Foundational Calculi for Network Aware Programming
— An Assessment from a Programming Perspective —

GIANLUIGI FERRARI
Dipartimento di Informatica, Università di Pisa
ROSARIO PUGLIESE
Dipartimento di Sistemi e Informatica, Università di Firenze
and
EMILIO TUOSTO
Dipartimento di Informatica, Università di Pisa

We classify and evaluate a number of foundational calculi for network-aware programming. The benefits and drawbacks of each calculus and its appropriateness to express metaphors for network-aware programming are evaluated along three different guidelines: the programming abstractions the calculus suggests; the underlying programming model; the security mechanisms provided. This evaluation will help in understanding the potentials and the advantages of using foundational calculi in the design of new programming languages for network-aware programming.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*

---

1. INTRODUCTION

Highly distributed systems and networks have now become a common platform for wide-area network (WAN) applications which use network facilities to access remote recourses and services. WAN applications distinguish themselves from traditional distributed applications not only in terms of *scalability* (huge number of users and nodes), *connectivity* (both availability and bandwidth), *heterogeneity* (operating systems and application software) and *autonomy* (of administration domains having strong control of their resources), but particularly in terms of the ability of dealing with *dynamic* and *unpredictable* changes of their network environment (e.g. availability of network connectivity, lack of resources, node failures, network reconfigurations and so on). The crucial point is that WAN applications are *network-aware*: WAN applications are aware of the sites (administrative domains) where they are currently located and of how they can cross and move to other sites.

Several researchers have proposed to use *mobility* as the basic mechanism to support network-aware programming, namely to program WAN applications. In the literature the term mobility is used to denote different mechanisms, ranging from simple mechanisms, which only provide the ability of downloading code for execution (e.g. [Arnold and Gosling 1997]), to more sophisticated ones, which support migration of entire computations (e.g. [White 1996; Acharya et al. 1997; Lange and Oshima 1998]). We can distinguish among at least five forms of mobility:

- *mobile computing* concerns computations that are executed on mobile devices
- *mobile net* concerns dynamic reconfiguration of networks;
- *mobile code* concerns migration (or downloading) and evaluation of executable code;
- *mobile process* concerns migration of both code and state;
- *mobile agents* concerns migration of code, state and authority to act on behalf of a principal on a wide-area network.

A number of useful WAN applications have been developed by experienced programmers using standard programming paradigms (e.g. Client-Server); however such applications are often a collections of *ad hoc* modules glued together with *ad hoc* mechanisms without any (semantic) foundation. Semantic foundations could play a central role to prove and to certify correctness of WAN applications. Hence, a key challenge is to identify what programming abstractions are most suitable for network-aware programming and to provide foundational and effective tools to support the development and certification (establishing and proving correctness) of WAN applications.

Foundational calculi have been used to provide formal foundations to the design of programming languages. A foundational calculus is both a kernel programming language and a computational model for describing and reasoning about the behaviour of programs. It provides a formal basis to identify and generate new programming abstractions and analytical tools. A well-known example of foundational calculus for programming languages is provided by the λ-calculus (and its enrichments).

Network-aware programming has prompted the study of the foundations of programming languages with advanced features including mechanisms for agent mobility, for managing security, and for coordinating and monitoring the use of resources. Recently, some foundational calculi have been designed that tackle most of the phenomena related to network-aware programming. Their programming models encompass abstractions to represent the execution contexts of the net where applications roam and run, and mechanisms to support the specification and the implementation of security policies.

This paper aims at analyzing a few foundational calculi for network-aware programming and at demonstrating how these calculi can be used as a basis in the design of new programming languages. We have chosen to focus on a list of representative foundational calculi that have been proposed in the literature. In particular, we evaluate and compare:

—The *Distributed π-calculus*: Dπ [Hennessy and Riely 1998; 1999; Riely and Hennessy 1999b; 1999a],

—The *Distributed* `join-calculus`: `Djoin` [Fournet and Gonthier 1996; Fournet et al. 1996; Abadi et al. 1998],

—The KLAIM *kernel calculus*: KLAIM [De Nicola et al. 1998; 1999; De Nicola et al. 2000],

—The *Ambient calculus*: `Ambient` [Cardelli and Gordon 2000; 1999; Cardelli et al. 1999].

Table I.   (sum-free) π-calculus syntax

p	$::=$	0	Null process
	$\mid$	$\overline{a}b.p$	Output
	$\mid$	$a(b).p$	Input
	$\mid$	$!p$	Replication
	$\mid$	$p \mid q$	Composition
	$\mid$	$(\nu a)p$	Restriction
	$\mid$	$[a = b]p$	Matching
	$\mid$	$[a \neq b]p$	Mismatching

To discuss the differences among the calculi, we do not consider a common formal framework. Instead, we propose to evaluate each calculus and its appropriateness to express metaphors for network-aware programming along three different guidelines:

—The programming abstractions the calculus suggests;

—The underlying programming model;

—The security mechanisms provided.

The rest of the paper is organized as follows. In Section 2, we briefly review the π-calculus and its asynchronous variant; both calculi can be considered as the common building blocks for all the other foundational calculi we are concerned with in the paper. These calculi, together with some simple programming examples, are presented in Sections 3, 4, 5 and 6. In Section 7, by means of a case study taken from an e-commerce scenario, we compare the programming abstractions of the calculi in terms of their suitability for structuring network-aware applications. In Section 8, we describe the security mechanisms provided by the calculi. In Section 9, we evaluate and compare the calculi. In Section 10, we briefly point out some related work.

2. π-CALCULUS AND ASYNCHRONOUS π-CALCULUS

The π-calculus [Milner et al. 1992] is the best known example of core calculus for mobility. The calculus is centered around the notion of *naming*: mobility is achieved via *name passing*. Hereafter, we assume as given an infinite set of names $\mathcal{N}$ and use lowercase latin letters $(a, b, \ldots, x, y, \ldots)$ as meta-variables over $\mathcal{N}$. As usual we identify two terms if one can be obtained from the other by alpha-renaming; indeed all the semantics issues will be defined up to the congruence induced by alpha-renaming, also when this is not explicitly mentioned.

The *syntax* of the π-calculus[1] is given in Table I. A process may be the void process, a process prefixed by an output action, a process prefixed by an input action, the (unbounded) replication of a process, the parallel composition of processes, the scope restriction of a name, or a process guarded by an equality or by a disequality between names. The prefix and the restriction combinators $a(b).-$ and $(\nu b)-$ act as binders for name b with scope the argument process. However, they have different natures: in the first case, b indicates the placeholders where the received name must be placed; in the second case, b is a new, private name. Notions of *free names* of a process p, $\mathtt{fn}(p)$, of *bound names* of p, $\mathtt{bn}(p)$, and of substitution arise

---

[1]Here we do not consider the *non-deterministic sum* operator because it is not used by any of the foundational calculi for network-aware programming.

Table II. π-calculus structural congruence

(MONOID)	| is associative and commutative, and 0 is its identity
(SCOPE)	$p \mid (\nu a)\, q \;=\; (\nu a)(p \mid q)$ if $a \notin \mathtt{fn}(p)$
(RES)	$(\nu a)\,(\nu b)\, p \;=\; (\nu b)\,(\nu a)\, p$
(NIL1)	$(\nu a)0 \;=\; 0$
(NIL2)	$!0 \;=\; 0$
(BANG)	$!p \;=\; p \mid !p$
(MATCH)	$[a = a]p \;=\; p$

Table III. π-calculus reduction relation

(COM) $a(b).p \mid \bar{a}c.q \longrightarrow p[c/b] \mid q$

$$(\text{PAR}) \quad \frac{p \longrightarrow p'}{p \mid q \longrightarrow p' \mid q}$$

$$(\text{RES}) \quad \frac{p \longrightarrow q}{(\nu a)p \longrightarrow (\nu a)q}$$

$$(\text{MISM}) \quad \frac{p \longrightarrow p' \quad a \neq b}{[a \neq b]p \longrightarrow p'}$$

$$(\text{CONG}) \quad \frac{p \equiv q \quad q \longrightarrow q' \quad q' \equiv p'}{p \longrightarrow p'}$$

as expected; $\mathtt{n}(p) = \mathtt{fn}(p) \cup \mathtt{bn}(p)$ is the set of *names* of p. We shall write $\mathtt{fn}(p, q)$ in place of $\mathtt{fn}(p) \cup \mathtt{fn}(q)$ (similarly for $\mathtt{bn}(\cdot)$ and $\mathtt{n}(\cdot)$). Finally, in $a(b)$ and in $\bar{a}b$, a is called the *subject* and b is called the *object*.

We assume the following syntactic conventions: $\bar{a}b.p \mid q$ stands for $(\bar{a}b.p) \mid q$, $a(b).p \mid q$ for $(a(b).p) \mid q$, $!p \mid q$ for $(!p) \mid q$, $(\nu\, a)p \mid q$ for $((\nu\, a)p) \mid q$ and $(\nu\, a_1 \ldots a_m)p$ for $(\nu\, a_1) \ldots (\nu\, a_m)p$. Moreover, trailing occurrences of 0 shall usually be omitted. Hereafter, a name declared *fresh* in a statement is assumed to be different from any other name there occurring.

The *operational semantics* of the π-calculus is defined using a structural congruence and a reduction relation. The *structural congruence*, $\equiv$, is defined as the least congruence relation over processes that satisfies the axioms in Table II. The structural congruence basically provides an equational algebra for manipulating and rearranging processes, thus simplifying the operational rules that define the reduction relation. For instance, process $a(b).\bar{b}c \mid (\nu d)\bar{a}d$ is rearranged to $(\nu d)(a(b).\bar{b}c \mid \bar{a}d)$ by opening the scope of the ν operator. Moreover, structural congruence performs some garbage collection of dead processes and handles replication ($!p$ stands for an unlimited number of copies of p running in parallel) and equality on names.

The *reduction relation* is defined by the rules in Table III. Rule (COM) says that an output and an input action with the same subject do synchronize. When the communication takes place, the output is consumed and the object of the output (i.e. the name exchanged in the communication) replaces the free occurrences (placeholders) of the object of the input action in its continuation process. Rules (PAR), (RES) and (MISM) state that the behaviour of a process is context invariant. Finally, rule (CONG) ensures that the structural equivalence does not modify the behaviour of processes.

The *asynchronous* π-calculus [Honda and Tokoro 1991; Boudol 1992; Amadio et al. 1996], π_a for short, is a simple variant of the π-calculus where asynchrony is achieved by imposing the void continuation to the output actions. In other

words, the (synchronous) π-calculus uses both input and output actions as prefixes while π_a has only input actions. Although from a theoretical point of view π_a is less expressive than π-calculus [Palamidessi 1997], π_a is still enough expressive in practice [Honda and Tokoro 1991].

The operational semantics of π_a can be defined like that of π-calculus; the only difference is that rule (COM) is replaced by rule

$$(\text{COM}_a) \qquad a(b).p \mid \bar{a}c \longrightarrow p[c/b].$$

The calculi considered in this paper build up on the *polyadic* versions of π-calculus [Milner 1993] and π_a. In these variants, *tuples* of names can be exchanged in communications. We will use $\tilde{\cdot}$ to denote a tuple of objects and $\{\tilde{\cdot}\}$ to denote the set of the components of $\tilde{\cdot}$. The input and output prefixes of polyadic π-calculus can then be written as $a(\tilde{b})$ and $\bar{a}\langle\tilde{b}\rangle$, respectively (when $\tilde{b}$ is empty, they become a and $\bar{a}\langle\rangle$).

To simplify the presentation of the examples, in all the calculi we consider in this paper, we assume the existence of a set of *basic data types*, such as integers, strings and booleans, that can be exchanged in communications. Moreover, when defining terms, we will sometimes write $A \stackrel{\triangle}{=} t$ to assign the name A to the term t. After that, A can be used in place of t to make other term definitions shorter. This notation is not intended to increase the expressive power of the languages (thus, for instance, A cannot occur in t) but only as a useful shorthand.

The π-calculus and its asynchronous variant π_a can naturally describe networks which reconfigure their communication linkages (net mobility)[2]. In the following example, borrowed from [Milner et al. 1992], we show three processes that dynamically change their connections. This example also shows that alpha-renaming is crucial to avoid captures of free names when the scope of the restriction operator changes. Consider the following π_a process

$$S \stackrel{\triangle}{=} c(x).p \mid (\nu a)(\bar{c}a \mid q \mid r)$$

and suppose that a occurs free in p and in q, and not in r. Let $b \notin \text{fn}(p \mid r \mid q)$ and $b \neq c$; by using alpha-renaming and (SCOPE), we can rearrange S as follows:

$$S \equiv (\nu b)(c(x).p \mid \bar{c}b \mid q[b/a] \mid r).$$

In this rearrangement the scope of the (private) name a is extruded to process p, and an alpha-conversion is necessary to avoid clashes with the free occurrences of a in p (which actually denote a different channel). To this aim, we take a *fresh* (i.e. not used before) name b and change the bound occurrences of a with b in $(\bar{c}a \mid q \mid r)$.

Now, by applying the reduction rules (COM_a), (PAR) and (RES) and the structural law (MONOID), we get the reduction

$$S \longrightarrow (\nu b)(p[b/x] \mid q[b/a] \mid r).$$

As a consequence of the communication at c, a new link (i.e. a new communication channel) is established between the existing processes p and q. This is obtained by

---

[2]However, process mobility can be elegantly codified [Sangiorgi 1992].
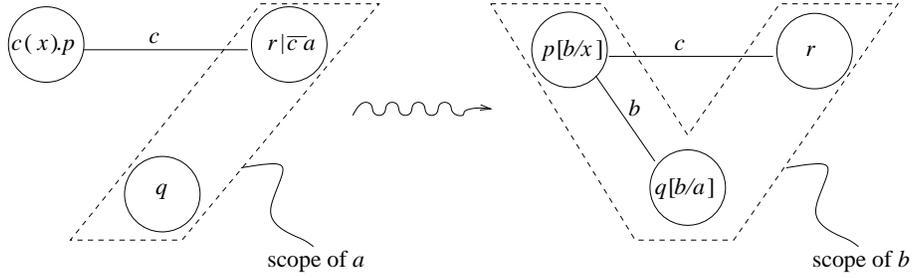
Fig. 1.   Evolution of process S

sending the private name b by which p and q may interact. The behaviour of S is pictorially represented in Figure 1.

3.   DISTRIBUTED π-CALCULUS

The Distributed π-calculus (Dπ for short) [Hennessy and Riely 1998] extends the polyadic π-calculus with explicit locations, located channels and with primitives for process mobility. Locations reflect the idea of having administrative domains and located channels can be thought of as channels under the control of certain authorities.

The *syntax*[3] of the calculus is in Table IV. To improve readability, we will use a, b, c ... as channel names, and h, k, ℓ, ... as location names; we use e, f, ... when the distinction does not play any role. Intuitively, a *system* consists of a set of allocated threads, $\ell[p]$, running independently in parallel, where the scope of some channel and location names can be restricted. *Threads* are essentially polyadic π-calculus processes that can additionally create new locations or names $((\nu\,e)p)$ and migrate to other locations $(\ell :: p)$. In Dπ tuples of allocated channels and location names can be transmitted over channels. The conditional (**if**) corresponds to the matching and mismatching operators of the π-calculus. The definitions of free and bound names are similar to those for π-calculus. By convention, $\ell :: p \mid q$ will stand for $(\ell :: p) \mid q$.

The *operational semantics* exploits a *structural equivalence*, $\equiv$, which is the least equivalence relation over processes that is closed under composition and restriction, and satisfies the laws in Table V and the corresponding laws for Dπ systems of laws (MONOID) and (RES) in Table II. The laws in Table V are essentially obtained by tailoring π-calculus laws to Dπ systems.

The *reduction relation* is defined by the rules in Table VI plus the corresponding rules for Dπ systems of rules (PAR), (RES) and (CONG) in Table III. Rule (MOVE) says that an agent p can move from location ℓ to k. Rule (COMM) says that two processes can communicate by performing complementary input and output actions at the same channel but only if they are located at the same location. This rule assumes the existence of a *pattern* substitution mechanism. We will not define this mechanism formally as it is the intuitive one, i.e. patterns and values are tuples

---

[3]In this section we do not consider the Dπ type system and its application to security. Therefore, we removed all the type expressions in the grammar while maintaining the "original" semantics.

Table IV.   Dπ syntax

Systems	$P - R$	::=	0	Null system
			$P \mid Q$	Composition
			$(\nu\, e)P$	Restriction
			$\ell[p]$	Allocated thread
Threads	$p - r$	::=	0	Null process
			$p \mid q$	Composition
			$(\nu\, e)p$	Restriction
			$u :: p$	Migration
			$\overline{u}\langle U \rangle.p$	Output
			$u(X).p$	Input
			$!p$	Replication
			if $u = v$ **then** p **else** q	Conditional
Identifiers	$u - w$	::=	$e \mid x$	
Patterns	$X - Z$	::=	$x \mid z[\tilde{x}] \mid \tilde{X}$	
Values	$U - W$	::=	$u \mid w[\tilde{u}] \mid \tilde{U}$	

Table V.   Dπ structural equivalence

(NIL)	$\ell[0]$	=	0	
(SPLIT)	$\ell[p \mid q]$	=	$\ell[p] \mid \ell[q]$	
(ITR)	$\ell[!p]$	=	$\ell[p] \mid \ell[!p]$	
(NEWC)	$\ell[(\nu\, a)p]$	=	$(\nu\, a)\ell[p]$	
(NEWL)	$\ell[(\nu\, k)p]$	=	$(\nu\, k)\ell[p]$	if $k \neq \ell$
(EXTR)	$Q \mid (\nu\, e)P$	=	$(\nu\, e)(Q \mid P)$	if $e \notin fn(Q)$

Table VI.   Dπ reduction relation

(MOVE)	$\ell[k :: p]$	$\longrightarrow$	$k[p]$
(COMM)	$\ell[\overline{a}\langle U \rangle.p] \mid \ell[a(X).q]$	$\longrightarrow$	$\ell[p] \mid \ell[q\{U/X\}]$
(THEN)	$\ell[\textbf{if } e = e \textbf{ then } p \textbf{ else } q]$	$\longrightarrow$	$\ell[p]$
(ELSE)	$\ell[\textbf{if } e = d \textbf{ then } p \textbf{ else } q]$	$\longrightarrow$	$\ell[q]$ if $e \neq d$

with the same structure. The conditional behaves as expected (rules (THEN) and (ELSE)). Notice that the conditional can be expressed in terms of the composition of the matching and mismatching constructs of the π-calculus.

