KLAIM: A Kernel Language for Agents Interaction and Mobility

Rocco De Nicola, Gian Luigi Ferrari, and Rosario Pugliese

Abstract—We investigate the issue of designing a kernel programming language for mobile computing and describe KLAIM, a language that supports a programming paradigm where processes, like data, can be moved from one computing environment to another. The language consists of a core Linda with multiple tuple spaces and of a set of operators for building processes. KLAIM naturally supports programming with explicit localities. Localities are first-class data (they can be manipulated like any other data), but the language provides coordination mechanisms to control the interaction protocols among located processes. The formal operational semantics is useful for discussing the design of the language and provides guidelines for implementations. KLAIM is equipped with a type system that statically checks access rights violations of mobile agents. Types are used to describe the intentions (read, write, execute, etc.) of processes in relation to the various localities. The type system is used to determine the operations that processes want to perform at each locality, and to check whether they comply with the declared intentions and whether they have the necessary rights to perform the intended operations at the specific localities. Via a series of examples, we show that many mobile code programming paradigms can be naturally implemented in our kernel language. We also present a prototype implementaton of KLAIM in Java.

Index Terms—Programming languages, mobile code languages, semantics of programming languages, language design, coordination models.

1 Introduction

NETWORKING has changed computers from isolated data processors into powerful communication and elaboration devices. The terms *global computers* and *global information structures* have recently been used to identify architectures of this kind and applications over them [8]. The World Wide Web (WWW) is the best known example of an application geographically distributed over a collection of processors and networks. Global structures/computers are rapidly evolving towards programmability; again, an illustrative example is the WWW. One could easily imagine applications with programs running at different sites and needing continuous interactions or applications that have to take decisions according to information retrieved from the global environment.

This new scenario has called for *new* programming languages and paradigms that support migratory (mobile) applications. For example, Java [3] permits local executions of self-contained programs downloaded from other sites. Similarly, Facile [23] supports mobility of programs by allowing processes to be transmitted in communications. Obliq [7] is a programming language with a static scoping discipline where mobile processes maintain their connections when they move from one site to the other. Other examples of languages supporting forms of mobility are CML [38] and Telescript [41].

From a theoretical perspective, much research has addressed mobility starting from the definition of π–calculus [32], which has been used as the basis for designing the concurrent, object oriented, programming language PICT [33]. Indeed, an abstract semantic framework that would allow one to formalize and understand global programming languages is clearly required. Such a semantic framework may be the formal basis to discuss controversial design/implementation issues (e.g., the scoping discipline of mobile processes) and provide support for mechanical reasoning about global programs.

A key issue when designing a language for network programming is security, e.g., *privacy* and *integrity* of data. It is important to prevent malicious agents from accessing private information or modifying private data. Tools are thus needed that enable sites receiving mobile agents for execution to set demands and limitations to ensure that the agents will not violate privacy or jeopardize the integrity of the information. Similarly, mobile agents need tools to ensure that their execution at other sites will not disrupt them or compromise their security. Languages for mobile agents often rely on policies (both at compilation and run-time) that over-restrict privileges and capabilities of mobile agents (e.g., Java [3]). This unnecessarily reduces the expressive power and capabilities of the agents. Moreover, there is no guarantee that certain desired security properties are enforced by the language implementation.

This paper presents a kernel programming language, Kernel Language for Agents Interaction and Mobility (KLAIM), for describing mobile agents and their interaction strategies. We introduce basic concepts and linguistic primitives together with a formal operational semantics. This is followed by a discussion of the pragmatics of the language and of a prototype implementation.

- *R. De Nicola and R. Pugliese are with the Dipartimento di Sistemi e Informatica, Università di Firenze, Via Lombroso 6/17, I-50134 Firenze, Italy. E-mail: {denicola, pugliese}@dsi2.dsi.unifi.it.*
- *G.L. Ferrari is with the Dipartimento di Informatica, Università di Pisa, Corso Italia 40, I-56100, Pisa, Italy. E-mail: giangi@di.unipi.it.*

Manuscript received 16 July 1997; revised 16 Dec. 1997.
Recommended for acceptance by G.-C. Roman and C. Ghezzi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106409.

The distinguishing features of our approach are the explicit use of localities for accessing data or computational resources and the presence of a simple type system to control access rights.

The choice of KLAIM's primitives was heavily influenced by process algebras [25], [30] and Linda [20], [10]. Indeed, our language can be seen as an asynchronous higher-order process calculus whose basic actions are the original Linda primitives enriched with explicit information about the location of the nodes where processes and tuples are allocated.

Explicit localities enable the programmer to distribute and retrieve data and processes to and from the sites of a net and to structure the tuple space as multiple, located spaces. Moreover, localities, considered as first-order data, can be dynamically created and communicated over the network. The overall outcome is a powerful programming formalism that, for example, can easily be used to model encapsulation. In fact, an encapsulated module can be implemented as a tuple space at a private locality, and this ensures controlled access to data.

The separation of the logical distribution of processes and their physical mappings over the net leads to the sharing of the control between programmers and a net coordinator. The actual coordination language is designed to handle all issues related to the physical distribution of processes. Coordinators have complete control over changes of configuration of the network that may be due to addition/deletion of software components and sites, or to transmission of program and of resources.

The actual structuring, in terms of processes and coordinators, provides a clean abstraction device for global programming languages and is instrumental in studying migratory applications and in understanding the extent of configuration decisions before carrying out the actual implementation. This will be illustrated by analyzing the effects of choosing specific scoping disciplines for accessing tuple spaces.

To take security issues into account, we extend KLAIM processes and coordinators with a simple type system that can be used to statically enforce security properties. More precisely, the type system permits one to check whether the operations KLAIM processes intend to perform over the sites of a net really do comply with their access rights.

We illustrate the pragmatics of the language by means of a number of programming examples which demonstrate how well established programming paradigms for mobile applications can be naturally programmed in KLAIM. The untyped version of KLAIM has been implemented as a set of Java packages.

The rest of the paper is organized as follows. Sections 2 and 3 introduce the syntax and the operational semantics of KLAIM, respectively. In Section 4 we present a type system for inferring process types and a methodology for controlling their access rights. This is followed by a discussion of the language pragmatics in Section 5, and by the description of the prototype implementation in Section 6. In Section 7, future research is discussed. Comments about the relationships of KLAIM with other languages and about alternative design choices are scattered along the paper as Remarks.

Preliminary presentations of the KLAIM language can be found in [15], [16].

2 KLAIM: Syntax and Informal Semantics

KLAIM consists of a core Linda with multiple tuple spaces and a set of operators, borrowed from Milner's CCS [30], for building processes. The distinguishing feature is that tuples and operations over them are located at specific sites of a net. We start this section by summarizing the main features of Linda (the interested reader is referred to, e.g., [22], [11], [10] for more details). Then, we present the syntax of KLAIM. The process algebraic operators will be briefly presented in the subsection that contains the syntax of KLAIM processes.

2.1 An Overview of Linda

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called tuple space (TS). A tuple space is a collection (formally a multiset) of tuples, where a tuple is a sequence of actual fields, i.e., expressions or values, and formal fields, i.e., variables. *Pattern-matching* is used to select tuples in a TS. Two tuples match if they 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.

Linda provides just four primitives for manipulating tuples. Two (nonblocking) operations, **out**(t) and **eval**(t), permit tuples to be added to a TS. The operation **out**(t) adds the tuple resulting from the evaluation of t to a TS. The operation **eval**(t) differs from **out**(t) because t is first added to the TS and then a new concurrent process is created for evaluating the tuple; this is not available for matching until its evaluation has been completed. Two (possibly blocking) operations, **in**(t) and **read**(t), permit tuples to be accessed in the TS. The operation **in**(t) evaluates t and looks for a matching tuple t' in the TS. Whenever t' is found, it is removed from the TS. The corresponding values of t' are then assigned to the variables of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The operation **read**(t) differs from **in**(t) because the tuple t' selected by pattern-matching is not removed from the TS.

