote. The rest of the term is identical.

```
def cell(a) ▷
    def log(s) ▷
        print("cell" ^ s ^ "\n"); reply to log in
    def applet[
        get() | some⟨x⟩ ▷
            log("is empty");
            none⟨⟩ | reply x to get
     ∧  put(x) | none⟨⟩ ▷
            log("contains " ^ x);
            some⟨x⟩ | reply to put
    :
        go(a); none⟨⟩
    ] in
    reply get, put to cell
in
ns.register("cell", cell)
```

5 The join-calculus with failures

We consider permanent failures for locations. Intuitively, a location fails with its sublocations in the location tree, which prevents any further reduction to take place. Reacting to failures in an asynchronous world is not easy, since it is hard to distinguish between a true failure and an alive location which is simply slow. On the other hand we may have an external information about the failure of one site, or we may want to explicitly kill a location on which an agent resides. In our model, we leave this open and abstract such failures as primitives.

In the join-calculus, we need to mark failed locations. This can be done by considering a new set of names for them. But since a location is a bound variable, it would be necessary to change all the occurrences of a location. Another solution is to give a state to a location, dead or alive, but then we will have many side-conditions in our semantics. We prefer to mark the location with a new constant $\Omega \notin \mathcal{L}$. A machine $\vdash_\phi$ is dead iff it contains an Ω, since every sublocation of a dead location is also dead, and alive otherwise. The rule (str-loc) may now introduce definitions $\Omega a[D : P]$, and we have to consider every rule of the semantics with the possibility of an Ω tag. Instead of doubling the number of rules, we designate by $\epsilon a[D : P]$ a definition the status of which is unknown and could be dead or alive. The rules are now written as follows:

$$\text{(str-loc)} \qquad \epsilon a[D : P] \vdash_\phi \;\rightleftharpoons\; \vdash_\phi \;||\; \{D\} \vdash_{\phi \epsilon a} \{P\} \qquad (a \text{ frozen})$$

$$\text{(red)} \qquad J \triangleright P \vdash J\sigma_{\text{rv}} \longrightarrow J \triangleright P \vdash P\sigma_{\text{rv}} \qquad (\phi \text{ alive})$$

$$\text{(comm)} \qquad \vdash_\phi x\langle \tilde{v} \rangle \;||\; J \triangleright P \vdash \longrightarrow \vdash_\phi \;||\; J \triangleright P \vdash x\langle \tilde{v} \rangle \qquad (x \in dv(J), \phi \text{ alive})$$

$$\text{(move)} \quad a[D : P \mid go\langle b, \kappa \rangle] \vdash_\phi \;||\; \vdash_{\psi \epsilon b} \longrightarrow \vdash_\phi \;||\; a[D : P \mid \kappa\langle\rangle] \vdash_{\psi \epsilon b} \qquad (\phi \text{ alive})$$

The remaining rules are unchanged. The liveness condition implies that we stopped any emission or migration *from* a dead site. But it is still possible to send a message or to migrate to a dead location. Remark also that the top of the location tree can never fail. For detection of the failure of a location and voluntary stops, we have

$$\begin{array}{lrcll} \text{(halt)} & a[D : P \mid \mathit{halt}\langle\rangle] \vdash_\phi & \longrightarrow & \Omega a[D : P] \vdash_\phi & (\phi \text{ alive}) \\ \text{(detect)} & \vdash_\phi \mathit{fail}\langle a, \kappa\rangle \;\; || \vdash_{\psi \epsilon a} & \longrightarrow & \vdash_\phi \kappa\langle\rangle \;\; || \vdash_{\psi \epsilon a} & (\psi \epsilon a \text{ dead}, \;\; \phi \text{ alive}) \end{array}$$

The (halt) rule simply marks the current location with Ω. The (detect) rule sends a signal on κ when the tested location failed. As usual, failures can be problematic. For instance, in an asynchronous world,, our semantics is too strong, since it assumes a global knowledge of the liveness of every locations. In rule (comm), one cannot emit a message if the location is not alive. In a real world, one cannot distinguish the reception of a message emitted just before the failure of a site and the test of a failure before the posting of this message. However this semantics can be approximated in the distributed implementation, and one may show that a trace equivalent meaning can be given to *fail*.

6 Typing the join-calculus

Our channels can be typed in the standard way. The type of polyadic channel carrying values of type $\tau_1, \tau_2, \ldots \tau_n$ is written $\langle \tau_1, \tau_2, \ldots \tau_n \rangle$. This allows to respect the types of built-in functions, but also the constraints which exist inside the join-calculus. Take the definition

$$\texttt{def}\ \ \mathit{apply}\langle \kappa, x\rangle \triangleright \kappa\langle x\rangle$$

If x has type α, the type of κ should be $\langle\alpha\rangle$. In fact, it is possible to give to *apply* the polymorphic type $\forall a.\langle\langle\alpha\rangle, \alpha\rangle$ as in ML. Consider now the definition

$$\texttt{def}\ \ \mathit{fun}\langle\kappa\rangle \mid \mathit{arg}\langle x\rangle \triangleright \kappa\langle x\rangle$$

The type of *fun* and *arg* are $\langle\langle\alpha\rangle\rangle$ and $\langle\alpha\rangle$, but not $\forall a.\langle\langle\alpha\rangle\rangle$ and $\forall a.\langle\alpha\rangle$, since the types of the co-defined names *fun* and *arg* are related. In this case, we cannot generalize the types of these two ports. This leads to the following simple typing discipline: a type variable which appears free in the type of several co-defined channels cannot be generalized, see [13] for more details. The type rules are similar to the ones of Damas and Miner. Again, the syntax of the join-calculus greatly helps since the grouping of definitions of channel names allows this check to be syntax driven. This is not the case for the pi-calculus [26].