To give a flavour of the Dπ programming model we show the code of a mobile counter, *Cnt*, that initially runs at location h. The counter is initialized to value n and uses the global names *inc*, *dec*, *val* and *jmp* to interact with the environment[4]:

$$Cnt \overset{\triangle}{=} h[(\nu a\, i)(\overline{a}\langle h\rangle.\overline{i}\langle n\rangle$$
$$\mid !a(\ell).i(x).\ell :: (\nu\, c)(\overline{c}\langle x\rangle$$
$$\mid !inc(z[y]).c(x).\overline{c}\langle x+1\rangle.z :: \overline{y}\langle\rangle$$
$$\mid !dec(z[y]).c(x).(\overline{c}\langle x-1\rangle \mid z :: \overline{y}\langle\rangle)$$
$$\mid !val(z[y]).c(x).(\overline{c}\langle x\rangle \mid z :: \overline{y}\langle x\rangle)$$
$$\mid jmp(\ell').c(x).h :: \overline{a}\langle\ell'\rangle.\overline{i}\langle x\rangle$$
$$))].$$

---

[4]In our examples we will abbreviate $(\nu\, a_1) \ldots (\nu\, a_n)P$ by $(\nu\, a_1 \ldots a_n)P$.

The counter can be regarded as an *abstract object*: a process willing to interact with *Cnt* can only use the interface, i.e. the global channel names *inc*, *dec*, *val* and *jmp*, and has no control over the private channels a and i. Clients can require standard services (i.e. *inc*, *dec* and *val*) by transmitting an allocated channel ($z[y]$); *Cnt* will return the result of the service at the channel y located at z. Clients can also ask the counter to move to a different location by sending the destination location at channel *jmp*.

We now explore the evolution of the counter. First, we rewrite process *Cnt* as

$$h[\ (\nu\,a\,i)(\ \overline{a}\langle h\rangle.\overline{i}\langle n\rangle\ \mid\ !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C)\ )\ ]$$

where

$$C \triangleq\ !inc(z[y]).c(x).\overline{c}\langle x+1\rangle.z :: \overline{y}\langle\rangle$$
$$\mid !dec(z[y]).c(x).(\overline{c}\langle x-1\rangle \mid z :: \overline{y}\langle\rangle)$$
$$\mid !val(z[y]).c(x).(\overline{c}\langle x\rangle \mid z :: \overline{y}\langle x\rangle)$$
$$\mid jmp(\ell').c(x).h :: \overline{a}\langle\ell'\rangle.\overline{i}\langle x\rangle.$$

Two atomic steps (local synchronizations over channels a and i) are used to set the initial location h of the counter and to initialize it with the value n. A further step is needed to actually install it and obtain:

$$h[\ (\nu\,a\,i)(\ (\nu\,c)(\overline{c}\langle n\rangle \mid C)\ \mid\ !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C)\ )\ ].$$

A possible client for the counter is process

$$m[h :: \overline{val}\langle m[v]\rangle \mid Q].$$

The client requires the service *val* by sending to the counter location h a process that asks for the service and provides a return allocated channel $m[v]$. The counter will send the result back to the client by spawning the mobile process $m :: \overline{v}\langle n'\rangle$ that moves itself to the client location m and delivers the value n' of the counter at channel v.

Another possible client is process

$$m[h :: \overline{jmp}\langle\ell''\rangle \mid Q].$$

The client calls for the counter to change its location by sending to the counter location h a process that asks for the service at channel *jmp* and provides the new location ℓ'' for counter. Let

$$C' \triangleq\ !inc(z[y]).c(x).\overline{c}\langle x+1\rangle.z :: \overline{y}\langle\rangle$$
$$\mid !dec(z[y]).c(x).(\overline{c}\langle x-1\rangle \mid z :: \overline{y}\langle\rangle)$$
$$\mid !val(z[y]).c(x).(\overline{c}\langle x\rangle \mid z :: \overline{y}\langle x\rangle).$$

The resulting computation is

$$h[\ (\nu\,a\,i)(\ (\nu\,c)(\overline{c}\langle n\rangle \mid C) \mid !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C))] \mid m[h :: \overline{jmp}\langle\ell''\rangle \mid Q]$$
$$\longrightarrow^* h[(\nu\,a\,i)((\nu\,c)(C' \mid h :: \overline{a}\langle\ell''\rangle.\overline{i}\langle n\rangle) \mid !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C))] \mid m[Q]$$
$$\longrightarrow^* h[(\nu\,a\,i)((\nu\,c)C' \mid \ell'' :: (\nu\,c)(\overline{c}\langle n\rangle \mid C) \mid !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C))] \mid m[Q]$$
$$\longrightarrow h[(\nu\,a\,i)((\nu\,c)C' \mid !a(\ell).i(x).\ell :: (\nu\,c)(\overline{c}\langle x\rangle \mid C))] \mid m[Q] \mid \ell'' :: (\nu\,c)(\overline{c}\langle n\rangle \mid C)$$

As a consequence of the interaction with the client, the counter changes its location, but leaves its original definition, together with some dangling services C' that

Table VII. Djoin syntax

Nets	S, T	::=	$\mathcal{D} \vdash_\varphi \mathcal{P}$	*Located solution*
		|	$S \parallel T$	*Distributed CHAM*
Processes	P, Q	::=	$x\langle \widetilde{v} \rangle$	*Asynchronous message*
		|	**def** D **in** P	*Local definition*
		|	$P \mid Q$	*Parallel composition*
		|	**go** $\langle \mathsf{a} \rangle ; P$	*Migration request*
		|	0	*Null process*
Definition	D, E	::=	$J \rhd P$	*Elementary clause*
		|	$\mathsf{a}[D : P]$	*Location constructor*
		|	$D \wedge E$	*Simultaneous definition*
Join patterns	J, J'	::=	$x\langle \widetilde{v} \rangle$	*Asynchronous reception*
		|	$J \mid J'$	*Joining messages*
Names	v, v'	::=	x	*Channel names* $(x \in \mathcal{N})$
		|	a	*Locations names* $(\mathsf{a} \in \mathcal{L})$

cannot be used anymore, at the old location h. Each time the counter is demanded to change location, it forwards the request to its original definition at location h.

Notice that to access a remote resource (e.g. the counter) one has to know where it is located and that thread definitions (i.e. replications) never change their location.

4. DISTRIBUTED JOIN-CALCULUS

The join-calculus [Fournet and Gonthier 1996] is an "extended subset" of π_a which combines the three operators for input, restriction and replication into a single operator, called *definition*, that has the additional capability of describing atomic *joint* reception of values from different communication channels. The Distributed join-calculus (Djoin for short) [Fournet et al. 1996] adds abstractions to express process distribution and process mobility.

The *syntax* of the calculus (taken from [Fournet et al. 1999]) is given in Table VII. A set of *location names* $\mathcal{L}$, ranged over by a, b, ..., and such that $\mathcal{L} \cap \mathcal{N} = \emptyset$ is additionally assumed. Location names can be exchanged in communications. We will use ψ and φ to denote finite strings of location names, i.e. elements of $\mathcal{L}^*$.

A *net* is a multiset of located solutions. A located solution, $\mathcal{D} \vdash_\varphi \mathcal{P}$, consists of a location label φ, of a multiset of running processes $\mathcal{P}$ and of a multiset of active rules $\mathcal{D}$, which define the possible reductions of processes. A *process* may send an asynchronous message on a name, define new names and reaction rules, fork in parallel components and move its execution to another location using **go**. A *definition* is composed of reaction rules, $J \rhd P$, and location constructors, $\mathsf{a}[D : P]$, separated by the $\wedge$ operator. $J \rhd P$ triggers the execution of process P when the join-pattern J is recognized; $\mathsf{a}[D : P]$ defines a new (sub)location a.

Definitions are the binding mechanism for names. A definition entirely describes the behaviour of its defined names. Hence, a process that receives a channel name may use it to send messages but cannot add new behaviours (i.e. elementary clauses) for that name. Definitions have a strict *lexical* discipline (i.e. the behaviour of names is statically defined). The *received* variables, $\mathtt{rv}(J)$, are the names to

Table VIII.   Defined, received and free variables

J:	$\mathtt{dv}(x\langle\tilde{y}\rangle)$	$\overset{\text{def}}{=} \{x\}$	$\mathtt{rv}(x\langle\tilde{y}\rangle)$	$\overset{\text{def}}{=} \{\tilde{y}\}$
	$\mathtt{dv}(J \mid J')$	$\overset{\text{def}}{=} \mathtt{dv}(J) \cup \mathtt{dv}(J')$	$\mathtt{rv}(J \mid J')$	$\overset{\text{def}}{=} \mathtt{rv}(J) \uplus \mathtt{rv}(J')$
D:	$\mathtt{dv}(J \,\rhd\, P)$	$\overset{\text{def}}{=} \mathtt{dv}(J)$	$\mathtt{fv}(J \,\rhd\, P)$	$\overset{\text{def}}{=} \mathtt{dv}(J) \cup (\mathtt{fv}(P) - \mathtt{rv}(J))$
	$\mathtt{dv}(D \wedge E)$	$\overset{\text{def}}{=} \mathtt{dv}(D) \cup \mathtt{dv}(E)$	$\mathtt{fv}(D \wedge E)$	$\overset{\text{def}}{=} \mathtt{fv}(D) \cup \mathtt{fv}(E)$
	$\mathtt{dv}(\mathtt{a}[D \,:\, P])$	$\overset{\text{def}}{=} \{\mathtt{a}\} \uplus \mathtt{dv}(D)$	$\mathtt{fv}(\mathtt{a}[D \,:\, P])$	$\overset{\text{def}}{=} \{\mathtt{a}\} \cup \mathtt{fv}(D) \cup \mathtt{fv}(P)$
P:	$\mathtt{fv}(x\langle\tilde{v}\rangle)$	$\overset{\text{def}}{=} \{x\} \cup \{\tilde{v}\}$	$\mathtt{fv}(\mathbf{go}\ \langle\mathtt{a}\rangle;\mathrm{P})$	$\overset{\text{def}}{=} \{\mathtt{a}\} \cup \mathtt{fv}(P)$
	$\mathtt{fv}(\mathbf{0})$	$\overset{\text{def}}{=} \emptyset$	$\mathtt{fv}(P \mid Q)$	$\overset{\text{def}}{=} \mathtt{fv}(P) \cup \mathtt{fv}(Q)$
	$\mathtt{fv}(\mathbf{def}\ D\ \mathbf{in}\ P)$	$\overset{\text{def}}{=} (\mathtt{fv}(P) \cup \mathtt{fv}(D)) - \mathtt{dv}(D)$		

which the sent messages are bound; the *defined* variables in a join pattern or in a definition, $\mathtt{dv}(J)$ and $\mathtt{dv}(D)$, are the names that are bound by the construct; the *free* variables, $\mathtt{fv}(P)$ and $\mathtt{fv}(D)$, are all the names which are not bound. Received, defined and free variables can be easily defined, as expected, by structural induction. Their definitions are in Table VIII, where $\uplus$ denotes disjoint union. For instance, according to the lexical scoping, in $\mathbf{def}\ x\langle\tilde{v}\rangle \rhd P_1\ \mathbf{in}\ P_2$ the scope of the received variables $\{\tilde{v}\}$ is P_1 whereas the scope of the defined variable x extends to the whole definition (i.e. both P_1 and P_2). Moreover, in $(D \vdash_\varphi P) \parallel S$ the scope of D extends to the whole net.

It is possible to define a *sublocation* relation over $\mathcal{L}^*$ by saying that if ψ is a prefix of φ then $\vdash_\varphi$ is a sublocation of $\vdash_\psi$. This means that at any time of the computation there is a tree of locations. Each location corresponds to a single physical site and if a location $\mathtt{a}$ occurs in the subtree of a location $\mathtt{b}$, then $\mathtt{a}$ is on the same site as $\mathtt{b}$. The fact that the same location cannot be given to two different sites, together with the fact that communication channels can be defined only once (a peculiar feature of the `join-calculus`), leads to the notion of well-formed nets. Formally, a net is *well-formed* if it respects the following two syntactical conditions:

—*uniqueness of locations*: a location $\mathtt{a}$ is defined only once in any definition D (i.e. there is exactly one local solution $\vdash_{\varphi\mathtt{a}}$ for each location name $\mathtt{a}$ appearing in a label);

—*uniqueness of receptors*: a defined channel x may only appear in the join-patterns of one location.

The *operational semantics* of `Djoin` was originally presented in [Fournet et al. 1996] in the style of a *distributed* chemical abstract machine [Berry and Boudol 1992] (that is, one chemical abstract machine for each location in the tree). Here, in order to preserve the presentation style, we give an alternative formulation in terms of structural rules and reduction semantics (see, also, [Levy 1997]). From the chemical metaphor, we maintain the simplifying assumption that the presentation of every rule assumes an implicit context and only involve those parts of (the multisets in) located solutions that change by the effect of the presented rule.

The *structural equivalence*, $\equiv$, is the least equivalence relation that is closed under composition of located solutions, of multisets of definitions and of multisets of processes, and satisfies the laws in Table IX. The first three structural laws state

Table IX.   `Djoin` structural equivalence

(JOIN)	$\vdash_\varphi P \mid Q$	$=$	$\vdash_\varphi P, Q$	
(NULL)	$\vdash_\varphi 0$	$=$	$\vdash_\varphi$	
(AND)	$D \wedge E \vdash_\varphi$	$=$	$D, E \vdash_\varphi$	
(DEF)	$\vdash_\varphi \mathbf{def}\ D\ \mathbf{in}\ P$	$=$	$D\sigma_{\mathbf{dv}} \vdash_\varphi P\sigma_{\mathbf{dv}}$	$range(\sigma_{\mathbf{dv}})$ fresh
(LOC)	$\mathsf{a}[D : P] \vdash_\varphi$	$=$	$\vdash_\varphi\ \|\ \{D\} \vdash_{\varphi\mathsf{a}} \{P\}$	a frozen

Table X.   `Djoin` reduction relation

(JOIN)	$J \rhd P \vdash_\varphi J\sigma_{\mathbf{rv}}$	$\longrightarrow$	$J \rhd P \vdash_\varphi P\sigma_{\mathbf{rv}}$
(COMM)	$\vdash_\varphi x\langle \tilde{v}\rangle\ \|\ J \rhd P \vdash$	$\longrightarrow$	$\vdash_\varphi\ \|\ J \rhd P \vdash x\langle \tilde{v}\rangle \qquad x \in \mathsf{dv}(J)$
(MOVE)	$\mathsf{a}[D : P \mid \mathbf{go}\ \langle \mathsf{b}\rangle;Q] \vdash_\varphi\ \|\ \vdash_{\psi\mathsf{b}}$	$\longrightarrow$	$\vdash_\varphi\ \|\ \mathsf{a}[D : P \mid Q] \vdash_{\psi\mathsf{b}}$

that $\mid$ and $\wedge$ are commutative and associative operators, and that 0 is the identity of parallel composition. Law (DEF) allows processes to activate reaction rules. The side condition is necessary to avoid name clashes: $\sigma_{\mathbf{dv}}$ replaces all the names defined by D, $\mathsf{dv}(D)$, with names fresh with respect to the set of names defined in the rest of the net. Finally, law (LOC) introduces a new location whenever location a is *frozen* that, by definition, means that there is no other solution in the net annotated with a string of the form $\varphi\mathsf{a}\psi$ (i.e., there are no sublocations of $\varphi\mathsf{a}$ in the net). Both side conditions are necessary to preserve well-formedness.

The *reduction relation* is defined by the rules in Table X plus the corresponding rules for the `Djoin` operators $\mid$, $\wedge$ and $\|$ of rule (PAR) (with the obvious side conditions for avoiding name clashes with the rest of the net) and the corresponding rule for `Djoin` nets of rule (CONG) in Table III. Rule (JOIN) activates a (guarded) process when a matching join-pattern is recognized. Rule (COMM) says that in order to emit on a given remote channel, it is first necessary to ship the message to the remote location where the channel is defined. Message transport is deterministic, point-to-point (because of uniqueness of name definitions) and transparent to the processes in the net. Finally, rule (MOVE) says that a location (and all its sublocations) can move to another existing location. In fact, locations are used to express mobile agents (by mapping agents to locations)[5].