Nondeterminism is inherent in the definition of Linda primitives. It arises when more **in**/**read** operations are suspended while waiting for a tuple. When such a tuple becomes available, only one of the suspended operations is nondeterministically selected to proceed. Similarly, when an **in**/**read** operation has more than one matching tuple one is arbitrarily chosen.

The Linda programming paradigm is known as *Generative Communication* [20]. Indeed, once a tuple is added to a TS (generated), its life-time is independent of the producer process's life-time.

In the original proposal [20] two predicative (nonblocking) forms, **inp** and **readp**, were also part of the language. They yield true or false depending on whether the TS contains a tuple matching their argument. When returning true they retrieve/remove the matching tuple. We did not consider these predicates because they are functional duplicates of their nonpredicative counterparts and are difficult to implement in a distributed environment. They may require expensive checks and synchronizations over entire tuple spaces [29].

The Linda asynchronous communication model allows programmers to explicitly control interactions among processes via shared data and to use the same set of primitives both for data manipulation and for process synchronization. This has the advantage of rendering explicit all the interactions of a program with its environment. The original Linda primitives are, however, not completely adequate for programming distributed systems. For example, data protection and security, which are key features of mobile applications, are problematic because the Linda communication model cannot guarantee data privacy. Also, modular programming disciplines are awkward to follow in practice as there is no way to guarantee that tuples coming from different contexts are not mixed up when two modules are put together. Multiple tuple spaces [21] are a first step toward the solution of these problems. In this paper we perform a further step by adding structure to multiple tuple spaces and allowing explicit manipulation of localities and locality names.

2.2 KLAIM Processes

Hereafter, we shall exploit the syntactic categories listed below; all of them are followed by the symbols we will use (sometimes with indices) to refer to their elements.

- $\mathcal{S}$ (s) is a set of *sites* (or *physical localities*). A site can be considered as the address of a node where processes and tuple spaces are allocated.
- *Loc* (l) is a set of (*logical*) *localities*. A locality may be thought of as the symbolic name for a site. Localities permit structuring programs over distributed environments while ignoring their precise allocations. A distinguished locality `self` ($\in$ *Loc*) is assumed. Programs can use `self` to refer to their execution site.
- *Vloc* (u) is a set of *locality variables*.
- *Val* (v) is a set of basic values.
- *Var* (x) is a set of value variables.
- *Exp* (e) is the category of *value expressions*. These are built up from values and value variables, by using a set of operators (not specified here).
- Ψ (A) is a set of parameterized *process identifiers*. Parameters can be of three different types: process, locality, and value; for the sake of simplicity, we fix this ordering for the formal parameters of any process identifier.
- $\mathcal{X}$ (X) is a set of *process variables*.

For simplicity, we will use ℓ to denote both localities and locality variables. Moreover, $\tilde{\ell}$ will indicate sequences of localities and $\{\tilde{\ell}\}$ the set of localities in $\tilde{\ell}$. A similar notation will also be used for other kinds of sequences.

We will use the standard notation $e[e'/x]$ to indicate the substitution of the value expression e' for the variable x in e; $e[\tilde{e}'/\tilde{x}]$ will denote the simultaneous substitution of each $x \in \tilde{x}$ with the corresponding $e' \in \tilde{e}'$ in e.

Tuples are sequences of actual fields (i.e., expressions, processes, localities, or locality variables) and formal fields; these are denoted by "!*var*", where *var* is a generic variable. We shall use *fields*(t) to denote the set of fields of t.

The Linda operations to generate tuples (**out**), to spawn a new process (**eval**), to read tuples (**read**), and to remove tuples (**in**) are located, e.g., the operation **out**(t)@ℓ is used to place the tuple t in the tuple space located at ℓ. Our primitives generalize Linda's original ones. We have a modified **eval** primitive; it has processes as arguments rather than tuples, and thus permits mobile agents to be programmed. As will be clarified later (Section 3), action **eval**(**out**(t)@ℓ.**nil**)@ℓ can be used to simulate the "expected" behavior of action **eval**(t)@ℓ. New sites are created through the prefix **newloc**(u). This operation creates a "fresh" site that can be accessed via the locality variable u.

The operators for building processes are borrowed from Milner's CCS [30]. They are commonly used in process algebras and correspond to basic notions. Namely, **nil** stands for the process that cannot perform any action, $a.P$ stands for the process that first executes action a and then behaves like P, $P_1 \mid P_2$ stands for the parallel composition of P_1 and P_2, and $P_1 + P_2$ stands for the nondeterministic composition of P_1 and P_2.

KLAIM *terms* are given by the abstract syntax in Table 1. As a matter of notation, in the following we often shall write a instead of a.**nil**.

TABLE 1
PROCESSES SYNTAX

P	$::=$	**nil**	(null process)
	$\mid$	$a.P$	(action prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$P_1 + P_2$	(choice)
	$\mid$	X	(process variable)
	$\mid$	$A\langle\tilde{P}, \tilde{\ell}, \tilde{e}\rangle$	(process invocation)
a	$::=$	$\mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(u)$	
t	$::=$	$e \mid P \mid \ell \mid\ !x \mid\ !X \mid\ !u \mid t_1, t_2$	

Variables occurring in KLAIM process terms can be bound by prefixes. More precisely, prefixes **in**(t)@ℓ._ and **read**(t)@ℓ._ act as binders for variables in the formal fields of t. Prefix **newloc**(u)._ binds the locality variable u.

Process identifiers are used in recursive process definitions. It is assumed that each process identifier A has a *single* defining equation $A(\tilde{X}, \tilde{u}, \tilde{x}) \overset{def}{=} P$. All free (value, process, and locality) variables in P are contained in $\{\tilde{X}, \tilde{u}, \tilde{x}\}$, and all occurrences of process identifiers in P are *guarded* (i.e., each process identifier occurs within the scope of a blocking **in** / **read** prefix).

A *process* is a term without free variables; localities occurring in processes are considered as constants. In the next section, we will see that they are names whose meaning is defined (i.e., mapped onto sites) by coordinators. Both processes and localities are first-class data and can be manipulated and generated like any other data occurring in tuples. Processes have *higher-order* capabilities, in that they can be exchanged in communications.

2.3 KLAIM Nets

Coordination appears to be a key concept for modeling and designing heterogeneous, distributed, open-ended systems. It applies typically to systems consisting of a large number of software components, programmed independently, possibly with different programming languages, which may change their configuration during execution. Coordination languages provide the primitive for defining configurations and interaction protocols of sets of software agents. Systems are designed and developed in a structured way, starting from the basic computational components and adding suitable software modules called *coordinators*. This approach increases the potential reuse of both software agents and coordinators at the cost of acceptable overheads.

In this section, we introduce the KLAIM coordination language. It is designed to handle all the issues related to the physical distribution of processes. Moreover, it controls changes of network configuration. Changes may be due to the addition/deletion of software components and sites, or to the transmission of programs and resources.

Given a finite set of sites, a KLAIM *net* is a set of *nodes*. A KLAIM node is a triple (s, P, ρ) where s is a site and ρ is the *allocation environment*, i.e., a (partial) function from *Loc* to S. Hereafter, ϕ will denote the empty environment, and $[s/l]$ the environment that maps the locality l to the site s. Processes at each site can potentially access any other site of the net; however, site visibility is (locally) controllable via the allocation environment: A site s' is *visible* at the node (s, P, ρ) only if s' belongs to the image of ρ. Finally, we introduce an operation to stratify environments. If ρ_1, ρ_2 are allocation environments then $\rho_1 \bullet \rho_2$ is the environment defined by:

$$\rho_1 \bullet \rho_2(l) = \begin{cases} \rho_1(l) & \text{if } \rho(l) \text{ is defined} \\ \rho_2(l) & \text{otherwise} \end{cases}$$

In $\rho_1 \bullet \rho_2$, ρ_1 is the inner environment and ρ_2 is the outer environment.