Another interesting fact is that a synchronous name x can be given a new kind of type abbreviation $\alpha \to \beta$, meaning it takes α-typed values and return β-typed ones, in a functional style. This means that when the abbreviation is expanded the corresponding asynchronous channel x has type $\langle\alpha, \langle\beta\rangle\rangle$. With this new typing for

synchronous channels, one may freely mix synchronous and asynchronous names without any ambiguity. There is no need for any typographic convention in order to distinguish these names. Moreover one may merge the syntax of the `let` and `def` statements.

7 Conclusion

We have shown several facets of the join-calculus: the core language, the distributed version, the failure model and a typing discipline.

We did not speak much of our distributed implementation, which is the work of Fournet and Maranget [14]. It demonstrates that the join-calculus can be used as an actual basis for distributed programming, and allows us to experiment with different style of programming with mobility. It also raises some new problems on compilation and optimization of such programs.

There is still some work to have an extensive theory of its semantics. The join-calculus is interesting by itself since it has the same expressiveness as the pi-calculus, but is quite different on several points. The distributed version may still look quite complex. Maybe this is intrinsic to the semantics of a language with mobility. In terms of clarity, our calculus with locations is very competitive with other proposals. For instance, it gives a strong basis to discuss security issues in distributed protocols.

Acknowledgements

I have benefited from discussions on the join-calculus with C. Fournet, G. Gonthier, L. Maranget and D. Rémy. A special thanks to C. Fournet, G. Gonthier and U. Nestmann for their help in the writing of this paper.

References

1. R. Amadio. An asynchronous model of locality, failure, and process mobility. To appear in COORDINATION 97, Berlin, 1997. Rapport Interne LIM February 1997, and INRIA Research Report 3109., 1997.
2. R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. In Montanari and Sassone [20]. LNCS 1119.
3. R. M. Amadio and M. Dam. Toward a modal theory of types for the π-calculus. In *Proc. Formal Techniques in Real Time and Fault Tolerant Systems 96, Uppsala*, 1996. LNCS 1135.
4. J.-P. Banâtre, M. Banâtre, and F. Ployette. Distributed system structuring using multi-functions. Rapport de Recherche 694, INRIA Rennes, June 1987.
5. G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings POPL '90*, pages 81–94, San Francisco, Jan. 17-19 1990.
6. A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the emerald system. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, pages 78–86, 1986.

7. G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, 1992.
8. L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, Jan. 1995.
9. L. Cardelli. Global computation. *ACM Sigplan Notices*, 32:1:66–68, 1997.
10. L. Cardelli and A. Gordon. A calculus of mobile ambients. Private communication, 1997.
11. C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, Jan. 21-24 1996. ACM.
12. C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In Montanari and Sassone [20]. LNCS 1119.
13. C. Fournet, C. Laneve, L. Maranget, and D. Rmy. Implicit typing la ML for the join-calculus. In *Proceedings of the 8th International Conference on Concurrency Theory*, Warsaw, Poland, 1–4July 1997. Springer. to appear in LNCS.
14. C. Fournet and L. Maranget. The join-calculus language. Available electronically (`http://pauillac.inria.fr/join`), 1997.
15. A. Giacalone, P. Mishra, and S. Prasad. FACILE: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*, 18(2):121–160, 1989.
16. C. Laneve. May and must testing in the join-calculus. Technical Report UBLCS 96-04, University of Bologna, March 1996.
17. L. Leth and B. Thomsen. Some facile chemistry. Technical Report ECRC-92-14, European Computer-Industry Research Centre, Munich, May 1992.
18. R. Milner. The polyadic π-calculus: a tutorial. In F. L. Bauer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*. Springer Verlag, 1993.
19. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, pages 1–40 & 41–77, Sept. 1992.
20. U. Montanari and V. Sassone, editors. *Proceedings of the 7th International Conference on Concurrency Theory*, Pisa, Italy, 26–29Aug. 1996. Springer. LNCS 1119.
21. R. D. Nicola, G. Ferrari, and R. Pugliese. Locality based linda: programming with explicit localities. To appear in LNCS FASE-TAPSOFT'97, 1997.
22. B. C. Pierce, D. Rémy, and D. N. Turner. A typed higher-order programming language based on the pi-calculus. In *Workshop on Type Theory and its Application to Computer Systems, Kyoto University*, July 1993.
23. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in Milner *Festschrift*, MIT Press, 1997.
24. B. C. Pierce and D. N. Turner. PICT user manual. Available electronically, 1997.
25. J. Riely and M. Hennessy. Distributed processes and location failures. Report 2/97, University of Sussex, Brighton, April 1997.
26. D. N. Turner. *The π-calculus: Types, polymorphism and implementation*. PhD thesis, LFCS, University of Edinburgh, 1995.
27. J. White. Telescript technology: the foundation for the electronic marketplace. Technical report, General Magic, 1994.

This article was processed using the LaTeX macro package with LLNCS style