We end this section with some simple examples of `Djoin` programming. The first example consists of a system where a client asks for a service to a remote server. Assume that s and c are the server and the client location names respectively. The system can be coded as follows:

$$S \triangleq \mathsf{s}[p\langle d,r\rangle \rhd P_1 \mid r\langle\rangle : 0] \wedge \mathsf{c}[k\langle\rangle \rhd P_2 : p\langle v,k\rangle] \vdash_\varepsilon.$$

By using the semantic rules we can derive the following computation (here it is assumed that $r \notin \mathtt{fv}(P_1)$)

$$
\begin{aligned}
S\ &\equiv\ p\langle d,r\rangle \rhd P_1 \mid r\langle\rangle \vdash_\mathsf{s}\ \|\ k\langle\rangle \rhd P_2 \vdash_\mathsf{c} p\langle v,k\rangle \\
&\longrightarrow\ p\langle d,r\rangle \rhd P_1 \mid r\langle\rangle \vdash_\mathsf{s} p\langle v,k\rangle\ \|\ k\langle\rangle \rhd P_2 \vdash_\mathsf{c}
\end{aligned}
$$

---

[5]Locations have been introduced in [Fournet et al. 1996] mainly as a tool to provide a simple model of failures in distributed systems. In the absence of failures, the execution of processes is independent of locations.

$$\equiv \quad p\langle d, r\rangle \, \triangleright \, P_1 \mid r\langle\rangle \, \vdash_{\mathsf{s}} \, P_1\{v/d\} \mid k\langle\rangle \quad \| \quad k\langle\rangle \, \triangleright \, P_2 \vdash_{\mathsf{c}}$$
$$\longrightarrow \quad p\langle d, r\rangle \, \triangleright \, P_1 \mid r\langle\rangle \, \vdash_{\mathsf{s}} \, P_1\{v/d\} \quad \| \quad k\langle\rangle \, \triangleright \, P_2 \vdash_{\mathsf{c}} k\langle\rangle$$
$$\equiv \quad p\langle d, r\rangle \, \triangleright \, P_1 \mid r\langle\rangle \, \vdash_{\mathsf{s}} \, P_1\{v/d\} \quad \| \quad k\langle\rangle \, \triangleright \, P_2 \vdash_{\mathsf{c}} P_2$$

Channel p is used to send the request to the server. After making the request, the client waits for the results at channel k.

Our second example shows how to define a counter cnt process in Djoin. The code of a process with a local definition of cnt is:

$$\textbf{def } cnt\langle x, k\rangle\triangleright$$
$$\quad \textbf{def } inc\langle k\rangle \mid count\langle n\rangle \triangleright count\langle n+1\rangle \mid k\langle\rangle$$
$$\quad \wedge \quad dec\langle k \mid count\langle n\rangle\rangle \triangleright count\langle n-1\rangle \mid k\langle\rangle$$
$$\quad \wedge \quad val\langle k \mid count\langle n\rangle\rangle \triangleright count\langle n\rangle \mid k\langle n\rangle$$
$$\quad \textbf{in } count\langle x\rangle \mid k\langle inc, dec, val\rangle$$
$$\textbf{in } \ldots$$

A client process gains the ability of accessing the counter by passing it an initial value and a continuation channel (e.g. the client may take the form **def** … **in** $cnt\langle 5, k\rangle$). The counter may be transformed into a mobile counter, mob_cnt, by allocating channels inc, dec and val at a new location a. The counter, i.e. channels inc, dec and val, will move as long as location a moves. Location a is a sublocation of the current location of mob_cnt, but will become a sublocation of b after moving with **go** $\langle\mathsf{b}\rangle$, before executing $count\langle x\rangle \mid k\langle inc, dec, val\rangle$. The code of a process with a local definition of mob_cnt is:

$$\textbf{def } mob\_cnt\langle\mathsf{b}, x, k\rangle\triangleright$$
$$\quad \textbf{def } \mathsf{a}[count\langle n\rangle \mid inc\langle k\rangle \triangleright count\langle n+1\rangle \mid k\langle\rangle$$
$$\quad \wedge \quad count\langle n\rangle \mid dec\langle k\rangle \triangleright count\langle n-1\rangle \mid k\langle\rangle$$
$$\quad \wedge \quad count\langle n\rangle \mid val\langle k\rangle \triangleright count\langle n\rangle \mid k\langle n\rangle$$
$$\quad : \textbf{go } \langle\mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle]$$
$$\quad \textbf{in } 0$$
$$\textbf{in } \ldots$$

We now explore the evolution of a net with the mobile counter definition. First, let us use D as shorthand for definition

$$count\langle n\rangle \mid inc\langle k\rangle \triangleright count\langle n+1\rangle \mid k\langle\rangle$$
$$\wedge \quad count\langle n\rangle \mid dec\langle k\rangle \triangleright count\langle n-1\rangle \mid k\langle\rangle$$
$$\wedge \quad count\langle n\rangle \mid val\langle k\rangle \triangleright count\langle n\rangle \mid k\langle n\rangle$$

and assume that the net has the following form:

$$mob\_cnt\langle\mathsf{b}, x, k\rangle\triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle\mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } 0 \vdash_{\mathsf{s}} 0$$
$$\|$$
$$D_{k_1, k_2} \vdash_{\mathsf{u}} mob\_cnt\langle\mathsf{u}, 3, k_1\rangle$$

where the mobile counter is initially located at the server location s and u is the user location (which contains the definition of the user process and of channels k_1 and k_2).

By rule (COMM), the net can evolve to

$$mob\_cnt\langle\mathsf{b}, x, k\rangle\triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle\mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } 0$$

$$\vdash_{\mathsf{S}} mob\_cnt\langle \mathsf{u}, 3, k_1 \rangle$$
$$\|$$
$$D_{k_1,k_2} \vdash_{\mathsf{u}} \mathsf{0}$$

By rule (JOIN), we get the net

$$mob\_cnt\langle \mathsf{b}, x, k\rangle \triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle \mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } \mathsf{0}$$
$$\vdash_{\mathsf{S}} \textbf{def } \mathsf{a}[D : \textbf{go } \langle \mathsf{u}\rangle; count\langle 3\rangle \mid k_1\langle inc, dec, val\rangle] \textbf{ in } \mathsf{0}$$
$$\|$$
$$D_{k_1,k_2} \vdash_{\mathsf{u}} \mathsf{0}$$

which, by rule (DEF), is structurally equivalent to

$$mob\_cnt\langle \mathsf{b}, x, k\rangle \triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle \mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } \mathsf{0},$$
$$\mathsf{a}[D : \textbf{go } \langle \mathsf{u}\rangle; count\langle 3\rangle \mid k_1\langle inc, dec, val\rangle] \vdash_{\mathsf{S}} \mathsf{0}$$
$$\|$$
$$D_{k_1,k_2} \vdash_{\mathsf{u}} \mathsf{0}$$

Now, by rule (MOVE), the net reduces to

$$mob\_cnt\langle \mathsf{b}, x, k\rangle \triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle \mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } \mathsf{0} \vdash_{\mathsf{S}} \mathsf{0}$$
$$\|$$
$$D_{k_1,k_2}, \mathsf{a}[D : count\langle 3\rangle \mid k_1\langle inc, dec, val\rangle] \vdash_{\mathsf{u}} \mathsf{0}$$

which, by rule (LOC), is structurally equivalent to

$$mob\_cnt\langle \mathsf{b}, x, k\rangle \triangleright \textbf{def } \mathsf{a}[D : \textbf{go } \langle \mathsf{b}\rangle; count\langle x\rangle \mid k\langle inc, dec, val\rangle] \textbf{ in } \mathsf{0} \vdash_{\mathsf{S}} \mathsf{0}$$
$$\|$$
$$D_{k_1,k_2} \vdash_{\mathsf{u}} k_1\langle inc, dec, val\rangle$$
$$\|$$
$$D[k_1/k] \vdash_{\mathsf{ua}} count\langle 3\rangle$$

where the counter has moved to a sublocation ua of user's location u.

An experimental implementation of `Djoin` called `JoCaml`, is available at the URL `http://pauillac.inria.fr/jocaml/`. `JoCaml` integrates the `Djoin` programming model with the Objective Caml programming language. `JoCaml` programs are translated in an intermediate code which consists of the Objective Caml intermediate code extended with libraries to support `Djoin` constructs.

5. KLAIM

KLAIM [De Nicola et al. 1998] is an asynchronous higher–order process calculus which extends the Linda [Gelernter 1985; Carriero and Gelernter 1989] coordination paradigm (processes communicate via a shared multiset of tuples) to distributed and mobile processes. Basically, KLAIM is a variant of π_a whose actions are the Linda primitives enriched with information about the addresses of the nodes where processes and tuples are allocated.

The *syntax* of the calculus is reported in Table XI. There are three types of values: basic values, sites (i.e. net addresses) and processes. Basic values[6] are simply

---

[6]Here we consider a simplified version of KLAIM that, differently from [De Nicola et al. 1998; 1999; De Nicola et al. 2000], does not include value *expressions* and *localities*.

Table XI.   KLAIM Syntax

Nets	N	$::=$	$s ::_{\rho} P$	Single node
		$\mid$	$N_1 \parallel N_2$	Net composition
Processes	P	$::=$	0	Null process
		$\mid$	$a.P$	Action prefixing
		$\mid$	$P_1 \mid P_2$	Process composition
		$\mid$	X	Process variable
		$\mid$	$A\langle \widetilde{V} \rangle$	Process invocation
Actions	a	$::=$	$\textbf{out}(t)@u$	Output
		$\mid$	$\textbf{in}(t)@u$	Input
		$\mid$	$\textbf{read}(t)@u$	Read
		$\mid$	$\textbf{eval}(P)@u$	Process creation
		$\mid$	$\textbf{newloc}(\widetilde{u})$	Node creation
Tuples	t	$::=$	$f \mid f,t$	
Fields	f	$::=$	$V \mid \,! Z$	
Values	V	$::=$	$v \mid P \mid Z$	
Variables	Z	$::=$	$x \mid X \mid u$	

elements of the set of names $\mathcal{N}$ (we shall use v, v_1, v_2 ... as generic basic values and x, y, z ... as generic variables for basic values). We also assume the existence of a set of *sites* S (ranged over by s, s_1, s_2, ...) and of a set of *site variables* $\mathcal{U}$ (ranged over by u, u_1, u_2 ...), that also includes the distinguished variable $\texttt{self}$. $\texttt{self}$ is used by processes to refer to their current execution site. Processes can be defined parametrically by equations of the form $A(\widetilde{Z}) \stackrel{\text{def}}{=} P$, where A is a *process identifier* ($A \in \Psi$, the set of process identifiers) and P is a process which may contain recursive calls of A (with the obvious parameter passing substitution). For each process identifier A there exists a *single* defining equation.

Variables occurring in process terms can be bound by action prefixes and process equations. More precisely, prefixes $\textbf{in}(t)@u.\_$, $\textbf{read}(t)@u.\_$ and $\textbf{newloc}(\widetilde{u})$ act as binders for variables in the formal fields of t and in $\{\widetilde{u}\}$, respectively. Definition $A(\widetilde{Z}) \stackrel{\text{def}}{=} P$ is considered as a binder for variables $\{\widetilde{Z}\}$.

Tuples are (finite) sequences of fields. Fields can be actual fields (i.e. values) and formal fields. Formal fields are denoted by "$!Z$". Notice that the syntactic category of values does not include sites, hence sites cannot explicitly occur in the code of processes.

KLAIM processes may perform three different kinds of actions: accessing tuple spaces (i.e. multisets of tuples), spawning processes and creating new nodes in a net. The (non–blocking) operation $\textbf{out}(t)@u$ adds the tuple resulting from the evaluation of t to the tuple space (TS, for short) located at u. Two (possibly blocking) operations, $\textbf{in}(t)@u$ and $\textbf{read}(t)@u$, access tuples in the TS located at u. The operation $\textbf{in}(t)@u$ evaluates t and looks for a matching tuple t' in the TS at u; if such a t' exists, it is removed from the TS. The corresponding values of t' are then assigned to the variables in the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The operation $\textbf{read}(t)@u$ differs from $\textbf{in}(t)@u$ only in that the matching tuple t' is not

Table XII. Matching rules			
$match(V, V)$	$match(!\,x, v)$	$match(!\,X, P)$	$match(!\,u, s)$

$$\frac{match(f_1, f_2)}{match(f_2, f_1)} \qquad \frac{match(f_1, f_2) \quad match(et_1, et_2)}{match((f_1, et_1), (f_2, et_2))}$$

removed from the TS at u. New threads of executions are dynamically activated through the operation **eval**$(P)@u$ that spawns a process (whose code is given by P) at the node named u. New nodes in a net can be created through the operation **newloc**$(\widetilde{u})$ and then accessed via the site variables $\widetilde{u}$. This operation is not indexed with a site identifier because it is always executed at the current execution node.

Nets are collections of nodes. A *node* is a term of the form $s ::_\rho P$, where the site s is the node address, P gives the processes running at s, and ρ is the *allocation environment*, namely a function mapping site variables to sites. ρ links the locality variables which occur free in P to certain sites. The idea is that allocation environments act as proxy mechanisms of the nodes in the net. Processes have not direct access to nodes and can get knowledge of a site either through their (local) allocation environment or through communication with other processes (which, again, exploits other allocation environments).

A node $s ::_\rho P$ is *well-formed* whenever $\rho(\texttt{self}) = s$. A net N of well-formed nodes is *well-formed* when, for any pair of nodes $s ::_\rho P$ and $s' ::_{\rho'} P'$, we have that if $s = s'$ then $\rho = \rho'$. We will only consider well-formed nets.

The *operational semantics* models tuples in a tuple space as independent processes. To this purpose KLAIM syntax is extended with processes of the form **out**(et) to denote *evaluated tuples* (referred to as et). Tuple spaces are then processes in a special form: parallel composition of evaluated tuples. Moreover, to avoid using environments for storing the bindings of site variables to sites, we allow processes to refer to sites directly (i.e. sites can occur where free site variables can).

KLAIM operational semantics exploits an *evaluation* function for tuples, $[\![\,\cdot\,]\!]_\rho$, that coincides with the identity function a part for the following cases:

$$[\![\,f, t\,]\!]_\rho = [\![\,f\,]\!]_\rho, [\![\,t\,]\!]_\rho \qquad [\![\,Z\,]\!]_\rho = \begin{cases} \rho(Z) & \text{if } Z \in dom(\rho) \\ undef & \text{otherwise} \end{cases} \qquad [\![\,P\,]\!]_\rho = P\{\rho\}$$

where $P\{\rho\}$ denotes the process term obtained by replacing in P any free occurrence of each site variable $u \in dom(\rho)$ that is not within the argument of an **eval** operation with $\rho(u)$.

We also need a matching predicate, *match*, defined by the rules in Table XII, that, given two tuples as arguments, checks whether they do match. Matching occurs when the argument tuples have the same number of fields and corresponding fields have matching values or variables: variables match any value of the same type, and two values match only if they are identical.

The *structural equivalence*, $\equiv$, is the least equivalence relation closed under the rules in Table XIII. The structural laws express that $\|$ is commutative and associative, that the null process can be safely removed and that it is always possible to distribute the processes located on the same node over clones of that node. The last rule deals with process invocation and says that if $A(\widetilde{Z}) \stackrel{\text{def}}{=} P$ then invocation

Table XIII.   KLAIM structural equivalence

$$N_1 \parallel N_2 \;=\; N_2 \parallel N_1$$

$$(N_1 \parallel N_2) \parallel N_3 \;=\; N_1 \parallel (N_2 \parallel N_3)$$

$$s ::_\rho P \;=\; s ::_\rho (P \mid \mathbf{0})$$

$$s ::_\rho (P_1 \mid P_2) \;=\; s ::_\rho P_1 \parallel s ::_\rho P_2$$

$$A\langle \widetilde{V} \rangle \;=\; P[\widetilde{V}/\widetilde{Z}] \qquad \text{if } A(\widetilde{Z}) \stackrel{\text{def}}{=} P$$

$A\langle \widetilde{V} \rangle$, with $\widetilde{V}$ of the same form as $\widetilde{Z}$, behaves like the body P of the process where $\widetilde{Z}$ has been replaced by $\widetilde{V}$.

The *reduction relation* is defined by the rules in Table XIV plus the corresponding rule for KLAIM nets of rule (CONG) in Table III. We use the following notations: ℓ denotes both sites and site variables; $\rho(\ell) = s$ denotes that either $\ell = s$ or ℓ is a site variable that ρ maps to site s; $[\![\, t \,]\!]_\rho = et$ denotes that the evaluation of tuple t using ρ succeeds, i.e. no component field gives rise to *undef*, and returns the evaluated tuple et; $st(N)$ denotes the set of sites (of the nodes) in N; if $\{\widetilde{s}\} = \{s_1, \ldots, s_n\}$ then $\Pi_{s' \in \{\widetilde{s}\}} s' ::_{\rho[s'/\mathtt{self}]} \mathbf{0}$ denotes the net $s_1 ::_{\rho[s_1/\mathtt{self}]} \mathbf{0} \parallel \ldots \parallel s_n ::_{\rho[s_n/\mathtt{self}]} \mathbf{0}$ where $\rho[s_i/\mathtt{self}]$ is the allocation environment ρ updated with the binding between $\mathtt{self}$ and s_i.

Let us now comment on the rules in Table XIV. Rule (OUT) says that the execution of an **out** operation adds an evaluated tuple to a tuple space. The local allocation environment is used both to determine the name of the node where the tuple must be placed and to evaluate the argument tuple. This implies that if the argument tuple contains a field with a process, the corresponding field of the evaluated tuple contains the process resulting from the evaluation of its site variables. Hence, processes in a tuple are transmitted after the interpretation of their free site variables through the local allocation environment. This corresponds to having a *static scoping* discipline for the (remote) generation of tuples. A *dynamic scoping* strategy is adopted for the **eval** operation, described by rule (EVAL). In this case the site variables of the spawned process are not interpreted using the local allocation environment: the linking of site variables is done at the remote site. Rule (IN) says that a process can perform an **in** action by synchronizing with a process which represents a matching tuple et'. The result of this synchronization is that tuple et' is consumed, i.e. the corresponding process becomes $\mathbf{0}$, and its values are used to replace, within the process which has performed the **in** operation, the free occurrences of the variables in the corresponding formal fields of et (this substitution is denoted by $[et'/et]$). Rule (READ) can be interpreted similarly to (IN). Only notice that, while **in** modifies the tuple space at s', **read** does not; in the conclusion of rule (READ) the accessed tuple is still at s'. Rule (NEW) describes the creation of new nodes. Their addresses can be freely chosen among those sites different from s. The environment of a new node is derived from that of the creating one with the obvious update for the $\mathtt{self}$ variable. Therefore, the new node inherits all the bindings of the creating node. Rule (PAR) is the analogous for KLAIM nets of rule (PAR) in Table III; the side condition avoids site name clashes when new nodes are created.