The abstract syntax for KLAIM *nets* is given by the grammar in Table 2.

TABLE 2
NETS SYNTAX

N	$::=$	$s ::_\rho P$	(node)
	$\mid$	$N_1 \parallel N_2$	(net composition)

Given a net N, we assume the existence of a function *st* which returns the sites of N. The composition $N_1 \parallel N_2$ is defined only if $st(N_1) \cap st(N_2) = \emptyset$, thus we can consider a net just as a set of nodes. We say that a net N is *well-formed* if whenever $s ::_\rho P$ is a node of N then $\rho(\texttt{self}) = s$ and the image of ρ is included in $st(N)$. We will only consider well–formed nets. To lighten notations, the allocation environments will not report the binding for **`self`**.

REMARK 2.1. In the present formulation of KLAIM, located tuple spaces have no hierarchical structure, i.e., located tuple spaces are not nested. However, the nesting of located tuple spaces can easily be modeled. It suffices to extend KLAIM coordination language with a combinator to allocate a complete net. Hence, a hierarchical net would be written:

$$s ::_\rho [N]$$

where ρ is the allocation environment that now returns either localities or sequences of sites. The idea is that s is the site where the net N is allocated. Site s and its environment ρ can then be used to control all interactions between N and other nets.

Allocated nets are very similar in spirit to the multiple ambients of Cardelli and Gordon [9]. A complete investigation of allocated nets is beyond the scope of the present paper and will be the subject of a further work.

3 OPERATIONAL SEMANTICS

The two syntactic levels of KLAIM are reflected at the semantic level. The operational semantics of KLAIM is given in the SOS style [35] and proceeds in two steps. The first step defines the *symbolic semantics* that specifies parts of process commitments, i.e., the control on localities and the effects of the actions on the tuple spaces. The full description of process behaviors is given in the second step, which packages processes and data into a net.

3.1 Process Semantics

The labeled transition system for processes describes the possible evolutions of KLAIM processes without providing the actual allocation of processes and tuple spaces. For this reason, the corresponding operational semantics is called *symbolic* in that neither value and locality expressions nor tuples are evaluated.

To describe the effects of the evaluation of processes which are placed within tuples fields, we introduce the auxiliary term $P\{\rho\}$ which indicates the process P packaged with the allocation of localities specified by ρ. Terms of the form $P\{\rho\}$ are called closures. For the sake of simplicity, we will use P to range over closures as well.

The structural rules of the symbolic semantics are reported in Table 3. The *transition*

$$P \underset{\rho}{\overset{\mu}{\longrightarrow}} P'$$

describes the evolution to P' of the process P. The label of the transition $\langle \mu, \rho \rangle$ provides an abstract description of the activities performed in the evolution. For instance, $\mu = s(t)@\ell$ describes the output (sending) of tuple t in the tuple space specified by ℓ. Similarly, $\mu = n(u)@\texttt{self}$ can be thought of as the request for binding a fresh site to the variable u. The environment ρ records the local bindings that must be taken into account to evaluate μ. Our use of allocation environments in the transition labels is similar to the use of Boolean expressions in the operational framework of [24].

3.2 Net Semantics

Following [4], [31] the operational semantics of KLAIM coordination language is defined by a structural congruence and a reduction relation. The structural congruence incorporates the basic semantics of net parallel composition, while reduction describes the basic computational paradigm of interactions among processes inside a net.

Nets are defined up to a *structural congruence* $\equiv$. This is the smallest congruence such that $\parallel$ is associative and commutative.

TABLE 3
THE STRUCTURAL RULES OF SYMBOLIC SEMANTICS

$$\mathbf{out}(t)@\ell.P \xrightarrow[\phi]{s(t)@\ell} P \qquad \mathbf{eval}(Q)@\ell.P \xrightarrow[\phi]{e(Q)@\ell} P$$

$$\mathbf{in}(t)@\ell.P \xrightarrow[\phi]{i(t)@\ell} P \qquad \mathbf{read}(t)@\ell.P \xrightarrow[\phi]{r(t)@\ell} P$$

$$\mathbf{newloc}(u).P \xrightarrow[\phi]{n(u)@\mathtt{self}} P$$

$$\frac{P \xrightarrow[\rho]{\mu} P'}{P+Q \xrightarrow[\rho]{\mu} P'} \qquad \frac{P \xrightarrow[\rho]{\mu} P'}{Q+P \xrightarrow[\rho]{\mu} P'}$$

$$\frac{P \xrightarrow[\rho]{\mu} P'}{P \mid Q \xrightarrow[\rho]{\mu} P' \mid Q} \qquad \frac{P \xrightarrow[\rho]{\mu} P'}{Q \mid P \xrightarrow[\rho]{\mu} Q \mid P'}$$

$$\frac{P \xrightarrow[\rho']{\mu} P'}{P\{\rho\} \xrightarrow[\rho' \bullet \rho]{\mu} P'\{\rho\}} \qquad \frac{P[\widetilde{P}/\widetilde{X}, \widetilde{\ell}/\widetilde{u}, \widetilde{e}/\widetilde{x}] \xrightarrow[\rho]{\mu} P'}{A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e}\rangle \xrightarrow[\rho]{\mu} P'} \quad A(\widetilde{X}, \widetilde{u}, \widetilde{x}) \overset{def}{=} P$$

To avoid cumbersome notations, we use ℓ to denote localities, locality variables and sites, and assume that allocation environments are extended to sites but for these they act as the identity function. The operational semantics of nets exploits an evaluation mechanism for tuples, and a pattern-matching to select tuples in a tuple space. The evaluation function for tuples, $\mathcal{T}[\![\ ]\!]$, exploits the allocation environment to resolve locality names and relies on an evaluation mechanism, $\mathcal{E}[\![\ e\ ]\!]$, for closed expressions (i.e., expressions without free variables). $\mathcal{T}[\![\ e\ ]\!]$ is inductively defined over the syntax of tuples by the rules in Table 4, where we use $\mathcal{E}[\![\ e\ ]\!]$ to denote the value of the closed expression e; the evaluation of a process, say $\mathcal{T}[\![\ P\ ]\!]\,\rho$, yields a process closure, i.e., $P\{\rho\}$.

TABLE 4
TUPLE EVALUATION FUNCTION

$\mathcal{T}[\![\, e\,]\!]\rho$	$=$	$\mathcal{E}[\![\, e\,]\!]$	$\mathcal{T}[\![\, !\,x\,]\!]\rho$	$=$	$!\,x$
$\mathcal{T}[\![\, P\,]\!]\rho$	$=$	$P\{\rho\}$	$\mathcal{T}[\![\, !\,X\,]\!]\rho$	$=$	$!\,X$
$\mathcal{T}[\![\, \ell\,]\!]\rho$	$=$	$\rho(\ell)$	$\mathcal{T}[\![\, !\,u\,]\!]\rho$	$=$	$!\,u$
$\mathcal{T}[\![\, t_1, t_2\,]\!]\rho$	$=$	$\mathcal{T}[\![\, t_1\,]\!]\rho, \mathcal{T}[\![\, t_2\,]\!]\rho$			

The rules defining the pattern-matching predicate are reported in Table 5.

TABLE 5
THE MATCHING RULES

$$\begin{array}{ccc}
match(v,v) & match(P,P) & match(s,s) \\[2ex]
match(!\,x,v) & match(!\,X,P) & match(!\,u,s) \\[2ex]
\dfrac{match(et_1,et_2)}{match(et_2,et_1)} & \multicolumn{2}{c}{\dfrac{match(et_1,et_2) \quad match(et_3,et_4)}{match((et_1,et_3),(et_2,et_4))}}
\end{array}$$

As in [18], [37], we model tuples as processes, and we introduce auxiliary processes to denote *evaluated tuples*, referred to as *et*. Thus, KLAIM syntax is extended with the process **out**(*et*) whose symbolic semantics are expressed by the following structural rule:

$$\mathbf{out}(et) \xrightarrow[\phi]{o(et)@\mathtt{self}} \mathbf{nil}\,.$$

Moreover, we use sites alike localities and locality variables.

The reduction rules of nets (rules in Table 6, and rules (11) and (12)) clearly distinguish between local and remote operations performed by located processes and provide a formal model to guide the implementation.

The evaluation of an **out** operation modifies a tuple space. Rule (1) adds a new tuple to the local tuple space of the process. Rule (2), on the other hand, adds a new tuple to the remote tuple space located at ℓ_2. In the latter rule, the evaluation of the tuple t depends on the allocation environment $\rho \bullet \rho_1$. This corresponds to having a *static scoping* discipline for the remote generation of tuples. Moreover, if the tuple t contains a field with a process, the corresponding field of the evaluated tuple *et* contains a closure. Hence, processes in a tuple are transmitted together with their local allocation environment.

A *dynamic scoping* strategy is adopted for the **eval** operation, described by rules (3) and (4). In this case the process spawned in the remote node is transmitted *without* the local allocation environment, and its execution is influenced by the remote allocation environment ρ_2.

For the communication operations **in** and **read** note that *in* modifies the tuple space (see rules (5) and (6)) while **read** does not (in the conclusions of rules (7) and (8) the tuple space encompassed within process P_2 is left unchanged by process evolution). Obviously, we have to distinguish between local rules ((5) and (7)) and remote rules ((6) and (8)).

Let us consider rule (5) (rules (6), (7), and (8) can be interpreted similarly). It says that a process can perform an **in** action at the local tuple space by synchronizing with a process which represents a matching tuple. The result of this synchronization is that the tuple is consumed, i.e., the corresponding process becomes **nil**, and its values are used to replace the corresponding (free) variables of the process which has performed the **in** operation.

Finally, rule (9) describes the asynchronous evolution of subcomponents of a node.

Rules (1)–(9) may modify the structure of the nodes of the net but they cannot introduce new nodes. The creation of a new node is described by rule (10). The environment of a new node is obtained from that of the creating one (with the obvious update for the `self` locality). The underlying idea is that the new node inherits all the knowledge about localities of the creating node.

REMARK 3.1. Obviously, other design choices could have been made. An alternative formulation of the rule for the creation of a new node is:

$$\frac{P \xrightarrow[\rho']{n(u)@\mathtt{self}} P' \quad s' \in S, \quad s' \text{ fresh}}{s ::_{\rho} P \rightarrowtail s\ ::_{\rho} P'[s'/u] \| s' ::_{[s'/\mathtt{self}]\bullet\phi} \mathbf{nil}}$$

The rationale behind this choice (adopted in [39]) is that any new node has no knowledge of the rest of the net.

TABLE 6
THE REDUCTION RELATION: PROCESS INTERACTIONS

$$\frac{P \xrightarrow[\rho']{s(t)@\ell} P' \qquad s = \rho' \bullet \rho(\ell) \qquad et = \mathcal{T}[\![t]\!]_{\rho' \bullet \rho}}{s ::_{\rho} P \rightarrowtail s ::_{\rho} P' \mid \mathbf{out}(et)} \tag{1}$$

$$\frac{P_1 \xrightarrow[\rho]{s(t)@\ell} P_1' \qquad s_2 = \rho \bullet \rho_1(\ell) \qquad et = \mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} P_2 \mid \mathbf{out}(et)} \tag{2}$$

$$\frac{P \xrightarrow[\rho']{e(Q)@\ell} P' \qquad s = \rho' \bullet \rho(\ell)}{s ::_{\rho} P \rightarrowtail s ::_{\rho} Q \mid P'} \tag{3}$$

$$\frac{P_1 \xrightarrow[\rho]{e(Q)@\ell} P_1' \qquad s_2 = \rho \bullet \rho_1(\ell)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \parallel s_2 ::_{\rho_2} Q \mid P_2} \tag{4}$$

$$\frac{P_1 \xrightarrow[\rho']{i(t)@\ell} P_1' \qquad s = \rho' \bullet \rho(\ell) \qquad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_{\rho} P_1 | P_2 \rightarrowtail s ::_{\rho} P_1'[et/\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}] | P_2'} \tag{5}$$

$$\frac{P_1 \xrightarrow[\rho]{i(t)@\ell} P_1' \qquad s_2 = \rho \bullet \rho_1(\ell) \qquad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2'} \tag{6}$$

$$\frac{P_1 \xrightarrow[\rho']{r(t)@\ell} P_1' \qquad s = \rho' \bullet \rho(\ell) \qquad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}, et)}{s ::_{\rho} P_1 | P_2 \rightarrowtail s ::_{\rho} P_1'[et/\mathcal{T}[\![t]\!]_{\rho' \bullet \rho}] | P_2} \tag{7}$$

$$\frac{P_1 \xrightarrow[\rho]{r(t)@\ell} P_1' \qquad s_2 = \rho \bullet \rho_1(\ell) \qquad P_2 \xrightarrow[\phi]{o(et)@\mathtt{self}} P_2' \qquad match(\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}, et)}{s_1 ::_{\rho_1} P_1 \parallel s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1'[et/\mathcal{T}[\![t]\!]_{\rho \bullet \rho_1}] \parallel s_2 ::_{\rho_2} P_2} \tag{8}$$

$$\frac{s ::_{\rho} P_1 \rightarrowtail s ::_{\rho} P_1'}{s ::_{\rho} P_1 | P_2 \rightarrowtail s ::_{\rho} P_1' | P_2} \tag{9}$$

$$\frac{P \xrightarrow[\rho']{n(u)@\mathtt{self}} P' \qquad s' \in \mathcal{S} : s' \neq s}{s ::_{\rho} P \rightarrowtail s ::_{\rho} P'[s'/u] \parallel s' ::_{[s'/\mathtt{self}] \bullet \rho} \mathbf{nil}} \tag{10}$$

To conclude the description of the reduction relation, we have to say how reduction behaves in presence of the operator of parallel composition of nets. Since the composition $N_1 \parallel N_2$ is defined only if $st(N_1) \cap st(N_2) = \emptyset$, we have:

$$\frac{N_1 \rightarrowtail N_1' \qquad st(N_1') \cap st(N_2) = \emptyset}{N_1 \parallel N_2 \rightarrowtail N_1' \parallel N_2} \tag{11}$$

Finally, we have to say how reduction behaves with respect to structural congruence. We have:

$$\frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2' \qquad N_2 \equiv N'}{N_2 \rightarrowtail N'} \tag{12}$$

REMARK 3.2. Despite the different programming paradigms, there are interesting similarities between Telescript and KLAIM. General Magic Telescript [41] is an object oriented language designed for network programming. A central concept in Telescript is the concept of *place*, which corresponds to our sites. A place can be thought of as the stationary process that can accept mobile agents. Agents travel from one place to another by invoking the **go** operation. This operation requires the agent's destination place (the ticket) and the route of the trip. The main advantage of KLAIM's approach is that the "possible stationary processes" can be programmed via the notion of locality without requiring the precise physical distribution of places. In other words, localities provide a powerful abstraction mechanism over sites. There are also some analogies between our **eval**/**out** operations and Telescript **go** operation: both allow mobile agents to be programmed.