We end this section with some examples of KLAIM programming. The first

Table XIV.   KLAIM reduction relation

$$(\text{OUT}) \quad \frac{\rho(\ell) = s' \qquad [\![\, t \,]\!]_\rho = et}{s ::_\rho \mathbf{out}(t)@\ell.P \parallel s' ::_{\rho'} P' \longrightarrow s ::_\rho P \parallel s' ::_{\rho'} (P' \mid \mathbf{out}(et))}$$

$$(\text{EVAL}) \quad \frac{\rho(\ell) = s' \qquad [\![\, t \,]\!]_\rho = et}{s ::_\rho \mathbf{eval}(Q)@\ell.P \parallel s' ::_{\rho'} P' \longrightarrow s ::_\rho P \parallel s' ::_{\rho'} (P' \mid Q)}$$

$$(\text{IN}) \quad \frac{\rho(\ell) = s' \qquad [\![\, t \,]\!]_\rho = et \qquad match(et, et')}{s ::_\rho \mathbf{in}(t)@\ell.P \parallel s' ::_{\rho'} \mathbf{out}(et') \longrightarrow s ::_\rho P[et'/et] \parallel s' ::_{\rho'} \mathbf{0}}$$

$$(\text{READ}) \quad \frac{\rho(\ell) = s' \qquad [\![\, t \,]\!]_\rho = et \qquad match(et, et')}{s ::_\rho \mathbf{read}(t)@\ell.P \parallel s' ::_{\rho'} \mathbf{out}(et') \longrightarrow s ::_\rho P[et'/et] \parallel s' ::_{\rho'} \mathbf{out}(et')}$$

$$(\text{NEW}) \quad \frac{\{\widetilde{s}\} \subseteq (\mathcal{S} \setminus \{s\})}{s ::_\rho \mathbf{newloc}(\widetilde{u}).P \longrightarrow s ::_\rho P[\widetilde{s}/\widetilde{u}] \parallel \Pi_{s' \in \{\widetilde{s}\}} s' ::_{\rho[s'/\mathtt{self}]} \mathbf{0}}$$

$$(\text{PAR}) \quad \frac{N_1 \longrightarrow N_1' \qquad (st(N_1') \setminus st(N_1)) \cap st(N) = \emptyset}{N_1 \parallel N \longrightarrow N_1' \parallel N}$$

example will be also useful to point out the differences between the two forms of mobility provided by KLAIM. One form is mobility with static scoping: a process moves along the nodes of a net with a fixed binding of resources. The other form is mobility with dynamic scoping: process movements break the links to the local resources. For instance, consider a net consisting of two sites s_1 and s_2. A client process C is allocated at site s_1 and a server process S is allocated at site s_2. The server can accept processes for execution. The client sends process Q to the server. The code of processes is:

$$C \stackrel{\text{def}}{=} \mathbf{out}(Q)@u.\mathbf{0}$$
$$Q \stackrel{\text{def}}{=} \mathbf{in}(''\text{foo}'', !x)@\mathtt{self}.\mathbf{out}(''\text{foo}'', x+1)@\mathtt{self}.\mathbf{0}$$
$$S \stackrel{\text{def}}{=} \mathbf{in}(!X)@\mathtt{self}.X$$

The behaviour of the processes above depends on the meaning of u and $\mathtt{self}$. It is the allocation environment that establishes the links between site variables and sites. Here, we assume that the allocation environment of site s_1, ρ_1, maps $\mathtt{self}$ into s_1 and u into s_2, while the allocation environment of site s_2, ρ_2, maps $\mathtt{self}$ into s_2. Finally, we assume that the tuple spaces located at s_1 and s_2 both contain the tuple $(''\text{foo}'', 1)$. The following KLAIM program represents the net described above:

$$s_1 ::_{\rho_1} C \mid \mathbf{out}(''\text{foo}'', 1) \quad \| \quad s_2 ::_{\rho_2} S \mid \mathbf{out}(''\text{foo}'', 1).$$

After the execution of $\mathbf{out}(Q)@u$, the tuple space at site s_2 contains a tuple where the code of process Q is stored. Indeed, it is the process Q' that is stored in the tuple, where:

$$Q' \stackrel{\text{def}}{=} \mathbf{in}(''\text{foo}'', !x)@s_1.\mathbf{out}(''\text{foo}'', x+1)@s_1.\mathbf{0}$$

as the site variables occurring in Q are evaluated using the environment at site s_1 where the action **out** has been executed. Hence, when executed at the server's site the mobile process Q increases tuple $''\text{foo}''$ at the client's site.

To move process Q for execution at s_2 with a dynamic scoping strategy the client code should be $\mathbf{eval}(Q)@u.\mathbf{0}$. Indeed, when $\mathbf{eval}(Q)@u$ is executed, Q is spawned at the remote node *without* evaluating its site variables according to the allocation environment ρ_1. Thus, the execution of Q will depend only on the allocation environment ρ_2 and Q will increase tuple $''\text{foo}''$ at the server's site.

In our second example a mobile process is used to collect the mailboxes of a user over distinct nodes (e.g. accounts). The code of the mobile process is:

$$Fwd(id, u_h) \stackrel{\text{def}}{=} \mathbf{in}(''\text{mbox}'', id, !x)@\mathtt{self}.$$
$$\mathbf{out}(''\text{mbox-at}'', id, \mathtt{self}, x)@u_h.\mathbf{0}$$
$$|$$
$$\mathbf{read}(''\text{remote-mbox}'', id, !u)@u_h.$$
$$\mathbf{eval}(Fwd(id, u_h))@u.\mathbf{0}$$

Process Fwd takes as parameters the identificator id and the home address u_h of the user. When executed, it withdraws the id's mailbox at the current execution site and sends it to address u_h; concurrently, it looks for the next site u to visit and, then, spawns a copy of itself at u.

Consider a net with three sites, s, s_1 and s_2. Assume that process $Fwd(''\text{usr}'', s)$ is running at site s which is the home site of the user $U(''\text{usr}'')$, and that the net has the following structure:

$$s ::_\rho U(''\text{usr}'') \mid Fwd(''\text{usr}'', s) \mid$$
$$\quad \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_1) \mid P \parallel$$
$$s_1 ::_{\rho_1} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_1) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_2) \mid Q \parallel$$
$$s_2 ::_{\rho_2} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_2) \mid R$$

A possible evolution of this net is illustrated below.

$$s ::_\rho \quad U(''\text{usr}'') \mid Fwd(''\text{usr}'', s)$$
$$\quad \mid \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_1) \mid P \parallel$$
$$s_1 ::_{\rho_1} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_1) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_2) \mid Q \parallel$$
$$s_2 ::_{\rho_2} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_2) \mid R$$
$$\downarrow^*$$
$$s ::_\rho \quad U(''\text{usr}'') \mid \mathbf{in}(''\text{mbox}'', ''\text{usr}'', !x)@\mathtt{self}.\mathbf{out}(''\text{mbox-at}'', ''\text{usr}'', \mathtt{self}, x)@s.\mathbf{0}$$
$$\quad \mid \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_1) \mid P \parallel$$
$$s_1 ::_{\rho_1} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_1) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_2) \mid$$
$$\quad Q \mid Fwd(''\text{usr}'', s) \parallel$$
$$s_2 ::_{\rho_2} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_2) \mid R$$
$$\downarrow^*$$
$$s ::_\rho \quad U(''\text{usr}'') \mid \mathbf{out}(''\text{mbox-at}'', ''\text{usr}'', s, m) \mid \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_1)$$
$$\quad \mid \mathbf{out}(''\text{mbox-at}'', ''\text{usr}'', s_1, m_1) \mid P \parallel$$
$$s_1 ::_{\rho_1} \mathbf{out}(''\text{remote-mbox}'', ''\text{usr}'', s_2) \mid Q \parallel$$
$$s_2 ::_{\rho_2} \mathbf{out}(''\text{mbox}'', ''\text{usr}'', m_2) \mid R \mid Fwd(''\text{usr}'', s)$$

An experimental implementation of Klaim, called X-Klaim, can be downloaded at `http://rap.dsi.unifi.it/klaim.html`. The implementation consists of two

Table XV. **Ambient** syntax

Processes	P, Q	::=	$(\nu\, n)P$	Restriction
		|	0	Inactivity
		|	$P \mid Q$	Composition
		|	$!P$	Replication
		|	$M[P]$	Ambient
		|	$M.P$	Capability action
		|	$(x).P$	Input action
		|	$\langle M \rangle$	Output action
Capabilities	M	::=	x	Variable
		|	n	Name
		|	$in\ M$	Can enter M
		|	$out\ M$	Can exit M
		|	$open\ M$	Can open M
		|	ε	Empty path
		|	$M.M$	Path

layers: the X-KLAIM compiler and the intermediate language KLAVA that is obtained by extending the Java language [Arnold and Gosling 1997] with a new package, called Klava. The Klava package [Bettini et al. 1998] contains all the classes which implement the X-KLAIM runtime system and operations.

6. AMBIENT

The **Ambient** calculus [Cardelli and Gordon 2000] relies on the notion of *ambient* that can be thought of as a bounded environment where processes cooperate. An ambient has a name, a collection of local agents and a collection of subambients. Ambients can be moved as a whole under the control of agents; these are confined to ambients.

The *syntax* of the calculus is reported in Table XV. **Ambient** processes use *capabilities* for controlling interaction. Indeed, by using capabilities, an ambient can allow other ambients to perform certain operations over it without having to reveal its actual name (which would give a lot of control over it). A name n is a capability to enter, exit or create a new copy of an ambient named n. Capability *in M* serves for entering into ambient M, *out M* for exiting out of M and *open M* for opening up M. The possession of one or all of these capabilities is insufficient to reconstruct the original ambient name from which they were extracted. Multiple capabilities can be combined into paths.

The process primitives (restriction, inactivity, composition and replication) are derived from π-calculus. We only remark that the restriction operator introduces new ambient names and, differently from π-calculus, it does not create new channel names. Process $n[P]$ is an ambient with name n and process P running inside. Nothing prevents the existence of two or more ambients with the same name. Process $M.P$ executes the action corresponding to capability M and then behaves like P. Communication is asynchronous and anonymous (no process or communication channel is explicitly referred), and takes place locally within a single ambient. The objects that can be communicated are ambient names and capabilities, that may be thought of as rights to commit some operations on ambient names. An output action releases an ambient name or a capability into the local ether of the

Table XVI. **Ambient** structural equivalence

(RES AMB)	$(\nu\,n)(m[P]) = m[(\nu\,n)P]$ if $n \neq m$
(EPS) $\varepsilon.P = P$	(CONC) $(M_1.M_2).P = M_1.(M_2.P)$

Table XVII. **Ambient** reduction relation

(IN)	$m[in\ n.P \mid Q] \mid n[R] \longrightarrow n[m[P \mid Q] \mid R]$
(OUT)	$m[n[out\ m.P \mid Q] \mid R] \longrightarrow n[P \mid Q] \mid m[R]$
(OPEN)	$open\ n.P \mid n[Q] \longrightarrow P \mid Q$
(COM)	$(x).P \mid \langle M \rangle \longrightarrow P[M/x]$
(AMB)	$\dfrac{P \longrightarrow Q}{n[P] \longrightarrow n[Q]}$

surrounding ambient. An input action captures an ambient name or a capability from the local ether and binds it to a variable within a scope. Both restriction and input actions are binding operators for names and variables, respectively, with the obvious scopes.

We assume the following syntactic conventions: $(x).P \mid Q$ stands for $((x).P) \mid Q$, $M.P \mid Q$ for $(M.P) \mid Q$ and $n[]$ for $n[0]$.

The *structural equivalence*, $\equiv$, is the least equivalence relation that satisfies the rules in Tables II and XVI. The first law in Table XVI permits moving restriction inside ambients while the last two laws deal with sequences of capabilities.

The *reduction relation* is given by the rules in Table XVII plus the corresponding for **Ambient** processes of rules (PAR), (RES) and (CONG) in Table III. The first two rules says that ambients are moved from inside an ambient m. In particular, (IN) says that action $in\ n$ instructs the ambient named m surrounding $in\ n.P$ to enter a sibling ambient named n. If no sibling n can be found, the operation blocks until a time when such a sibling exists. If more than one n sibling exists, any one of them can be chosen. Rule (OUT) says that action $out\ m$ instructs the ambient n surrounding $out\ m.P$ to exit out of its parent ambient named m. If the parent is not named m, the operation blocks until a time when such a parent exists. Note that when a process P causes its surrounding ambient to move, all the other processes contained in its ambient are moved too. Rule (OPEN) says that action $open\ n$ has the effect of dissolving the boundary of an ambient named n located at the same level as $open\ n.P$, unleashing the ambient's content. If no ambient n is found, the operation blocks until a time when such an ambient exists. If more than one ambient n exists, any one of them can be chosen. Rule (COM) states that communications are local; this means that a process performing an input, $(x).P$, and an asynchronous output, $\langle M \rangle$, synchronize only if they are enclosed within the same ambient. Rule (AMB) propagates reductions inside nested ambients.

As an example of the expressiveness of the **Ambient** primitives, let us show how they can be used to encode *objective moves*, i.e. actions that 'make ambients move from the outside' [Cardelli and Gordon 2000]. Objective moves are similar in spirit to the primitives for mobility of Dπ and KLAIM, as opposed to those of **Djoin** and **Ambient**. Informally, $mv\ in\ n.P$ should allow P to enter into ambient n, and, symmetrically, $mv\ out\ n.P$ should allow P to move out of ambient n.

Let us start defining the following ambients

$$n^{\downarrow}[P] \ \triangleq\ n[P \mid !open\ enter]$$
$$n^{\uparrow}[P] \ \triangleq\ n[P] \mid !open\ exit$$
$$n^{\downarrow\uparrow}[P] \ \triangleq\ n[P \mid !open\ enter] \mid !open\ exit$$

where *enter* and *exit* are assumed to be two distinguished (global) names. Now, we can define the following 'objective' actions

$$mv\ in\ n.P \ \triangleq\ (\nu\,k)k[in\ n.enter[out\ k.(P \mid open\ k)]]$$
$$mv\ out\ n.P \ \triangleq\ (\nu\,k)k[out\ n.exit[out\ k.(P \mid open\ k)]]$$

When these objective actions are used in conjunction with the above defined ambients, the operational rules of the calculus allow us to deduce the following reductions:

$$mv\ in\ n.P \mid n^{\downarrow\uparrow}[Q] \longrightarrow^* n^{\downarrow\uparrow}[P \mid Q]$$
$$n^{\downarrow\uparrow}[mv\ out\ n.P \mid Q] \longrightarrow^* P \mid n^{\downarrow\uparrow}[Q]$$

that correspond to the intended semantics of the objective moves.

We end this section with two short examples. The first example shows how lists of messages can be implemented in `Ambient`. The list of messages $l \triangleq [M_1, \ldots, M_n]$ is represented as the ambient $l^{\downarrow\uparrow}[\langle M_1 \rangle \mid l^{\downarrow\uparrow}[\langle M_2 \rangle \ldots l^{\downarrow\uparrow}[\langle M_n \rangle] \ldots]]$. Operations over lists are defined as:

$$void\ l \ \triangleq\ l^{\downarrow\uparrow}[]$$
$$cons\ M\ l \ \triangleq\ (\nu\,k)(k[l^{\downarrow\uparrow}[\langle M \rangle]] \mid k^{\uparrow}[in\ l.mv\ out\ k.open\ k.in\ l \mid mv\ out\ l.open\ k])$$
$$head\ l\ m \ \triangleq\ (\nu\,k)(m[open\ k] \mid mv\ in\ l.(x).mv\ out\ l.open\ l.k[in\ m.\langle x \rangle])$$
$$tail\ l \ \triangleq\ mv\ in\ l.(x).mv\ out\ l.open\ l.\mathbf{0}$$

where $k \notin fn(M) \cup \{l, m\}$. For example, with these definitions, we can show that

$$cons\ M\ l \mid void\ l \longrightarrow^* l^{\downarrow\uparrow}[\langle M \rangle \mid l^{\downarrow\uparrow}[]]$$

Indeed, we have that

$cons\ M\ l \mid void\ l$
$\quad \longrightarrow (\nu\,k)(k[l^{\downarrow\uparrow}[\langle M \rangle]] \mid l^{\downarrow\uparrow}[k^{\uparrow}[mv\ out\ k.open\ k.in\ l \mid mv\ out\ l.open\ k]])$
$\quad \longrightarrow^* (\nu\,k)(k[l^{\downarrow\uparrow}[\langle M \rangle]] \mid l^{\downarrow\uparrow}[open\ k.in\ l \mid k^{\uparrow}[mv\ out\ l.open\ k]])$
$\quad \longrightarrow (\nu\,k)(k[l^{\downarrow\uparrow}[\langle M \rangle]] \mid l^{\downarrow\uparrow}[in\ l \mid mv\ out\ l.open\ k])$
$\quad \longrightarrow^* (\nu\,k)(k[l^{\downarrow\uparrow}[\langle M \rangle]] \mid l^{\downarrow\uparrow}[in\ l] \mid open\ k)$
$\quad \longrightarrow (\nu\,k)(l^{\downarrow\uparrow}[\langle M \rangle] \mid l^{\downarrow\uparrow}[in\ l])$
$\quad \longrightarrow^* (\nu\,k)(l^{\downarrow\uparrow}[\langle M \rangle \mid l^{\downarrow\uparrow}[]])$
$\quad \equiv l^{\downarrow\uparrow}[\langle M \rangle \mid l^{\downarrow\uparrow}[]]$

where the last step holds because k does not occur free in $l^{\downarrow\uparrow}[\langle M \rangle \mid l^{\downarrow\uparrow}[]]$.