REMARK 3.3. Several theoretical works in noninterleaving semantics of process calculi have adopted the notion of locality to capture logical distribution of processes (e.g., [6], [13] and the references therein). The basic idea of these approaches is to allow the observation of actions together with the locations (access paths) where they take place. In our approach, localities are not used as a tool for observing the distribution of processes, but rather as a programming device to structure and control the distribution of processes and data. The formal models presented in [2], [19] are closely related to the work presented here. These approaches deal with mobility much like π–calculus (channel and locality names can be passed in interactions). Significantly, localities in KLAIM can be used to simulate the private name passing and the scope extrusion mechanisms of π–calculus, so that a natural encoding of (asynchronous) π–calculus in KLAIM can be easily programmed.

3.3 Scoping and Mobility

The role of a net is to allocate and coordinate a set of processes. Hence, beyond formally describing all the issues related to physical distribution, net semantics is essential to study migratory applications and for understanding design decisions before carrying out an implementation. This can be better understood by analyzing the effects of choosing specific scoping disciplines on mobile agents when accessing tuple spaces.

The operational semantics of nets adopts a static scoping discipline for the evaluation of **out** operations. On the other hand, a dynamic scope discipline is adopted for remote **eval** operations: The meaning of localities used by a process spawned at a remote site depends on the remote allocation environment.

Indeed, whenever a process P located at the site s_1 wishes to insert a tuple t into the remote tuple space located at s_2, the local environment of P, namely ρ_1, is used for evaluating t. A dynamic scoping discipline for **out** can be obtained by replacing rule (2) in Table 6 with the following:

$$\frac{P_1 \xrightarrow[\rho]{s(t)@\ell} P_1' \quad s_2 = \rho \bullet \rho_1(\ell) \quad et = \mathcal{T}[\![t]\!]\rho_2}{s_1 ::_{\rho_1} \| \, s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \, \| \, s_2 ::_{\rho_2} P_2 \mid \mathbf{out}(et)}$$

where the remote environment ρ_2 is used for evaluating t.

REMARK 3.4. Alternatively, we could also use the rule:

$$\frac{P_1 \xrightarrow[\rho]{s(t)@\ell} P_1' \quad s_2 = \rho \bullet \rho_1(\ell)}{s_1 ::_{\rho_1} P_1 \| s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \| s_2 ::_{\rho_2} P_2 \mid \mathbf{out}(t)@\,\texttt{self}.\,\mathbf{nil}}$$

Namely, a process is placed in s_2 which will eventually take care of the local evaluation of the tuple t.

Dynamic scoping for **out** can be also simulated (without any modification to the operational rules for nets) by writing **eval**(**out**(t)@`self`)@ℓ.P instead of **out**(t)@ℓ.P. The execution of **eval** spawns process **out**(t)@`self` at site s_2 (resulting from the evaluation of ℓ) and, therefore, t is evaluated by using the local environment at s_2.

When process P located at s_1 wants to spawn a process Q at the remote site s_2, a dynamic scoping discipline is followed. The local environment ρ_2 is used for giving meaning to the localities which may be referred in Q. A static scoping discipline for **eval** can be obtained by spawning $Q\{\rho_1\}$ rather than Q. More precisely, rule (4) in Table 6 could be replaced by the following:

$$\frac{P_1 \xrightarrow[\rho]{e(Q)@\ell} P_1' \quad s_2 = \rho \bullet \rho_1(\ell) \quad Q' = Q\{\rho_1\}}{s_1 ::_{\rho_1} P_1 \| s_2 ::_{\rho_2} P_2 \rightarrowtail s_1 ::_{\rho_1} P_1' \| s_2 ::_{\rho_2} Q' \mid P_2}$$

In this case the remote spawning of process Q consists in transmitting Q packaged with its allocation environment ρ_1.

Again, **eval** with static scoping can be simulated via the primitives of the language, in particular, by passing processes (and then closures) as fields of tuples and using private localities to store intermediate results. With this in mind, we can write **newloc**(u).**out**(Q)@u.**in**(!X)@u.**eval** (X)@ℓ.P instead of **eval**(Q)@ℓ.P. When **eval**(X) is executed at site s_2, X is bound to the process Q packaged with ρ_1. Hence, a closure rather than a plain process is activated at site s_2, which is different from the case of **eval**(Q).

4 TYPING AND SECURITY

Security, e.g., *privacy* and *integrity* of data, is a key issue in the development of mobile applications. One can easily imagine malicious mobile agents attempting to access private information. A server receiving a mobile agent for execution thus needs to impose strong requirements to ensure that the agent will not violate privacy and jeopardize the integrity of the information. Similarly, mobile agents must ensure that their execution at the server site will not damage them or compromise their security.

In this section, we introduce a type system for KLAIM and show how it can be used to statically enforce security properties. More precisely, the type system permits one to check whether the operations KLAIM processes intend to perform over the sites of a net really do comply with their access rights.

The typing analysis of KLAIM programs is structured into two phases reflecting the two-level syntax of KLAIM. The first phase deduces process intentions (read, write, withdraw, execute, ...) in relation to the various localities they are willing to interact with or they want to migrate to. This is done by an inference system which assigns types to processes, and also, partially, checks whether these behave in accordance with their declared intentions. The second phase of the typing analysis checks whether each process has the necessary rights to perform the intended operations, i.e., it does not violate the access rights as granted by the net coordinator.

4.1 Types

We will use $\{r, i, o, e, n\}$ to indicate the set of process *capabilities*; r denotes the capability to execute a **read** operation, i the capability to execute an **in** operation, etc.

Polarities are nonempty subsets of $\{r, i, o, e, n\}$. We use Π, ranged over by π (which may be indexed), to denote the set of all polarities. Polarities are used differently by processes and nets. The polarity of a locality or of a locality variable, say ℓ, within a process contains information about the operations the process intends to perform at ℓ. In a net, on the other hand, polarities are used to fix access rights. Type checking will guarantee that only intentions that match access rights, as granted by the coordinator, are allowed.

Orderings between polarities can be used to model hierarchies of access rights. Obviously, if a process is able to perform an **in** operation at ℓ then it is also able to perform a **read** at ℓ. Also, type checking should ensure that, if a process has capabilities π, then it can execute all operations that require capabilities smaller (greater, in the ordering $\sqsubseteq_\Pi$ defined below) than π. These intuitions lead to the *subpolarity relation*, obtained as the least reflexive and transitive relation induced by the following rules:

$$\{i\} \sqsubseteq_\Pi \{r\} \qquad \frac{\pi_1 \subseteq \pi_2}{\pi_2 \sqsubseteq_\Pi \pi_1} \qquad \frac{\pi_1 \sqsubseteq_\Pi \pi_1' \quad \pi_2 \sqsubseteq_\Pi \pi_2'}{(\pi_1 \cup \pi_2) \sqsubseteq_\Pi (\pi_1' \cup \pi_2')}$$

One could think of associating a polarity with each process or with each locality to completely characterize the intentions of processes and the rights of localities. It is clear that this would not be enough to take into account process migrations and the different access rights of the different localities.

An obvious choice, for assigning types to a process, would be to associate with it a single polarity that describes all the operations the process intends to perform, while ignoring the specific localities it refers to. However, in this way, we would not characterize different intentions relative to different localities. Associating polarities with each of the localities referred to within a process would also be unsatisfactory. It hinders the possibility of keeping track of the capabilities of remotely executed processes, which might be different from those of sender processes. For example, consider a process that does not have the right to access a remote tuple space (e.g., a database), but does have the right to send a process for remote execution at a (server) node that is willing to grant the necessary right.

To take into account remote executions (*migrations*) of processes, we need to further structure our types and to associate with each locality not just a polarity but also the type that is required for the processes executed at that locality.

A *type* is a finite map that assigns pairs consisting of polarities and types to both localities and locality variables. The first component of the pair associated with ℓ describes the polarity of ℓ, while the second describes the types of the processes executed at ℓ.