As last example, consider the case of a process which wants to enter a "secure" ambient, i.e. an ambient whose name w is restricted. The following program (borrowed from the firewall example of [Cardelli and Gordon 2000]) describes the protocol, based on passwords k and k', that allows Q to enter w. The third name k'' is necessary to confine Q thus preventing it to interfere with the protocol.

$$(\nu w)(w[k[out\ w.in\ k'.in\ w] \mid open\ k'.open\ k''.P]) \mid k'[open\ k.k''[Q]]$$
$$\equiv$$
$$(\nu w)(w[k[out\ w.in\ k'.in\ w] \mid open\ k'.open\ k''.P] \mid k'[open\ k.k''[Q]])$$
$$\downarrow$$
$$(\nu w)(w[open\ k'.open\ k''.P] \mid k[in\ k'in\ w] \mid k'[open\ k.k''[Q]])$$
$$\downarrow$$
$$(\nu w)(w[open\ k'.open\ k''.P] \mid k'[k[in\ w] \mid open\ k.k''[Q]])$$
$$\downarrow$$
$$(\nu w)(w[open\ k'.open\ k''.P] \mid k'[in\ w \mid k''[Q]])$$
$$\downarrow$$
$$(\nu w)(w[open\ k'.open\ k''.P \mid k'[k''[Q]]])$$
$$\downarrow^*$$
$$(\nu w)(w[P \mid Q])$$

An experimental implementation of the `Ambient` calculus, called Ambit, can be found at `http://www.luca.demon.co.uk/Ambit/Ambit.html` and consists of just an applet `Java`. A distributed implementation of the calculus is presented in [Fournet et al. 1999]. The implementation relies on a formal translation of `Ambient` in `Djoin` and uses `JoCaml` as implementation language.

7. AN ELECTRONIC MARKETPLACE

We aim at evaluating the calculi presented in the previous sections by also taking into account their programming abstractions for implementing distributed applications. To this purpose, in this section we will describe a simple application for the electronic commerce and present simple implementations in each calculus.

To make the presentations clearer, we assume that each calculus provides programmers with a typical control structure of programming languages: a conditional primitive of the form **if** b **then** P **else** Q (written **if** b **then** P when $Q = 0$). Moreover, since we are mainly interested to mobility control, we make a further assumption for simplifying data management and suppose that each calculus also provides programmers with a data type *list* equipped with the classical operations over lists: append an element to the end $l\#e$, get the first element $hd(l)$, get the list after removal of its first element $tl(l)$. Both our simplifying assumptions are justified by the fact that all the calculi we consider are *Turing powerful*, hence the new constructs do not increase their expressive power. Another useful syntactic convention we will use is the following: if P_i for $i \in \{1, \ldots, n\}$ are processes then $\prod_{i=1}^{n} P_i$ will stand for $P_1 \mid \cdots \mid P_n$.

The scenario we want to model is pictorially represented in Figure 2. A market place client, *Buyer*, asks to an information point, *Market*, for the list of camera shops that are within a given geographical area. *Buyer* initially only knows the *Market*'s address while, obviously, *Market* knows the addresses of all camera shops. Once *Buyer* has the list, it starts searching for the shop in the list that offers a certain model of camera at the lowest price. Mobility naturally arises since once *Buyer* has obtained the list, it can send the agent *Collector* that performs the
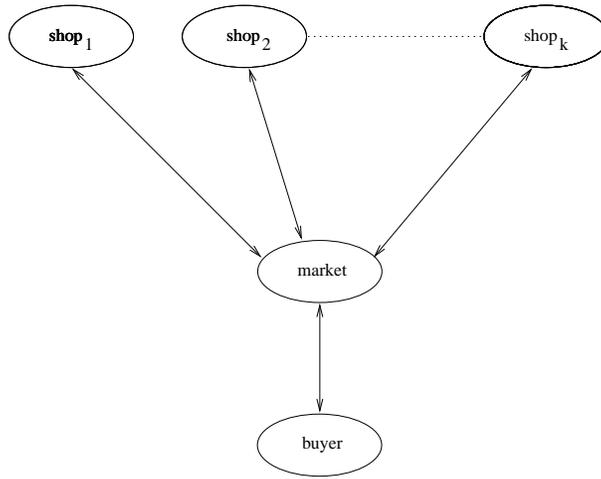
Fig. 2.   An electronic marketplace scenario

search. *Collector* will interact with each shop in the list and compute the lowest price for the camera. When all the shops in the list have been visited, the agent will send the result of the search back to *Buyer*.

The electronic marketplace is a paradigmatic example that shows the advantages of code mobility with respect to other "more traditional" interaction paradigms. On the one hand, code mobility permits a higher concurrency degree and increases asynchrony between clients and servers. For instance, while *Collector* is running, *Buyer* can concurrently perform tasks that do not depend on the result of the search. Moreover, every execution thread of *Collector* can potentially run on different machines. On the other hand, code mobility can also be used in presence of transient disconnections and network failures because it strictly reduces the number of times the network is used for remote communications. Indeed, instead of remotely asking for the price of the wanted camera to each shop, *Buyer* sends a process that visits each shop in turn and locally inquires for the camera price.

The implementations of the scenario described above in each of the process calculi we are considering are presented in the following subsections. We make the following simplifying assumptions:

—*Buyer* has no behaviour which does not depend on the result of the search;

—*Market* only handles camera shops (their number is k);

—each shop can sell n different models of cameras and the wanted one is among these;

—whenever a camera is not stocked, the shop sets the relative price to *max_price*, a numeric constant that represents a very high price (a price that no Buyer would ever spend for that camera).

7.1   Dπ implementation

The buyer process, *Buyer*, that runs at location b, behaves as follows:

—it sends to location m (that of *Market*) a request for the list of camera shops within its geographical area gab, together with a return address $collh[res]$ of the form $location[channel]$,

—it starts the execution of a mobile process, *Collector*, which, together with some initialization values, migrates to location $collh$, and

—it waits for the result, of the form $< price, shop >$, of the search at channel $best$.

$$Buyer \triangleq b[\ (\nu\ collh, res, init)$$
$$(\ m :: \overline{sh\_list}\langle gab, collh[res]\rangle$$
$$|\ collh :: (\overline{init}\langle max\_price, nowhere\rangle\ |\ Collector)$$
$$|\ best(howmuch, where).\ \ldots\ )\ ]$$

The (mobile) process *Collector* iterates the following behaviour:

—it waits for a list of shops at channel res;

—when it receives the list, it reads the actual best price and shop at channel $init$;

—if the list is empty, it returns the result of the search to the buyer location b at channel $best$;

—otherwise, it migrates to the first shop in the list, takes the price of the wanted camera and activates another copy of itself by passing it the list of the shops that still must be visited and the appropriate information about best price and shop, depending on the fact that the new price is lower than the one currently stored or not.

$$Collector \triangleq\ !res(sl).init(price, where).$$
$$\textbf{if } sl = empty \textbf{ then } b :: \overline{best}\langle price, where\rangle$$
$$\textbf{else } hd(sl) :: camera(p).$$
$$\textbf{if } p < price \textbf{ then } collh :: \overline{res}\langle tl(sl)\rangle.\overline{init}\langle p, hd(sl)\rangle$$
$$\textbf{else } collh :: \overline{res}\langle tl(sl)\rangle.\overline{init}\langle price, where\rangle$$

The jth shop is programmed as a process, $Shop_j$, that is located at s_j and behaves as follows:

—whenever prompted at channel ga_shop_j, it sends its geographical area ga_j and position s_j to location m at channel sp received along ga_shop_j,

—concurrently, at channel $camera_i$, for $i \in \{1, \ldots, n\}$, it sends the price of the corresponding camera.

$$Shop_j \triangleq s_j[\ !ga\_shop_j(sp).m :: \overline{sp}\langle ga_j, s_j\rangle\ |\ \prod_{i=1}^{n} !\overline{camera_i}\langle pr_i^j\rangle\ ]$$

The market process, *Market*, located at m, iterates the following behaviour:

—whenever prompted at channel sh_list, it activates a copy of itself for finding the list of camera shops within the geographical area gab received along channel sh_list;

—the new copy uses channel sl to manage the shop list and channel c as a counter for the number of shops with which interaction already took place, and behaves as follows:

—it asks to each shop handled by *Market* for its geographical area and position;

—if the geographical area of the shop is gab, the shop position is included in the list;

—whenever all the k shops have been considered, the list is returned to the location and the channel received along channel sh_list.

$$Market \triangleq m[\ !sh\_list(gab, z[res]).(\nu\ c, sl, sp_1, \ldots, sp_n)$$

$$(\overline{sl}\langle empty \rangle \mid \overline{c}\langle 0 \rangle \mid \prod_{j=1}^{k} s_j :: ga\_shop_j\langle sp_j \rangle \mid$$

$$\prod_{j=1}^{k} sp_j(ga_j, s_j).\ \textbf{if}\ ga_j = gab$$

$$\textbf{then}\ sl(x).\overline{sl}\langle x\# s_j \rangle.c(y).\textbf{if}\ y = k - 1$$
$$\textbf{then}\ sl(w).z :: \overline{res}\langle w \rangle$$
$$\textbf{else}\ \overline{c}\langle y + 1 \rangle$$
$$\textbf{else}\ c(y).\textbf{if}\ y = k - 1$$
$$\textbf{then}\ sl(w).z :: \overline{res}\langle w \rangle$$
$$\textbf{else}\ \overline{c}\langle y + 1 \rangle$$

$$)\ ]$$

The overall system is obtained by composing *Buyer*, *Market* and all $Shop_j$, for $j \in \{1, \ldots, k\}$, in such a way that shops locations s_j and channels ga_shop_j, used for initial interaction between *Market* and $Shop_j$, are not visible to *Buyer*.

$$System \triangleq Buyer \mid (\nu\ s_1, \ldots, s_k, ga\_shop_1, \ldots, ga\_shop_k)(Market \mid \prod_{j=1}^{k} Shop_j)$$

7.2 Djoin implementation

All of the Djoin processes we are going to introduce are given in the form of a location constructor, hence, technically, they all are *definitions*.

Let us start describing process *Buyer*. When location b is created, *Buyer*

—gives rise to a new location constructor for process *Collector* and to a reaction rule that will react whenever the result of the search is found;

—sends a request for the list of camera shops within the geographical area gab, together with a return address res, at channel sh_list;

—sends the appropriate initialization values to the mobile agent *Collector* at channel $init$.

$Buyer \stackrel{\triangle}{=} \mathsf{b}[\ Collector$
$\qquad \wedge\ best\langle howmuch, where\rangle \rhd \ldots$
$\qquad : sh\_list\langle gab, res\rangle \mid init\langle max\_price, nowhere\rangle\ ]$

When location c is created, *Collector* will give rise to a reaction rule that is fired whenever a shop list sl is returned at channel res and, at the same time, a price and a shop location are given at channel $init$. Such a reaction rule allows *Collector* to iterate its behaviour until all the shops in sl have been visited. The shop list sl returned by *Market* is implemented as a list of pairs of the form $< location, name >$. Each element of a pair can be selected by using the projection operators: $\_\downarrow_1$ returns the first element and $\_\downarrow_2$ the second one. *Collector* uses

—the location of a shop for visiting it and for creating a sublocation c of the shop, and

—the name of a shop for sending the price request, together with a return channel ret, to it.

Finally, *Collector* activates again its reaction rule with values that depend on the new price for the camera.

$Collector \stackrel{\triangle}{=} \mathsf{c}[\ res\langle sl\rangle \mid init\langle price, where\rangle \rhd$
$\qquad\qquad \textbf{if } sl = empty \textbf{ then } best\langle price, where\rangle$
$\qquad\qquad\quad \textbf{else go } \langle (hd(sl)\downarrow_1)\rangle;\ \textbf{def } ret\langle p\rangle \rhd$
$\qquad\qquad\qquad\qquad\qquad \textbf{if } p < price$
$\qquad\qquad\qquad\qquad\qquad\quad \textbf{then } res\langle tl(sl)\rangle \mid init\langle p, hd(sl)\downarrow_1\rangle$
$\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } res\langle tl(sl)\rangle \mid init\langle price, where\rangle$
$\qquad\qquad\qquad\qquad\qquad \textbf{in } hd(sl)\downarrow_2 \langle camera, ret\rangle$
$\qquad\quad : \mathsf{0}\ ]$

Each shop has two kinds of reactions. Whenever prompted at channel ga_shop_j, the shop returns, along channel sp_j received along ga_shop_j, its geographical area ga_j, its location s_j and channel $shop_j$ at which the price list can be looked at. Whenever prompted at channel $shop_j$, the shop returns the price of *camera* at the return channel ret.

$Shop_j \stackrel{\triangle}{=} \mathsf{s}_j[\ ga\_shop_j\langle sp_j\rangle \rhd sp_j\langle ga_j, \mathsf{s}_j, shop_j\rangle$
$\qquad\qquad \wedge\ shop_j\langle camera, ret\rangle \rhd \prod_{i=1}^{n} \textbf{if } camera = camera_i \textbf{ then } ret\langle pr_i\rangle$
$\qquad\qquad : \mathsf{0}\ ]$

Market is programmed just as a rule that reacts each time the list of the shops within a given geographical area gab is requested at channel sh_list. The process activated by the rule behaves like its counterpart in $\mathrm{D}\pi$. It uses the same channels sl and c with the same meaning. The shop list in $\mathtt{Djoin}$ is a list of pairs $< location, name >$, as we said before.

$$Market \stackrel{\triangle}{=} \mathsf{m}[\ sh\_list\langle gab, res\rangle \rhd$$

$$\mathbf{def} \bigwedge_{j=1}^{k} sp_j\langle ga_j, s_j, shop_j\rangle \rhd \ \mathbf{if}\ ga_j = gab$$

$$\mathbf{then}\ app\langle s_j, shop_j\rangle$$
$$\mathbf{else}\ incr\langle c\rangle$$
$$\wedge\ \ sl\langle x\rangle \mid app\langle s, shop\rangle \rhd sl\langle x\#(s, shop)\rangle \mid incr\langle c\rangle$$
$$\wedge\ \ sl\langle x\rangle \mid send\langle res\rangle \rhd res\langle x\rangle$$
$$\wedge\ \ c\langle y\rangle \mid incr\langle c\rangle \rhd c\langle y + 1\rangle$$
$$\wedge\ \ c\langle y\rangle \rhd \ \mathbf{if}\ y = k\ \mathbf{then}\ send\langle res\rangle$$

$$\mathbf{in}\ c\langle 0\rangle \mid sl\langle empty\rangle \mid \prod_{j=1}^{k} ga\_shop_j\langle sp_j\rangle$$

$$: 0\ ]$$

Finally, the system is made by joining the definition of *Buyer*, *Market* and $Shop_j$, for $j \in \{1, \ldots, k\}$, and by allocating all them at the same initial location. Notation $\bigwedge_{j=1}^{k} Shop_j$ stands for $Shop_1 \wedge \cdots \wedge Shop_k$.

$$System \stackrel{\triangle}{=} Buyer \wedge Market \wedge \bigwedge_{i=j}^{k} Shop_j\ \vdash_\varepsilon 0$$

7.3 KLAIM implementation

Process *Buyer* is parameterized with respect to its geographical area *gab*, location name *m* for the site of *Market* and the model of camera *camera* that has to be searched for. The process behaves as follows:

—it creates a new, private site referred to by variable *res* where the result of the search will be waited for;

—then, it puts a tuple at the tuple space referred to by *m* to ask for the list of camera shops within *gab*;

—finally, it sends for execution at the site referred to by *res* a process that, whenever it is able to locally access the list of shops, invokes the mobile agent *Collector* by passing it the appropriate initialization values: the list of shops to visit *sl*, a price *max_price* and a location *nowhere* to be refined, the model of camera and the return address.

$$Buyer(gab, m, camera) \stackrel{\mathrm{def}}{=}$$
$$\mathbf{newloc}(res).\mathbf{out}(''\mathrm{sh\_list}'', gab, res)@m.$$
$$(\ \mathbf{eval}(\mathbf{in}(''\mathrm{sl}'', !sl)@\texttt{self}.$$
$$Collector\langle sl, max\_price, nowhere, camera, res\rangle)@res$$
$$\mid \mathbf{in}(''\mathrm{best}'', !howmuch, !where)@res.\ \ldots\ )$$

Process *Collector* behaves as expected: it visits each shop in the list, computes the lowest price for the camera (while storing information about the shop that offers

it) and terminates when the list is empty (all shops have been visited). To make its presentation clearer, we take advantage of the possibility offered by KLAIM to define processes and give name *Test* to the process which is argument of the **eval** primitive.