KLAIM types, ranged over by δ, are elements of a universe which is defined by the following domain equation:

$$\Delta = Fin((Loc \cup VLoc) \mapsto (\Pi \times \Delta))_\perp.$$

The construction of Δ rests on a standard construction over *complete partial orders* (cpo). Let $\langle D, \sqsubseteq_D \rangle$, be a cpo; then $H(D)$ is the set of partial functions with finite domain from $Loc \cup Vloc$ to the cpo $\Pi \times D$ defined by:

$$H(D) = Fin((Loc \cup VLoc) \mapsto (\Pi \times D))_\perp$$

This set of functions can be ordered via the relation $\sqsubseteq_{H(D)}$ stating that the more defined the partial function the smaller it is.

1) $\perp \sqsubseteq_{H(D)} f$, for all $f \in H(D)$
2) $f \sqsubseteq_{H(D)} g$ when
   - $dom(g) \subseteq dom(f)$, and
   - $\forall \ell \in dom(g) : f(\ell) \sqsubseteq_{\Pi \times D} g(\ell)$, where $\sqsubseteq_{\Pi \times D}$ is the obvious ordering on $\Pi \times D$.

It is not difficult to show that if $\langle D, \sqsubseteq_D \rangle$ is a (ω–algebraic) cpo then also $\langle H(D), \sqsubseteq_{H(D)} \rangle$ is a (ω–algebraic) cpo.

Let $\langle \Delta, \preccurlyeq \rangle$ be the initial solution[1] of the recursive domain equation for Δ; $\preccurlyeq$ is called the *subtype relation*. As usual, $\sqcap$ shall denote the greatest lower bound, and ϕ shall denote the element of Δ with empty domain. If $\delta \in \Delta$, then $\delta^1(\ell)$ is used to denote the *i*th component of the pair $\delta(\ell)$, if it is defined; otherwise, $\delta^1(\ell)$ yields $\emptyset$ and $\delta^2(\ell)$ yields ϕ. Moreover, $\delta \downarrow \ell$ denotes the greatest lower bound of the set $\{\delta^2(\ell)\} \cup \{\delta(\ell') \downarrow \ell \mid \delta(\ell') \text{ is defined}\}$.

Notation $\delta[\delta^1(\ell) := \pi]$ denotes a type δ_1 such that $\delta_1^1(\ell) = \pi$, $\delta_1^1(\ell) = \phi$, if $\delta(\ell)$ is undefined, $\delta_1^2(\ell) = \delta^2(\ell)$ otherwise, and $\delta_1(\ell') = \delta(\ell')$ for $\ell' \neq \ell$. Notation $\delta[\delta_1 / \delta^2(\ell)]$ has the same effect as a substitution (thus, $\delta[\delta_1 / \delta^2(\ell)]$ denotes δ itself if $\delta(\ell)$ is undefined).

The typed version of KLAIM is obtained by associating a type with locality variables and with process variables whenever they are bound. Hereafter, for the sake of simplicity, we will also call the typed version of the language KLAIM.

The abstract syntax of *terms* (*processes*, as usual, are closed terms) is reported in Table 7. Recall that ℓ stands for a generic locality or locality variable. To avoid name clashing and thus overloading of types, we will assume that *Vloc*, the set of locality variables, is partitioned into two subsets: *NVloc*, used as arguments of **newloc**, and *TVloc*, used as formals of tuples.

TABLE 7
TYPED KLAIM SYNTAX

P	$::=$	$\mathbf{nil}$	$a.P$	$P_1 \mid P_2$	$P_1 + P_2$	X	$A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle$	
a	$::=$	$\mathbf{out}(t)@\ell$	$\mathbf{in}(t)@\ell$	$\mathbf{read}(t)@\ell$	$\mathbf{eval}(P)@\ell$	$\mathbf{newloc}(u : \delta)$		
t	$::=$	e	P	ℓ	$!\,x$	$!\,X : \delta$	$!\,u : \delta$	t_1, t_2

A type is associated with process and locality parameters of process identifiers and, as usual, it is assumed that each process identifier A has a *single* defining equation $A(\widetilde{X} : \widetilde{\delta}, \widetilde{u} : \widetilde{\delta}, \widetilde{x}) \overset{def}{=} P$.

[1] The construction H on cpos may be straightforwardly turned into a functor $\mathcal{H}$ in the category $\underline{\mathbf{CPO}}^{\mathbf{E}}$, the category of cpos with embeddings as morphisms. The action of the functor $\mathcal{H}$ on cpos is defined as for H. If $i : D \lhd D'$ is an embedding, $\mathcal{H}(i) : \mathcal{H}(D) \to \mathcal{H}(D')$ (the action of the functor on embeddings) is obtained as:

$$(\mathcal{H}(i))(\perp) = \perp \qquad (\mathcal{H}(i))(f) = i \circ f.$$

By using standard techniques, we can prove that $\mathcal{H}$ is a continuous and covariant functor in $\underline{\mathbf{CPO}}^{\mathbf{E}}$ which preserves ω–algebraicity [27]. Therefore, the theory in [36] ensures the existence and uniqueness in $\underline{\mathbf{CPO}}^{\mathbf{E}}$ of the initial fixed point of the functor $\mathcal{H}$, i.e., the initial solution of the recursive domain equation for Δ.

We are now ready to introduce the formal syntax of typed nets, whose role is to allocate and coordinate processes, and to assign access rights. The type of sites is similar to that of processes: it associates pairs ⟨polarity, type⟩ with localities and locality variables. This is declared by means of two functions, Λ and Υ. For each site s of the net, Λ describes the access rights of processes located at s on the other sites of the net, while Υ describes the locality variables that processes located at s may use.

A *net* is a triple $N : \langle \Lambda, \Upsilon \rangle$ where N is as defined in Section 2.3, and Λ and Υ have the following structure:

$$\Lambda : st(N) \to (st(N) \to \Pi) \text{ and } \Upsilon : st(N) \to VLoc.$$

4.2 Deriving Process Types

This section presents an inference system that assigns types to processes. The type system records the operations that processes are willing to perform at specific localities and checks whether process operations comply with the declared types of the variables.

Type contexts Γ are functions mapping process variables and identifiers into types. Hereafter, ϕ will denote the empty context. The auxiliary function *update*, defined structurally over tuples syntax, will be used to update type contexts; it behaves like the identity function for all fields but !X : δ. Formally, it is defined by:

$$update(\Gamma, t) = \begin{cases} update(update(\Gamma, t_1), t_2) & \text{if } t = t_1, t_2 \\ \Gamma[\delta / X] & \text{if } t = !X : \delta \\ \Gamma & \text{otherwise} \end{cases}$$

The type judgments for processes take the form $\Gamma \vdash P : \delta$ where Γ is a type context providing the type of process variables and identifiers of P. A statement such as $\Gamma \vdash P : \delta$ asserts that the capabilities of P are those in δ, within the context Γ.

The type of a process variable or identifier is always determined by the type context, Γ, that has been set up by the other inference rules. Definedness of Γ(X) (Γ(A)) is guaranteed by the fact that processes are closed terms.

$$\frac{}{\Gamma \vdash X : \Gamma(X)} \qquad \frac{}{\Gamma \vdash A : \Gamma(A)}$$

The simplest process (the null process) has no capability.

$$\Gamma \vdash \mathbf{nil} : \phi$$

The process **out**(t)@ℓ.P puts the tuple t in the tuple space whose address is specified by ℓ and then behaves like P. The typing rule of the out operation

$$\frac{\Gamma \vdash P : \delta}{\Gamma \vdash \mathbf{out}(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{o\}]}$$

states that the type of **out**(t)@ℓ.P (possibly) extends that of P at ℓ with capability o. Since **out** is not a binder, P is typed within the same context (Γ) as **out**(t)@ℓ.P.