$Collector(sl, price, where, camera, res) \stackrel{\text{def}}{=}$
   if $sl = empty$
      then out($''$best$''$, $price$, $where$)@res
      else eval($Test\langle tl(sl), price, where, camera, res\rangle$)@$hd(sl)$

$Test(sl, price, where, camera, res) \stackrel{\text{def}}{=}$
   read($camera, !p$)@$\texttt{self}$.
   if $p < price$
      then $Collector\langle sl, p, \texttt{self}, camera, res\rangle$
      else $Collector\langle sl, price, where, camera, res\rangle$

Process $Shop_j$, that represents a single shop, is parameterized with respect to its geographical area ga and the price of each camera pr_i^j offered by the shop. $Shop_j$ behaves as its Dπ counterpart and gives information about its geographical area and its price list.

$Shop_j(ga_j, pr_1^j, \ldots, pr_n^j) \stackrel{\text{def}}{=}$
   $\mathbf{in}(''\text{ga\_shop}'', !sls)@\texttt{self}.\mathbf{out}(j, ga_j, \texttt{self})@sls$
   $\mid \prod_{i=1}^{n}\mathbf{out}(''\text{camera}_i'', pr_i^j)@\texttt{self}$

Process *Market* is parameterized with respect to a list of variables $u_1, \ldots, u_n$ that refer to the sites of the shops. When the process is asked for a shop list, it invokes a copy of itself (in order to be able to serve several requests at the same time) and, concurrently, serves the request by creating a new site, referred to by sls, and by sending process Mkl for execution at sls. Mkl, a part for the communication mechanism used, behaves as its Dπ counterpart: it computes the shop list sl, by using counter c for storing the number of shops that have been already considered, and, when all shops have been taken into account, puts a tuple containing sl at the site received together with the shop list request.

$Market(u_1, \ldots, u_k) \stackrel{\text{def}}{=}$
   $\mathbf{in}(''\text{sh\_list}'', !gab, !res)@\texttt{self}.$
   $(Market\langle u_1, \ldots, u_k\rangle \mid \mathbf{newloc}(sls).\mathbf{eval}(Mkl\langle gab, u_1, \ldots, u_k\rangle)@sls)$

$Mkl(gab, u_1, \ldots, u_k) \stackrel{\text{def}}{=}$
   $\mathbf{out}(''\text{c}'', 0)@\texttt{self}.\mathbf{out}(''\text{sl}'', empty)@\texttt{self}.$
   $\prod_{j=1}^{k} \mathbf{out}(''\text{ga\_shop}'', \texttt{self})@u_j.\mathbf{in}(j, !ga_j, !u_j)@\texttt{self}.$

$$\textbf{if } gab = ga_j$$
$$\textbf{then } \textbf{in}(''\text{sl}'', !x)@\texttt{self}.\textbf{out}(''\text{sl}'', x\#u_j)@\texttt{self}.$$
$$\textbf{in}(''\text{c}'', !y)@\texttt{self}.\textbf{out}(''\text{c}'', y+1)@\texttt{self}$$
$$\textbf{else } \textbf{in}(''\text{c}'', !y)@\texttt{self}.\textbf{out}(''\text{c}'', y+1)@\texttt{self}$$
$$| \ \textbf{in}(''\text{c}'', k)@\texttt{self}.\textbf{in}(''\text{sl}'', !w)@\texttt{self}.\textbf{out}(''\text{sl}'', w)@res$$

The overall system is defined by a net with a node for each of processes *Buyer*, *Market* and $Shop_j$, for $j \in \{1, \ldots, k\}$. Nodes allocation environments are defined in such a way that, initially, *Buyer* can only access to the site of *Market* while the latter is able to access to the site of each $Shop_j$.

$$System \stackrel{\triangle}{=} \ s_b ::_{\{s_b/\texttt{self}, s_m/m\}} Buyer\langle gab, m, '' \text{camera}'' \rangle$$
$$\| \ s_m ::_{\{s_m/\texttt{self}, s_1/u_1, \ldots, s_n/u_n\}} Market\langle u_1, \ldots, u_k \rangle$$
$$\| \prod_{j=1}^{k} s_j ::_{\{s_j/\texttt{self}\}} Shop_j\langle ga_j, pr_1^j, \ldots, pr_n^j \rangle$$

7.4 Ambient implementation

To simplify the presentation, we will make also use of the *objective moves* together with the special ambients introduced in Section 6.

Agent *Buyer* is an ambient called b with inside three concurrent ambients. The first one, called req_list, moves out of b and into m (the ambient of *Market*) and represents the request for the list of camera shops. This ambient contains the *Buyer*'s geographical area gab and the capabilities needed for the *Market* can send the answer to *Collector*. The second one, called res, contains process *Collector* and its initialization values (price and shop location). The last one, called *best*, contains the part of process *Buyer* that is waiting for the result of the search.

$$Buyer \stackrel{\triangle}{=} b[\ (\nu \ req, res, bprice, bwhere)$$
$$(req\_list[\langle req \rangle \ | \ out \ b.in \ m.(req^{|\uparrow}[\langle gab \rangle \ | \ req^{|\uparrow}[\langle in \ res \rangle])])$$
$$| \ res[bprice[\langle max\_price \rangle] \ | \ bwhere[\langle nowhere \rangle] \ | \ Collector]$$
$$| \ best[open \ res.open \ bprice.(price).open \ bwhere.(where)....] \ )$$
$$]$$

Process *Collector*, first, moves its enclosing ambient res out of b, then, iterates the following behaviour: waits for a list of shops to be visited, opens (public) ambient $pret$ enclosing the list, reads the list and checks if it is empty, and

—if the list is empty, moves ambient res (that contains the result of the search: a price and a shop location, each enclosed into an enveloping ambient) into ambient *best* and terminates;

—while the list is not empty, reads the actual best price and relative location, moves ambient res into the first shop in the list, reads the price for the wanted camera offered by the shop, compares this price with the actual best price, updates the current information (depending on the result of the comparison) and iterates the same behaviour.

For reading the price of the wanted camera offered by a shop, a protocol is followed that relies on (public) ambient pk and is similar to (but simpler than) that described in Section 6 for the case of a process that wants to enter a "secure" ambient. Ambients $read$ e $done$ are needed to avoid that the price of the camera can by mistake be read in place of a list of shops: in practice, $read$ is used for safely disclosing ambient $camera$, and $done$ for safely disclosing $read$ after the price has been assigned to p.

$Collector \triangleq$
  $out\ b.open\ pret.$
  $!(x).\ \textbf{if}\ x = empty\ \textbf{then}\ in\ b.in\ best$
        $\textbf{else}\ open\ bprice.(price).open\ bwhere.(where).in\ hd(x).$
        $(\nu\ read, done)\ ($
            $pk[out\ res.in\ camera.in\ res.in\ read]$
            $\mid read[\ open\ camera$
                   $\mid (p).(\ done[out\ read]$
                       $\mid out\ hd(x).\textbf{if}\ p < price$
                               $\textbf{then}\ \langle tl(x)\rangle \mid$
                                    $bprice[\langle p\rangle] \mid bwhere[\langle hd(x)\rangle]$
                               $\textbf{else}\ \langle tl(x)\rangle \mid$
                                    $bprice[\langle price\rangle] \mid bwhere[\langle where\rangle]$
                    $)\ ]$
            $\mid open\ done.open\ read\ )$

Each shop provides two different kinds of information. The first kind of information, enclosed within ambient $ansga$, is provided whenever ambient ga_shop can locally be opened, i.e. whenever a request for the shop geographical area is found. The second kind of information is the shop's price list and is dispatched by using (public) ambient pk (as explained before).

$$Shop_j \triangleq s_j[\ !open\ ga\_shop.ansga[out\ s_j.in\ m.\langle ga_j\rangle] \mid \prod_{i=1}^{n} !camera_i[open\ pk \mid \langle p_j^i\rangle]\ ]$$

When a request for a list of shops arrives, i.e. ambient req_list can be opened, $Market$ activates a copy of itself for serving the request. The server process behaves as follows. First, it reads the geographical area of $Buyer$ and the capability res needed to send the result of the search back. Then, in order to compute the wanted list of shops, it creates private ambients sl, c and end. Ambients sl and c are used similarly to the corresponding locations in the Dπ implementation. Ambient end is used as a synchronization mechanism: when it can be opened, the computation of the list of shops has finished and the result, enclosed into ambient $pret$, is sent back to $Buyer$. Requests to shops for their geographical area are sent by using ambient ga_shop; shops answers are dispatched by using ambient $ansga$.

$Market \triangleq$
  $m[\ !open\ req\_list.(rq).mv\ in\ rq.(gab).mv\ out\ rq.open\ rq.$

$$mv\ in\ rq.(res).mv\ out\ rq.open\ rq.(\nu\ sl, c, end)($$
$$sl^{\downarrow\uparrow}[\langle empty\rangle] \mid c^{\downarrow}[\langle 0\rangle]$$
$$\mid \prod_{j=1}^{k} (\ ga\_shop[out\ m.in\ s_j]$$
$$\mid\ open\ ansa.(ga_j).$$
$$\mathbf{if}\ gab = ga_j$$
$$\mathbf{then}\ mv\ in\ sl.(x).(\ \langle x\#s_j\rangle$$
$$\mid\ mv\ out\ sl.mv\ in\ c.(y).$$
$$\mathbf{if}\ y = (k-1)$$
$$\mathbf{then}\ mv\ out\ c.end[]$$
$$\mathbf{else}\ \langle y+1\rangle\ )$$
$$\mathbf{else}\ mv\ in\ c.(y).\mathbf{if}\ y = (k-1)\ \mathbf{then}\ mv\ out\ c.end[]\ \mathbf{else}\ \langle y+1\rangle$$
$$\mid\ open\ end.open\ c.mv\ in\ sl.(w).mv\ out\ sl.open\ sl.pret[out\ m.res.\langle w\rangle])]$$

Like for the $D\pi$ implementation, the overall system is obtained by composing *Buyer*, *Market* and all *Shop$_j$*, for $j \in \{1, \ldots, k\}$, in such a way that ambients s_j are not visible to *Buyer*.

$$System \stackrel{\triangle}{=} Buyer \mid (\nu\ s_1, \ldots, s_k)(Market \mid \prod_{j=1}^{k} Shop_j)$$

7.5   An evaluation of the implementations

In this section, we compare the different implementations according to few parameters. We will take into account the expressiveness of the calculi with respect to *resource handling*, the accordance with the *software architecture* of Figure 2 and with the *thin client + application server* paradigm that is usually followed in the design of distributed applications for wide-area networks, and the *ease of programming* offered by the underlying computational model.

Before describing the evaluation, let us spend a few words on the thin client + application server paradigm (TC+AS, for short). Such a scheme consists of a (typically) remote service invoked by a client. From a logical point of view, the application server consists of two components. The first one handles the interactions with clients: it is an interface that receives their requests. Here, we abstract away from the query mechanisms that, in actual implementations, could rely on query languages (for instance, `Java` [Arnold and Gosling 1997] provides the module JDBC for handling database queries). The second server component handles the requests and, possibly, returns the results to clients. Clearly, the complexity of the application heavily relies on the second component. In our example, *Buyer* behaves as a client both with respect to *Market* (the former asks to the latter for a list of shops) and with respect to the shops (*Buyer* asks to each *Shop* in the list for the price of the wanted camera).

Resource Handling. In the first three implementations, each shop is a process located at a given location (shops locations are all different) and a shop price list is implemented as n parallel processes (one for each camera model).

In $D\pi$, the price of a camera model can be directly accessed at a channel with the same name of the camera. In `Djoin`, the price list of shop $shop_j$ is implemented

as a process, enabled by the firing of a reaction rule prompted at channel $shop_j$, which consists of n parallel processes, one for each camera model, each guarded by a condition on the model of camera. In KLAIM, the price list of a shop is modelled as a multiset of tuples of the form $\mathbf{out}(''\text{camera}_i'', pr_i^j)$ at the tuple space of the node where the shop is located (the implementation relies on the *non-destructive* KLAIM primitive **read** that allows processes to access tuples without consuming them).

In `Ambient`, each shop is an ambient: price list and information about its geographical area are implemented as concurrent sub-ambients. The price of a camera model is accessed by means of ambient migration and of a protocol that also requires the participation of the shop.

Software Architecture. All the implementations match the logical architecture in Figure 2. Indeed, *Buyer* just knows the address of *Market*, while *Market* also knows the shops addresses. Notice that in the `Ambient` implementation, ambients are restricted, not channels as in the other implementations.

TC+AS. All the first three implementations match the paradigm: each client simply makes its requests to the server (both *Market* and $Shop_j$) by means of an output action (as we said before, we abstract from the query mechanisms).

In `Djoin`, each client just points out its requests at the appropriate channels: it is the run-time support underlying the language that takes care of dispatching messages to the corresponding processes (i.e. those processes that define the channels where messages are transmitted). Hence, client-server interactions are simpler than those in $D\pi$ and in KLAIM, where the client must also know the location of the server.

In the `Ambient` implementation, the mechanism that clients must use to call for services is based on a quite complex previously established protocol that also requires the server to take part actively in the interaction. The client must know the address of the server and send an appropriate ambient containing the parameters for the invocation (such as, e.g., a *return* capability). The server must handle the ambient sent by the client.

Easy of Programming. In $D\pi$, the possibility of creating restricted allocated channels and of extruding their scope makes it easy to program point-to-point private communications. However, the lack of recursion in the calculus, makes the definition of mobile agents not at all satisfactory. Indeed, let us consider agent *Collector*. In order to be able to iterate its behaviour while moving among locations, the agent must be defined by using the replication operator. Now, when the agent's general definition is instantiated, the new instance is put in parallel with the agent's definition. Therefore, whenever the instance migrates, differently from `Djoin` and `Ambient`, the agent's definition will remain at its original location *collh*. Hence, each time the agent behaviour must be iterated to visit a new location, a message has to be sent to the original location *collh* for creating a new instance of *Collector*. ¿From a logical point of view, this corresponds to a kind of *zigzag* movement between the original location *collh* and those of the shops to visit.

`Djoin` resembles a functional programming language: reaction rules can be seen as function definitions, and messages as function invocations. Mobility (of mes-

sages) is implicit in the `join-calculus` primitives and the `Djoin` primitive **go** for process mobility does not enrich the expressiveness of the calculus. In the marketplace implementation, this primitive is used to move agent *Collector* to a sub-locality of a shop: this is not at all necessary because the interaction between the agent and the shop could also take place while leaving the agent at its original location. The point is that, from the one hand, **go** cannot be used for moving an agent to the *same* locality of another one (two agents located at two different localities can never reach the same locality) and, from the other hand, co-location is not necessary for communication to take place. We remark that **go** is useful particularly to move location trees in case of failures [Fournet et al. 1996].

Klaim fully exploits process mobility. Moreover, the possibility of associating names to processes and of invoking them by using the corresponding name simplify program presentation (consider, for instance, processes used as arguments of operations).

`Ambient` linguistic mechanisms, from a programming point of view, are too low-level. The programmer is also in charge of the ambient "routing" and this must be done step by step.

8. SECURITY MECHANISMS

An effective language for network-aware programming, and the underlying computational model, should face with *security issues* since early design stages. Here, with security we mean the ability of protecting resources and data as regards as secrecy and integrity. The language and its model should provide mechanisms for supporting the specification and the enforcement of security policies. The ideal situation is that of designing the language together with its secure kernel and to implement the corresponding secure abstract machine. In other words, security should be taken into account in the design stages and not considered later on top of existing infrastructures. Formal semantics plays a crucial rule to prove correctness of security infrastructures. Furthermore, "secure" programming languages should be equipped with both a formal calculus and compositional logics in order to be able to reason about security properties. Interpreters and development environments for those languages should be equipped with semantic-based verification tools that programmers may use to prove properties of mobile code. Partial examples of this principle are provided by the *Java Bytecode Verifier* and the *Proof Carrying Code* approach. In the first approach [Yellin 1995; Stata and Abadi 1999], Java applets from non local sites are checked by the bytecode verifier before loading. In the second approach [Necula 1997; Necula and Lee 1998], a mobile code is equipped with a proof that the code satisfies certain security constraints.

In this section we focus on the security mechanisms provided by the calculi and outline their security models. Dπ, Klaim and `Ambient` rely on *access control policies* while `Djoin` relies on *cryptography*.

8.1 Access Control Policies

Access control policies regulate users' access to resources on the basis of authorization rules which express the type of usages each user is allowed of certain resources. To specify and enforce access control policies, Dπ, Klaim and `Ambient` exploit *capability-based* type systems.

Capability-based type systems use *access types* to provide information about access rights for resource usages. In general, access types have a hierarchical structure. The hierarchy of access rights is reflected in the subtype relation (denoted by $\sqsubseteq$). The underlying idea is that if a process P has access type ac1 and ac1 is a subtype of ac2 then P could be considered as having access type ac2 as well. In other words, P can be safely used in all the cases where a process of type ac2 is expected. The general principle of capability-based type systems is that

> *mobile agents are (type) checked before being executed to ensure that they do not violate the access policies of the current execution site.*

In the rest of this section we outline the mechanisms which are adopted to specify and enforce access control violations by relying on typing information. Rather than presenting in detail each type system, we focus on an example of Dπ programming (mainly borrowed from [Riely and Hennessy 1999a]). The same example could be expressed in KLAIM and, with some more efforts, in Ambient too.

Assume that *accnt* is a location representing a bank account and has methods to deposit/fetch money and to close the account. The access control policy of the *accnt* location can be specified by the type

$$accnt : \mathtt{loc}\{deposit : \mathtt{rw}\langle int\rangle, withdraw : \mathtt{rw}\langle int\rangle, close : \mathtt{rw}\langle\rangle\}.$$

This type is an abstraction of the *accnt* behaviour and tells us that *accnt* is a location providing three channels (methods): *deposit*, *withdraw* and *close*. All these behaviours have read/write (as flagged by the r and w respectively) capabilities. The first two methods have an integer parameter, while the last has no parameter.