The typing rules for **read** and **in** update the context with the types of the process variables they bind. The second half of their premises checks whether process P does not misuse the locality variables bound by **read** and **in**. Thus, for each locality variable u with type δ_u one checks that the remote operations of P at u ($\delta \overline{\downarrow} u$) really do respect δ_u. The resulting type is obtained by extending the type of P at ℓ with the corresponding capability (r or i).

$$\frac{update(\Gamma, t) \vdash P : \delta \qquad \delta_u \preccurlyeq \delta \overline{\downarrow}\, u \text{ for all} (!u : \delta_u) \in fields(t)}{\Gamma \vdash \mathbf{read}(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{r\}]}$$

$$\frac{update(\Gamma, t) \vdash P : \delta \qquad \delta_u \preccurlyeq \delta \overline{\downarrow}\, u \text{ for all} (!u : \delta_u) \in fields(t)}{\Gamma \vdash \mathbf{in}(t)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{i\}]}$$

where $\{\tilde{u}\}$ are all the locality variables bound by **read** and **in**.

The typing rule of **eval** extends the type of P at ℓ with e and records that the remote operations of P have to be extended with those (δ') of the spawned process Q.

$$\frac{\Gamma \vdash P : \delta \qquad \Gamma \vdash Q : \delta'}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P : \delta[\delta^1(\ell) := \delta^1(\ell) \cup \{e\}][\delta^2(\ell) \sqcap \delta') / \delta^2(\ell)]}$$

The typing rule for **newloc** extends the type of P at `self` with n and at u with the type δ' declared for u, while it checks whether the operations that P is willing to perform at u ($\delta^2(u)$) comply with δ'.

$$\frac{\Gamma \vdash P : \delta \qquad \delta' \preccurlyeq \delta \overline{\downarrow}\, u}{\Gamma \vdash \mathbf{newloc}(!u : \delta').P : \delta[\delta^1(\mathtt{self}) := \delta^1(\mathtt{self}) \cup \{n\}][\delta'/\delta^2(u)]}$$

The typing rules for parallel composition and choice state that the intentions of the composed processes are in both cases the union, formally the greatest lower bound, of those of the components. The binding context is left unchanged.

$$\frac{\Gamma \vdash P : \delta_1 \qquad \Gamma \vdash Q : \delta_2}{\Gamma \vdash P + Q : \delta_1 \sqcap \delta_2} \qquad \frac{\Gamma \vdash P : \delta_1 \qquad \Gamma \vdash P : \delta_2}{\Gamma \vdash P \mid Q : \delta_1 \sqcap \delta_2}$$

The typing rule for process definition, first updates the type context with the types of the process variables that occur as parameters of A and with a candidate type δ for A. The resulting context is exploited to infer the type δ for P. Secondly, for each formal locality variable u_i, one checks that the operations of P at u_i (i.e., $\delta^2(u_i)$) match the type declaration δ_{u_i}. Finally, the inferred type is assigned to A.

$$\frac{\Gamma[\widetilde{\delta_X} / \tilde{X}][\delta / A] \vdash P : \delta \qquad \delta_{u_i} \preccurlyeq \delta^2(u_i) \text{ for all } u_i \in \{\tilde{u}\}}{\Gamma \vdash A : \delta}$$

where $A(\tilde{X} : \widetilde{\delta_X}, \tilde{u} : \widetilde{\delta_u}, \tilde{x}) \overset{def}{=} P$ is the defining equation for the process identifier A.

The typing rule for process invocation, first determines the type of the process identifier and those of the process arguments. It then, checks whether each of the types inferred for the process arguments agree with the one of the corresponding formal parameter. No requirement is imposed on the other arguments. The type of locality variables is controlled when one of the rules for **in**, **read,** and **newloc** is applied. Localities are controlled when well-typedness of nets is checked.

$$\frac{\Gamma \vdash A : \delta \qquad \Gamma \vdash P_i : \delta_i \text{ and } \delta_{X_i} \preccurlyeq \delta_i \text{ for all } P_i \in \{\tilde{P}\}}{\Gamma \vdash A\langle \tilde{P}, \tilde{\ell}, \tilde{e} \rangle : \delta\{\tilde{\ell} / \tilde{u}\}}$$

where $\delta\{\tilde{\ell}/\tilde{u}\}$ is such that:

$$\delta\{\tilde{\ell}/\tilde{u}\}(\ell_i) = \left\langle \delta^1(u_i) \cup \delta^1(\ell_i), (\delta^2(u_i) \sqcap \delta^2(\ell_i)) \{\tilde{\ell}/\tilde{u}\} \right\rangle,$$

for $\ell_i \in \{\tilde{\ell}\}$, and

$$\delta\{\tilde{\ell}/\tilde{u}\}(\ell') = \left\langle \delta^1(\ell'), \delta^2(\ell')\{\tilde{\ell}/\tilde{u}\} \right\rangle,$$

for $\ell' \notin \{\tilde{\ell}\}$ such that $\delta(\ell')$ is defined. The inferred type states that $A(\tilde{P}, \tilde{\ell}, \tilde{e})$ intends to perform at $\ell_i \in \{\tilde{\ell}\}$ and $u_i \in \{\tilde{u}\}$ the same operations that $A(\tilde{X}, \tilde{u}, \tilde{x})$ intends to perform at u_i. Indeed, statically we are unable to establish which occurrences of $u_i \in \{\tilde{u}\}$ in δ must be replaced by ℓ_i.

4.3 Typing Nets

This section presents the criteria for establishing whether a net is well-typed. The types of the processes in a net will be required to agree with those of the sites where they are located. More specifically, the types of the processes, as determined by the type inference system, are checked against those fixed by the net coordinator, taking into account where each process has been located.

The pair of functions, Λ and Υ associate a type with each site of a net. This is the type that is compared with the one for located processes (which expresses their expected behavior) to check whether the net is well-typed.

Given a net $N : (\Lambda, \Upsilon)$, the type δ_s of each site $s \in st(N)$ is obtained as: $\forall \ell \in (dom(\rho_s) \cup dom(\Upsilon(s)))$:

$$\delta_s(\ell) = \begin{cases} \left\langle \Lambda(s)(\rho_s(\ell)), \delta_{\rho_s(\ell)} \right\rangle & \text{if } \ell \in dom(\rho_s) \\ \left\langle \{i, o, e, n\}, \delta_s \right\rangle & \text{if } \ell \in dom(\Upsilon(s)) \cap NVLoc \\ \left\langle \{i, o, e, n\}, \perp \right\rangle & \text{if } \ell \in dom(\Upsilon(s)) \cap TVLoc \end{cases}$$

Notice that, for any site s, δ_s is well-defined since, by definition of net, if $\ell \in dom(\rho_s)$ then $\Lambda(s)(\rho_s(\ell))$ is a polarity. Namely, the first item of the definition of δ_s uses the allocation environment ρ_s of s to determine the site associated to ℓ, hence its polarity and type. The last two items deal with locality variables; the only restriction we statically put on them is that a fresh node inherits the rights of the creating one.

In [17] a soundness theorem is proved, namely well–typed KLAIM nets (and processes) never lead to run-time errors due to misuse of access rights. For a net be well–typed, it will be required that the types of the processes in the net agree with the access rights of the sites where they are located. More specifically, the types of the processes, as derived by the type inference system, are checked against those fixed by the net coordinator, while taking into account where each process has been located. The soundness theorem establishes that well-typedness is an *invariant* of the operational semantics. This result is essentially a variant of standard *subject reduction*, that takes into account the fact that new sites can be dynamically created. The soundness theorem and the related technicalities are not presented here since they are not needed to appreciate the primitives and the pragmatics of KLAIM.