Access rights restrictions to the *accnt* location are expressed by exploiting the subtype relation. For instance, an agent whose type is

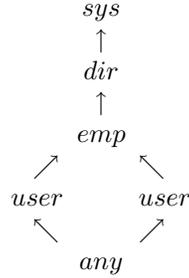$$accnt : \mathtt{loc}\{deposit : \mathtt{w}\langle int\rangle\}$$

has only the write capability w on the method *deposit*, namely the agent can only deposit money. Similarly, the type

$$accnt : \mathtt{loc}\{deposit : \mathtt{w}\langle int\rangle, withdraw : \mathtt{w}\langle int\rangle\}$$

describes the access rights of a process which is allowed to deposit and withdraw money.

The types above permit capturing and monitoring immediate accesses to locations/channels without considering information flow among *roles*. Here with roles we mean classes of users with different rights; for instance, the bank director has more rights than the employer. Since late 70's, several studies addressed the problem of developing theories of information flow [Denning 1976; 1977]. In a capability-based type system, information flow is naturally modelled by a lattice of security levels mirroring the levels of security policies among the roles.

In our running example, the lattice of security levels could have the following form:

$$
\begin{array}{c}
sys \\
\uparrow \\
dir \\
\uparrow \\
emp \\
\nearrow \qquad \nwarrow \\
user \qquad\qquad user \\
\nwarrow \qquad \nearrow \\
any
\end{array}
$$

Hence, *sys* (the role of the system software administrator) is the highest security level, *dir* represents the security level of the bank director, *emp* is the employer level, *user* is the security level of clients. Finally, *any*, the lower security level, is available to every role in the system. Intuitively, if an agent has security level σ then it can only use capabilities whose security level is at most σ. The introduction of types and security levels in the language is reflected by the operational rules. For instance, the operational rule for code mobility among locations becomes:

$$
\ell[k_\rho :: p]_\sigma \longrightarrow k[p]_\rho \qquad\qquad \text{if } \rho \sqsubseteq \sigma.
$$

Now, the question we need to answer is:

> In which sense do capability-based type systems do ensure that all accesses are in accordance with the underlying policy?

This corresponds to saying which are the mechanisms that enforce (i.e. implement) the access policy specified by typing information. This task is performed by the *type checker* that verifies that both statically and dynamically loaded software modules match the access policies of the locations. All agents that have successfully passed the type checking phase can be downloaded, installed and executed.

The formal treatment starts by specifying security violations as run-time errors of the form

$$
P \xrightarrow{\ \Gamma\ } err\ k.
$$

Such a predicate states that in a system P a thread during its evolution has violated the security policy Γ and this violation occurs at location k. For instance, we have

$$
\ell[k_\rho :: p]_\sigma \text{ produce an error at } \ell \text{ if } \rho \not\sqsubseteq \sigma;
$$

namely, a thread cannot use migration in order to expand its security level. The next step is to prove that the type system ensures the absence of run-time errors. To this purpose the following two properties must hold.

Subject Reduction: Well-typing is preserved by systems evolution.

Type Safety: Well-typed systems cannot generate immediate run-time errors, i.e. they cannot produce violations of the security policy.

The capability-based type systems of Dπ, KLAIM and Ambient detect security violations following this strategy. An innovative aspect of the Dπ capability-based

Table XVIII.   `Sjoin` syntax

Values	v, v'	::=	...	
		|	$\{\tilde{v}\}_v$	*Encryption*
Processes	P, Q	::=	...	
		|	**decrypt** v **using** v' **to** $\tilde{x}$ **in** P **else** Q	*Decryption*
Definitions	D, E	::=	...	
		|	**fresh** x	*Fresh name*
		|	**keys** x^+, x^-	*Fresh pair of keys*

type system is the handling of *open* networks [Hennessy and Riely 1999]. In an open network a subset of hosts (localities) may be untrusted. In this case, the structure of the type system is enriched with

—partial types that label some locations as untrusted;

—run-time type checking to enforce security restrictions for processes coming from untrusted locations.

Recently, Hennessy and Yoshida [Yoshida and Hennessy 1999] have proposed an interesting approach to the design of type systems for security. The basic idea of their proposal is to assign to the migrating process a *dependent type*: i.e. a type expression that may be instantiated to particular types depending on the values of certain parameters. In other words, types of mobile processes depend on the site where they are running (a similar idea has been used in [De Nicola et al. 2000] but in a less general framework). In this way types of processes are essentially an *interface* whose purpose is to limit the resources that processes can access. When an agent migrates, the local resources are bound to its interface and, automatically, the access control is obtained.

8.2 Cryptography

In a wide-area network all of the communication channels are not necessarily secure. *Private channels* are useful abstractions for secure channels and are used by all the calculi we have presented. However, actual implementations usually rely on *public channels* and enforce security properties by means of encrypted communications. Shared-key or public-key encryption are adopted to guarantee secrecy and integrity of messages [Abadi 1999]. This idea is the basis of the security mechanisms provided by the `join-calculus`. The innovative contributions of the `join-calculus` approach consist of the design of an (intermediate) virtual machine that provides cryptographic mechanisms and of the mapping from `join-calculus` (with private channels) to `Sjoin` (*Secure Join Calculus*, [Abadi et al. 1998]), the language of the virtual machine. The correctness of this framework is ensured by a *sound*, fully-abstract translation of the high-level constructs in the low-level cryptographic primitives (we refer to [Abadi 1999] for a more detailed presentation of the idea of using full abstraction as a tool for implementing secure systems). In [Fournet et al. 1996] it is shown that `Djoin` can be naturally encoded in the `join-calculus`. Hence, it is possible to encode `Djoin` into `Sjoin` by relying on the mapping from the `join-calculus` to `Sjoin` which we describe in the rest of this section.

`Sjoin` enriches the `join-calculus` with cryptographic primitives in the spirit of Abadi and Gordon's spi-calculus [Abadi and Gordon 1999]. Table XVIII illustrates

the extensions of `join-calculus` with cryptographic primitives. The *encryption* construct builds a new value by encrypting the tuple of values $\tilde{v}$ under the key v. The *decryption* construct tries to decrypt message v with a known key v'. If the decryption succeeds then P runs with $\tilde{x}$ replaced by the results of the decryption, otherwise Q will be executed. Definition **fresh** x introduces the fresh name x. Definition **keys** x^+, x^- introduces the encryption key x^+ and its inverse decryption key x^-. In general, x^+ differs from x^- (this is the usual hypothesis in public-key cryptosystems). However, x^+ and x^- can coincide (as it happens in shared-key cryptosystems); in this case, the construct is written **key** x. In the definition of the language, there are a few implicit assumptions on the underlying cryptographic system. We make them explicit below:

(1) A tuple of values $\tilde{v}$ encrypted under a key v', written $\{\tilde{v}\}_{v'}$, can only be decrypted using v'. The only way to produce the ciphertext $\{\tilde{v}\}_{v'}$ is to encrypt $\tilde{v}$ under v'. If v' is secret, the execution environment cannot guess or forge $\tilde{v}$ (*perfect encryption*).

(2) There is enough redundancy in the ciphertexts to tell whether decryption of a value with a given key has actually succeeded or not.

(3) The only way to form a new key is to get a fresh pair of complementary keys.

All communications go through a network interface which consists of two public channels *emit* and *recv*. Processes send messages to the network at channel *emit* and receive messages from the network at a continuation channel previously sent at channel *recv*. The expression **def** $emit\langle m \rangle \mid recv\langle \chi \rangle \, \triangleright \, \chi\langle m \rangle$ **in** P describes the behaviour of the nework interface, where P is the parallel composition of all processes in the network (plus a process that generates *noise* and prevents traffic analysis, and a process that makes channels *emit* and *recv* public).

In this model, processes in the network need to filter the incoming network messages. This filtering can be programmed as:

let $\tilde{y} = $ **filter** k **in** $P \quad \triangleq$
    def $\chi\langle m \rangle \, \triangleright \,$ **decrypt** m **using** k **to** $\tilde{y}$ **in** P **else** $emit\langle m \rangle \mid recv\langle \chi \rangle$
    in $recv\langle \chi \rangle$

This process receives a message m from the network at (the continuation) channel χ and attempts to decrypt m with a key k. In case of success, P is executed with $\tilde{y}$ replaced by the result of decryption, otherwise, m is sent back to the network and a new attempt is done. Notice that this polling strategy is very expensive.

Now, we outline the translation function from `join-calculus` to `Sjoin`. We assume that, for all the `join-calculus` names x, the mappings $x \mapsto x^-$, $x \mapsto x^+$ and $x \mapsto \omega_x$ are injective maps. Moreover, we use **repl** P as a shorthand for process **def** $x\langle\rangle \, \triangleright \, (P \mid x\langle\rangle)$ **in** $x\langle\rangle$.

The translation mapping relies on two special `Sjoin` processes defined as follows:

$$E_x[\tilde{v}] \quad \triangleq \quad \textbf{def key } k \textbf{ in}$$
$$(\textbf{repl } emit\langle \{k\}_{x^+}\rangle) \mid \textbf{let } d' = \textbf{filter } k \textbf{ in } (\textbf{repl } emit\langle\{\tilde{v}\}_{d'}\rangle)$$

$$R_x \quad \triangleq \quad \textbf{repl } \Big(\textbf{ let } k = \textbf{filter } x^- \textbf{ in def key } k' \textbf{ in}$$
$$(\textbf{repl } emit\langle\{k'\}_k\rangle) \mid \textbf{let } \tilde{y} = \textbf{filter } k' \textbf{ in } \omega_x\langle\tilde{y}\rangle \Big).$$

Table XIX. The translation $[\![\cdot]\!]$

On processes		
$[\![x\langle \tilde{v}\rangle]\!]$	$=$	$E_x[\widetilde{v^+}]$
$[\![\mathbf{def}\ D\ \mathbf{in}\ P]\!]$	$=$	$\mathbf{def} \bigwedge_{x\in\mathbf{dv}[D]} \mathbf{keys}\ x^+, x^-\ \mathbf{in}\ (\mathbf{def}\ [\![D]\!]\ \mathbf{in}\ \prod_{x\in\mathbf{dv}[D]} R_x)\mid [\![P]\!]$
On patterns		
$[\![x\langle \tilde{y}\rangle]\!]$	$=$	$\omega_x\langle\widetilde{y^+}\rangle$

Processes $E_x[\tilde{v}]$ and R_x are used by the sender and by the receiver to establish a session-key k' (for this, a shared key k is used that is exchanged encrypted under x^+), which is then used to encrypt the values $\tilde{v}$. Whenever the decryption succeeds, the decrypted message is sent at the internal channel ω_x that resides on the same machine where R_x is executed.

Table XIX illustrates the non-trivial cases of the mapping from `join-calculus` to `Sjoin`. Notice that $[\![\ ]\!]$ automatically associates two keys (x^+ and x^-) to each defined (i.e. private) channel x and that communication at x is rendered as encrypted communication at public channels *emit* and *recv* plus local communication at the internal channel ω_x.

As a simple example of this mapping, let us consider the `join-calculus` process

$$P \triangleq \mathbf{def}\ x\langle y_1, y_2\rangle \triangleright z\langle y_1\rangle\ \mathbf{in}\ x\langle u, v\rangle$$

Process P defines a channel x, that can be thought of as a secure channel, by attaching to it the definition $x\langle y_1, y_2\rangle \triangleright z\langle y_1\rangle$. Thus, whenever a pair y_1, y_2 occurs at x, its first component is forwarded at z. In the body of P the pair u, v is sent at x. By applying the clauses in Table XIX we get the following `Sjoin` process

$$[\![P]\!] = \mathbf{def\ keys}\ x^+, x^-\ \mathbf{in}$$
$$(\mathbf{def}\ \omega_x\langle y_1^+, y_2^+\rangle \triangleright E_z[y_1^+]\ \mathbf{in}\ R_x) \mid E_x[u^+, v^+].$$

$E_x[u^+, v^+]$ sends u^+ and v^+ encrypted under x^+ at channel *emit*. R_x can receive this message through channel *recv*, decrypt it using x^- and forward its content at the internal channel ω_x, that can be safely used because $\omega_x\langle y_1^+, y_2^+\rangle \triangleright E_z[y_1^+]$ and R_x are located on the same machine. Hence, although the body $x\langle u, v\rangle$ and the definition $x\langle y_1, y_2\rangle \triangleright z\langle y_1\rangle$ are located on different machines, communication at x is secure.

9. EVALUATION

In this section we evaluate and compare the foundational calculi we consider in this paper along the following axes:

— *Communication.* The communication model takes into account various aspects: local or remote communications, interaction media (channels, tuples, ether...), explicit naming versus anonimycity of the partner, synchronous or asynchronous interactions.

— *Mobility.* As mentioned in Section 1, several forms of mobility are possible. The calculi provide different mobility abstractions which allows (combination of) code, process, or agent mobility.

Table XX.   Calculi for Network-Aware Programming: An Assessment

	Dπ	Djoin	KLAIM	Ambient
Communication	local channel-based explicit naming synchronous	local channel-based explicit naming asynchronous	local/remote tuple-based anonymous associative asynchronous	local message-based anonymous asynchronous
Mobility	process/agent	process/code	code/process/ agent	agent
First-class objects	channels/locations	channels/locations	processes/sites	capabilities and ambient names
Administrative Domains	flat model locations	hierarchical model locations tree	flat model nodes and allocation env.	hierarchical model ambients
Coordination Mechanisms	located threads and located channels	local processes and local definitions	allocation env. and nodes	ambients
Security Mechanisms	types static/dynamic type checking	cryptography security architecture SJoin	types static/dynamic type checking	types static type checking
Experimental Implementations	non available	JoCaml Objective-Caml	X-KLAIM KLAVA	JoCaml Ambit

— *First-class objects.* There are some differences with respect to the nature of objects exchanged in communications.

— *Administrative domains.* All the calculi offer different abstractions to fit the notion of administrative domain. Intuitively an administration domain reflects the idea of having a group of threads running under the control of the same authority which monitors the use of resources.

— *Coordination mechanisms.* Coordination is a key concept for modeling and designing network-aware applications. Complex applications are designed and developed in a structured way, starting from the basic computational components and adding suitable software modules called *coordinators* which handle the interactions among the components.

— *Security mechanisms.* Security policies are a relevant aspect to consider in wide area distributed programming. Type systems (Dπ, KLAIM, Ambient) and cryptography primitives (Djoin) are complementary mechanisms that provide security at different level.

We believe that the six criteria listed above are essential to characterize and to guide the design of network programming languages. Of course, the calculi may be compared also considering other important features. For instance, KLAIM reserves particular attention to data (modelled as tuples) and acknowledges the importance of data as an essential component of programming languages for WAN applications. Indeed, KLAIM [De Nicola et al. 1998] supports programming of WAN applications which interact by sharing data. The data and the operations over data available in the KLAIM system are the Linda [Carriero and Gelernter 1989] ones; however, they could be based on XML [Bray et al. 1997] and a query language for XML.

Table XX summarizes our evaluations.

Communications

All the calculi support a local communication model. KLAIM also provides mechanisms for remote communications: processes can interact by asynchronous distributed object method invocations (as it can be seen looking at the `Klava` package [Bettini et al. 1998]). In the other approaches remote communications can be obtained by a combination of mobility and local communication

Communication primitives are based on an asynchronous model (WAN applications are inherently asynchronous systems). The only exception is given by Dπ communication mechanism which is local and synchronous. `Ambient` and KLAIM communication paradigm is anonymous (tuples have no name and `Ambient` messages are communicated through the ether) and, for KLAIM, associative (tuples are content addressable). Anonymous communications have been advocated [Cabri et al. 1998b] to be more suitable than the one based on *naming* (adopted by Dπ and `Djoin`) to design and develop WAN applications.

Communication primitives can be exploited to coordinate and synchronize process activities. Basic synchronization constructs can be represented in each of the calculi. In this respect, the synchronization mechanism of `Djoin` appears to be the more refined: it allows two processes to synchronize on a set of channels (those channels defined in the join pattern). The uniqueness of join receptor (there is a single location where synchronization for a given receptor is dealt with) simplifies the distributed implementation of `Djoin` synchronization. Moreover, the uniqueness of join receptor and the join pattern confer to the `Djoin` programming model a "functional" feature. For instance, in the first `Djoin` implementation of the counter process, a client of the counter calls the (local) "function" *cnt* by triggering the corresponding pattern and passing to *cnt* an initial value and a continuation.

Mobility

In each of the calculi mobility abstractions and migration policies are under programmer's control. Both Dπ and KLAIM permit process and agent mobility; additionally, KLAIM permits node mobility (via its higher order communication mechanism). The `Ambient` mobility primitives are similar in spirit to those of `Djoin` in that they permit movement of nested structures (ambients and trees of locations, respectively) but, while `Djoin` permits moving code and processes, `Ambient` permits moving data and active computations (i.e. agents). Moreover, `Ambient` is the only calculus that also supports mobile computing, in fact ambients can be also viewed as mobile devices.

First-class objects

In each of the calculi, addresses (namely, locations in Dπ and `Djoin`, sites in KLAIM, and ambients in `Ambient`) are first class entities. Furthermore, i) Dπ and `Djoin` allow channel names to be communicated, ii) KLAIM permits higher order communication in that processes may appear in tuples and can be downloaded by **in/read** primitives, and, finally, iii) `Ambient` allows sequences of capability to be exchanged.

Administrative domains

Locations reflect the idea of administrative domains: computations at a certain location are under the control of a specific authority. Despite of the similarities,

there are some differences on the way locations are exploited. In fact, Dπ localities are mainly adopted to program migration and to define allocated channels, i.e. as a way to model the distributed object invocation mechanisms. Furthermore, a process which needs a non-local channel name (i.e. a remote resource method), say a, has to know the location where a lays. In `Djoin`, instead, there is a more structured location concept: a location is a tree composed by the root location and its sub-locations. When a process defined on `a` moves itself to another site, the whole tree rooted at `a` moves with the process. Moreover, in `Djoin` locations are not necessary for determining the allocation site of remote channels. It is the communication infrastructure (uniqueness of join receptor) that worries about determining the service channel site. In this respect, this mechanism is very close to the `CORBA` [Group 1998] philosophy, where the network structure is completely transparent to the applications. In KLAIM, as it happens for Dπ and `Djoin`, whenever a process knows a site s it may access to the services granted at s. However, the role of sites in KLAIM differs from that of locations in Dπ. Locations that only host the null process can safely be removed in Dπ (the structural law $\ell[0] = 0$) while this is not allowed in KLAIM. Indeed, this corresponds to a different treatment of net addresses: in KLAIM sites exist independently of the allocated processes, and, in particular, processes can migrate to or tuples can be placed in a tuple space at a certain site provided that the site already exists.

`Ambient` handles net addresses in the same way KLAIM does; however, differently from the other calculi, in `Ambient` the knowledge of the name of a location is not enough to access its service: it is necessary to know the route to the location.

Coordination mechanisms

`Ambient` capabilities provide a full fledged coordination language to move, compose and rearrange administration domains: the only way to manipulate the structure of ambients is by using capabilities. However, since `Ambient` primitives are asynchronous, a special care is needed to program effective coordination policies which control both migration and the delivery of messages (communication is asynchronous).

The net coordination mechanism of KLAIM relies on allocation environments that map logical addresses (i.e. variables) to physical ones (i.e. sites). This mechanism allows applications to have a logical view of the structure of the net without actually taking care of its (real) physical structure. KLAIM provides a powerful platform to program network services where the set of available services (coded as tuples in the tuple spaces) at any given moment may dynamically change. When a service is not available at a node, it is enough to visit the nodes of the net (by using the mobility primitives) to find a node which provides the wanted service. Once the node is found, the service can be downloaded and executed. Some commercial platforms, such as, e.g., Jini [Edmonds 1999; Arnold et al. 1999], provide similar computational mechanisms to integrate dynamic network systems.

Dπ provides the notion of system as main coordination mechanism (see Table IV). A system is a set of allocated threads that are executed in parallel. Allocated threads use allocated channels that can be viewed as a coordination mechanism: the action of receiving the allocated channel $\ell[a]$ has the effect of making ℓ available and hence all services provided at that location become available.

In `Djoin`, nets (see Table VII) and join patterns are the coordination mechanisms of the language. We remark that the routing mechanism of `Djoin` is not under the control of the programmer: it is the underlying support that provides for the message routing.

Security mechanisms

Dπ, KLAIM and `Ambient` exploit notions of types as security mechanism and, a part for `Ambient` that only adopt static checking, use both static and dynamic checking in order to gain programmability of access control policies. Instead, `Djoin` relies on a lower-level security mechanism based on cryptography: applications are compiled in an intermediate secure layer, `Sjoin`, that encrypts/decrypts the messages exchanged through the network.

The different security models, types for access control and cryptography, are not conflicting. A secure programming language should have both features. Indeed, types may be used to program access control policies and reason about properties of programs; cryptography may ensure authenticity and privacy of messages that in a wide area network may be violated by malicious intruders or network failures.

10. CONCLUSION AND RELATED WORK

In this paper we evaluated some foundational calculi for mobile programming along two main guidelines: *programming language design* and *application design*. Programming language design has been exploited to identify the programming abstractions and paradigms which are directly inspired by these calculi. Application design has been exploited to understand the potentials of the calculi to support wide-area network applications.

A topic that has not been fully considered in this paper is *failures handling*. Since in asynchronous systems such as the Internet no limits on relative processors speed and communication delays exist, it is difficult for the owner to exactly determine whether a mobile agent is lost due to a failure or whether its execution has only been delayed due to slow processors or communication links. Among the calculi for network-aware programming we have considered, only `Djoin` has an explicit construct to cope with failures: **go** [Fournet et al. 1996] permits moving location trees when failures occur.

Related work. A number of distributed extension of the π-calculus have been proposed to address specific problems of network programming. Here we briefly remind some proposals.

—The π_ℓ calculus [Amadio and Prasad 1994] has been introduced to model failures of distributed programs.

—The *located* π_1 [Amadio 1997; 2000] has been introduced as a formal framework to model some basic properties of `Djoin` directly inside the asynchronous π-calculus enriched with a specific type discipline.

—The *Nomadic* π-calculus [Wojciechowski and Sewell 1999; Sewell et al. 1999] has been introduced to model and study properties of communication infrastructures of mobile processes.

In the last few years, several models and programming languages for mobile applications in distributed systems and networks have been proposed that rely on a Linda-like communication paradigm. To deal with such issues as modularity, naming and security, the original Linda communication paradigm [Gelernter 1985; Carriero and Gelernter 1989] has firstly been extended by providing support for multiple tuple spaces [Gelernter 1989]. On the top of this more recent model, a number of new features have been added which aim at, e.g.,

—adding programmability to tuple spaces by associating reaction rules to communication events [Minsky and Leitcher 1995; Omicini and Zambonelli 1998; Cabri et al. 1998b; 1998a],

—dynamically creating private tuple spaces [Scientific Computing Associate 1994],

—allowing processes to transiently share their own tuple spaces [Picco et al. 1999],

—hierarchically structuring tuple spaces [Ciancarini 1991; Carriero et al. 1995; Omicini and Zambonelli 1998; Bettini et al. 2000],

—restricting (the form of the tuples that can be put into) tuple spaces and the pattern-matching mechanism [Van Der Goot et al. 1997] or the operations that processes can perform over tuple spaces [De Nicola et al. 1999; De Nicola et al. 2000].

Only a couple of calculi (a part for `Djoin`) have been proposed in the literature that resemble the `Ambient` calculus. The Seal calculus [Vitek and Castagna 1998] can be roughly described as a π-calculus with hierarchical locations, mobility and resource access control. Rather than a programming model, the calculus can be thought of as a substrate for implementing higher level languages and advanced distributed applications that provides the system programmer with mechanisms for fully controlling localities and for low-level protection. Recently, to cope with the interferences that arise when implementing interaction protocols, a variant of the `Ambient` calculus have been proposed [Levi and Sangiorgi 2000] that makes use of *co-actions*. In such a variant, each `Ambient` action has a complementary action and a reduction can occur only whenever two complementary actions do synchronize.

REFERENCES

ABADI, M. 1999. Protection in programming-language translations. See Vitek and Jensen [1999], 19–34.

ABADI, M., FOURNET, C., AND GONTHIER, G. 1998. Secure implementation of channel abstractions. In *Proceedings of LICS'98*. IEEE Computer Society Press, 105–116.

ABADI, M. AND GORDON, A. D. 1999. A calculus for cryptographic protocols: The spi-calculus. *Information and Computation 148(1)*, 1–70.

ACHARYA, A., RANGANATHAN, M., AND SALTZ, J. 1997. Sumatra: A language for resources-aware mobile programs. In *Mobile Object System: Towards The Programmable Internet*, J. Vitek and C. Tschudin, Eds. Lecture Notes in Computer Science, vol. 1222. Springer, 111–130.

ACM, Ed. 1997. *Proceedings of the 24th ACM SIGPLAN-SIGACT on Principles of programming languages*. ACM Press.

AMADIO, R. 1997. An asynchronous model of locality, failures and process mobility. See Garlan and Le Mètayer [1997], 374–391.

AMADIO, R. 2000. On modelling mobility. *Theoretical Computer Science 240(1)*, 215–254.

AMADIO, R., CASTELLANI, I., AND SANGIORGI, D. 1996. On bisimulations for the asynchronous π–calculus. See Montanari and Sassone [1996], 147–162.

AMADIO, R. AND PRASAD, S. 1994. Localities and failures. In *Proceedings of FST-TCS'94*, S. Thiagarajan, Ed. Lecture Notes in Computer Science, vol. 880. Springer, 205–216.

ARNOLD, K. AND GOSLING, J. 1997. *The Java Programming Language*. Addison Wesley.

ARNOLD, K., WOLLRATH, A., O'SULLIVAN, B., SCHEIFLER, R., AND WALDO, J. 1999. *The Jini specification*. Addison-Wesley, Reading, MA, USA.

BAL, H., BELKHOUCHE, B., AND CARDELLI, L., Eds. 1999. *Workshop on Internet Programming Languages*. Lecture Notes in Computer Science, vol. 1686. Springer.

BERRY, G. AND BOUDOL, G. 1992. The chemical abstract machine. *Theoretical Computer Science 96(1)*, 217–248.

BETTINI, L., DE NICOLA, R., FERRARI, G., AND PUGLIESE, R. 1998. Interactive mobile agents in X-KLAIM. See IEEE [1998], 110–115.

BETTINI, L., LORETI, M., AND PUGLIESE, R. 2000. Structured nets in KLAIM. In *Proceedings of the ACM SAC'2000, Special Track on Coordination Models, Languages and Applications*. ACM Press, 174–180.

BOUDOL, G. 1992. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis. May.

BRAY, T., PAOLI, J., AND SPERBERG-MCQUEEN, C. 1997. Extensible Markup Language (XML). *The World Wide Web Journal 2(4)*, 29–66.

CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. 1998a. How to Coordinate Internet Applications based on Mobile Agents. See IEEE [1998], 104–109.

CABRI, G., LEONARDI, L., AND ZAMBONELLI, F. 1998b. Reactive tuple spaces for mobile agent coordination. In *Proceedings of the 2nd Int. Workshop on Mobile Agents*, K. Rothermel and F. Hohl, Eds. Vol. 1477. Springer, 237–248.

CARDELLI, L., GHELLI, G., AND GORDON, A. D. 1999. Mobility types for mobile ambients. In *Proceedings of ICALP'99*, J. Wiedermann, P. van Emde Boas, and M. Nielsen, Eds. Lecture Notes in Computer Science, vol. 1644. Springer, 10–24.

CARDELLI, L. AND GORDON, A. D. 1999. Types for mobile ambients. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages*. ACM Press, 79–92.

CARDELLI, L. AND GORDON, A. D. 2000. Mobile ambients. *Theoretical Computer Science 240(1)*, 177–213.

CARRIERO, N. AND GELERNTER, D. 1989. Linda in context. *Communications of the ACM 32(4)*, 444–458.

CARRIERO, N., GELERNTER, D., AND ZUCK, L. 1995. Bauhaus linda. See Ciancarini et al. [1995], 66–76.

CIANCARINI, P. 1991. PoliS: a Programming Model for Multiple Tuple Spaces. In *Proceedings 6th ACM/IEEE Int. Workshop on Software Specification and Design (IWSSD)*, C. Ghezzi and G. Roman, Eds. IEEE Computer Society Press, 44–51.

CIANCARINI, P., NIERSTRASZ, O., AND YONEZAWA, A., Eds. 1995. *Object-Based Models and Languages for Concurrent Systems*. Lecture Notes in Computer Science, vol. 924. Springer.

DE NICOLA, R., FERRARI, G., AND PUGLIESE, R. 1998. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering 24(5)*, 315–330.

DE NICOLA, R., FERRARI, G., AND PUGLIESE, R. 1999. Types as specifications of access policies. See Vitek and Jensen [1999], 117–146.

DE NICOLA, R., FERRARI, G., PUGLIESE, R., AND VENNERI, B. 2000. Types for access control. *Theoretical Computer Science 240(1)*, 215–254.

DENNING, D. E. 1976. A lattice model of secure information flow. *Communications of the ACM 19(5)*.

DENNING, D. E. 1977. Certification of programs for secure information flow. *Communications of the ACM 20(5)*.

EDMONDS, K. 1999. *Core Jini*. Prentice Hall.

FOURNET, C. AND GONTHIER, G. 1996. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT on Principles of programming languages*. ACM Press, 372–385.

FOURNET, C., GONTHIER, G., LÉVY, J., MARANGET, L., AND RÉMY, D. 1996. A calculus of mobile processes. See Montanari and Sassone [1996], 406–421.

FOURNET, C., LÉVY, J., AND SCHMITT, A. 1999. A distributed implementation of ambients. Available at `http://join.inria.fr/ambients.html`.

GARLAN, D. AND LE MÈTAYER, D., Eds. 1997. *Proceedings of COORDINATION'97*. Lecture Notes in Computer Science, vol. 1282. Springer.

GELERNTER, D. 1985. Generative communications in Linda. *ACM Transactions on Programming Languages and Systems 7(1)*, 80–112.

GELERNTER, D. 1989. Multiple tuple spaces in linda. In *Proceedings of PARLE'89*. Lecture Notes in Computer Science, vol. 365. Springer, 20–27.

GROUP, O. M. 1998. Corba: Architecture and specification. Available at `http://www.omg.org`.

HENNESSY, M. AND RIELY, J. 1998. Resource access control in systems of mobile agents. In *High-Level Concurrent Languages*, U. Nestmann and B. Pierce, Eds. entcs, vol. 16(3). Elsevier Science Publishers, 3–17. Full version as CogSci Report 2/98, University of Sussex, Brighton.

HENNESSY, M. AND RIELY, J. 1999. Type-safe execution of mobile agents in anonymous networks. See Vitek and Jensen [1999], 95–115.

HONDA, K. AND TOKORO, M. 1991. An object calculus for asynchronous communication. In *Proceedings of ECOOP'91*, P. America, Ed. Lecture Notes in Computer Science, vol. 512. Springer, 133–147.

IEEE, Ed. 1998. *Proceedings of 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE'98)*. IEEE Computer Society Press.

LANGE, D. AND OSHIMA, M. 1998. *Programming and Deploying Mobile Agents with Java Aglets*. Addison-Wesley.

LEVI, F. AND SANGIORGI, D. 2000. Controlling interference in ambients. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of programming languages*. ACM Press, 352–364.

LEVY, J. 1997. Some results in the join-calculus. In *Proceedings of TACS'97*, M. Abadi and T. Ito, Eds. Lecture Notes in Computer Science, vol. 1281. Springer, 233–249.

MILNER, R. 1993. The polyadic π-calculus: A tutorial. In *Logic and Algebra of Specification*, F. Hamer, W. Brauer, and H. Schwichtenberg, Eds. Springer.

MILNER, R., PARROW, J., AND WALKER, D. 1992. A calculus of mobile processes. *Information and Computation 100(1)*, 1–77.

MINSKY, N. AND LEITCHER, J. 1995. Law-governed linda as a coordination model. See Ciancarini et al. [1995], 125–146.

MONTANARI, U. AND SASSONE, V., Eds. 1996. *Proceedings of CONCUR'96*. Lecture Notes in Computer Science, vol. 1119. Springer.

NECULA, G. AND LEE, P. 1998. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, 333–344.

NECULA, G. C. 1997. Proof-carrying code. See ACM [1997].

OMICINI, A. AND ZAMBONELLI, F. 1998. TuCSoN: a coordination model for mobile information agents. In *Proceedings of the 1st International Workshop on Innovative Internet Information Sytems (IIIS'98)*. 177–187.

PALAMIDESSI, C. 1997. Comparing the expressive power of the synchronous and the asynchronous π-calculus. See ACM [1997], 256–265.

PICCO, G. P., MURPHY, A. L., AND ROMAN, G. 1999. LIME: Linda meets mobility. In *Proceedings of the 21st International Conference on Software Engineering*. ACM Press, 368–377.

RIELY, J. AND HENNESSY, M. 1999a. Secure resource access for mobile agents. Available at: `http://www.csc.ncsu.edu/faculty/riely/papers.html`.

RIELY, J. AND HENNESSY, M. 1999b. Trust and Partial Typing in Open Systems of Mobile Agents. In *Proceedings of the 26th ACM SIGPLAN-SIGACT on Principles of programming languages*. ACM Press, 93–104.

SANGIORGI, D. 1992. Expressing mobility in process algebras: First-order and higher-order paradigms. Ph.D. thesis, Department of Computer Science, University of Edinburgh.

SCIENTIFIC COMPUTING ASSOCIATE, NEW HAVEN, C. 1994. Paradise: User's guide and reference manual.

SEWELL, P., WOJCIECHOWSKI, P., AND PIERCE, B. 1999. Location independence for mobile agents. See Bal et al. [1999]. Full version with title *Location-Independent Communication for Mobile Agents: a Two-Level Architecture* appeared as Technical Report 462, Computer Laboratory, University of Cambridge, April 1999.

STATA, R. AND ABADI, M. 1999. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems 21(1)*, 90–137.

VAN DER GOOT, R., SCHAEFFER, J., AND WILSON, G. V. 1997. Safer tuple spaces. See Garlan and Le Mètayer [1997], 289–301.

VITEK, J. AND CASTAGNA, G. 1998. Towards a calculus of secure mobile computations. See Bal et al. [1999].

VITEK, J. AND JENSEN, C. D., Eds. 1999. *Secure Internet programming: security issues for mobile and distributed objects*. Number 1603 in Lecture Notes in Computer Science. Springer.

WHITE, J. 1996. Mobile agents. In *Software Agents*, J. Bradshaw, Ed. AAAI Press and MIT Press, 437–471.

WOJCIECHOWSKI, P. AND SEWELL, P. 1999. Nomadic pict: Language and infrastructure design for mobile agents. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*. 2–12.

YELLIN, F. 1995. Low level security in java. In *Proceedings of the 4th International Conference on the World-Wide Web*.

YOSHIDA, N. AND HENNESSY, M. 1999. Assigning types to processes. CogSci Report 99.02, School of Cognitive and Computing Sciences, University of Sussex, UK.