To highlight the utility of KLAIM types, let us consider a system composed of a process *Server*, which makes available in its local space a tuple containing locality l, and two identical processes $Client_1$ and $Client_2$, which access the tuple space at l_S to read an address u and then send process P for execution at u.

$$Server \overset{def}{=} \mathbf{out}(l)@\mathtt{self}.\,\mathbf{nil}$$

$$Client_i \overset{def}{=} \mathbf{read}(!\,u)@\,l_S.\,\mathbf{eval}(P)@\,u.\,\mathbf{nil}$$

If P has type δ, each process $Client_i$, $i = 1, 2$, has type:

$$\delta_c = l_S \mapsto \left\langle \{r\}, \phi \right\rangle,\ u \mapsto \left\langle \{e\}, \delta \right\rangle$$

Suppose now that only $Client_1$ has the right to send processes for evaluation at the location denoted by u. The net coordinator can thus allocate *Server* on site s and the two processes *Client* on sites s_1 and s_2, and can give the following access rights to s_1 and s_2

$$\delta_{s_1} = s \mapsto \left\langle \{r\}, \phi \right\rangle,\ u \mapsto \left\langle \{e\}, \delta \right\rangle$$
$$\delta_{s_2} = s \mapsto \left\langle \{r\}, \phi \right\rangle.$$

REMARK 4.1. There are some similarities between types in KLAIM and Telescript [41] *permit* and *authority*. The latter are used to limit the access rights of mobile agents.[2] The advantage of our approach is that the use of the type system makes mechanical static verifications of access rights possible.

Type systems have already been proposed for calculi of mobile processes, though not addressing security issues. Here, we mention the type system proposed by Pierce and Sangiorgi [34] and refined by Kobayashi, Pierce, and Turner [28]. In [34], a type system is developed for π-calculus [32] which uses types of channels to record information on whether channels are used to *read* or to *write*. This type system was extended in [28] by associating *multiplicities* with types in order to describe how many times each channel can be used. The main difference with our approach lies in the treatment of localities and, more importantly, in the role played by type information at the level of the net coordinator to check and enforce access rights of processes.

The present work shares parts of its underlying rationale with the work by Volpano and Smith [40], though those authors only consider a *sequential* procedural language and the type system is used to control a specific *noninterference* security property.

5 Programming Mobile Code Applications

In this section we illustrate how to use KLAIM to program *Mobile Code Applications* (MCAs). In the programming examples, we assume that natural numbers and identifiers are basic values.

MCAs are distributed applications whose distinctive feature is the exploitation of "code mobility." According to the classification proposed in [14], we can single out three paradigms, apart from the traditional *client-server* paradigm (CS), which are largely used to build MCAs:

- *Remote Evaluation* (RE). Any component of a distributed application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.

[2] In Telescript an agent permit can also specify *allowances* of a mobile agent, e.g., the maximum lifetime in seconds, the maximum size in bytes, etc.

- *Mobile Agent* (MA). A process (i.e., a program and an associated state of execution) on a given node of a network can migrate to a different node where it continues its execution from the current state.
- *Code On-Demand* (COD). A component of a distributed application running on a given node, can dynamically download from a different component and link the code to perform a given task.

Suitable programming constructs are needed to support these approaches. Indeed, several programming languages, such as Java [3], Facile [23], Obliq [7], and Telescript [41] were designed to provide facilities for process mobility and distribution; see [14] for a detailed survey.

Our aim here is to show, by means of simple programming examples, that the KLAIM programming constructs are powerful enough to implement the programming paradigms of MCAs.

Both the CS and RE paradigms can be programmed by exploiting the flexibility of KLAIM data structures, i.e., tuples. Indeed, tuple fields may contain both data values and processes (i.e., program codes). Let us now show how to program RE (which is basically a CS in a language with higher order facilities like KLAIM). Suppose we want to require that server located at location l executes (evaluates) code P where the values $v_1, \ldots, v_n$ must be assigned to variables $x_1, \ldots, x_n$. To this end, we can use the instruction:

$$\mathbf{out}(\mathbf{in}(!y_1, \ldots, !y_n)@l.\,A\left\langle y_1, \ldots, y_n\right\rangle, v_1, \ldots, v_n)@l$$

where we assume that $A(x_1, \ldots, x_n) \overset{def}{=} P$ and that the server performs:

$$\mathbf{in}(!X, !x_1, \ldots, !x_n)@\mathtt{self}.\,\mathbf{out}(x_1, \ldots, x_n)@\mathtt{self}.\,X$$

or a similar activity.

Suppose now that we want to execute process P at a (perhaps remote) location l, the paradigm MA can be implemented by means of:

- the instruction $\mathbf{eval}(P)@l$, if a dynamic scoping discipline for resolving location names is adopted,
- the sequence $\mathbf{newloc}(!u).\mathbf{out}(P)@u.\mathbf{in}(!X)@u.\mathbf{eval}(X)@l$, otherwise.

Since P is a closed term, i.e., P does not contain free variables, we can think of P as a closure $\langle process, data\rangle$. Thus, we have that processes migrate while taking their states with them.

Finally, if we want to download a program code P stored in a tuple with one field only (which contains P) from a (perhaps remote) location l, the COD paradigm is simply programmed by means of an instruction of the form $\mathbf{read}(!X)@l$.

In the next three subsections we discuss three specific examples that take advantage of the above described facilities.

5.1 Remote Procedure Call

A caller process, *caller*, sends a request to the callee, *callee*, and waits for a response. The request, together with the name of the procedure and its actual parameters, contains the *caller*'s private locality where the response has to be delivered.

$$\begin{aligned} caller = {} & \mathbf{newloc}(u).\,\mathbf{out}(procid, e_1, \ldots, e_n, u)@\ell_{callee}\,. \\ & \mathbf{in}(!y_1, \ldots, !y_k)@u.\left\langle next\ behavior\right\rangle. \end{aligned}$$

Process *callee* waits for an invocation, executes the related procedure and sends back the results using the locality, passed together with the service request.

$$\begin{aligned} callee = {} & \mathbf{in}(!pid, !x_1, \ldots, !x_n, !u)@\mathtt{self}.(callee\ | \\ & \left\langle pid(x_1, \ldots, x_n)\right\rangle.\mathbf{out}(r_1, \ldots, r_k)@u.\,\mathbf{nil}). \end{aligned}$$

When processes are allocated in a net, the local environment of *caller* assigns to the locality ℓ_{callee} the site where *callee* is located. Hence, we have:

$$N = s_1 ::_{\{s_1/\mathtt{self}, s_2/\ell_{callee}\}} caller \parallel s_2 ::_{\{s_2/\mathtt{self}\}} callee$$

A crucial role in this example is played by $\mathbf{newloc}(u)$ which permits a private data space to be created and accessed only via the variable u.

5.2 Dynamic Newsgatherer

We now consider *remote programming*. This programming discipline permits writing agents which can dynamically move over the network and can interact locally with other agents. An agent placed by a user at the server's location can thus be decoupled from the user and interact with the server without using the net.

Consider the following scenario. User P needs additional information on a piece of data represented by *item* (*item* could be, for example, the title of a book whose price P wants to know). Part of the behavior of P depends on this information. However, there are some activities which are independent of it. P can look for the required information in a database distributed over the network. We assume that at each node of the database reachable from ℓ_{item} contains either a tuple of the form $(item, v)$, with the desired information, or a tuple of the form $(item, \ell_{next})$, with the information about the next node to search for the additional information.

The user process P asks for the execution at ℓ_{item} (the starting point of the search, which can be chosen according to the search key *item*) of the agent *gatherer*, which dynamically travels between nodes looking for a tuple that contains information on *item*. This agent takes as its parameters the research key *item* and a fresh locality u, which provides the address of the user's private tuple space where the result of the search has to be placed. Once *gatherer* has been spawned, P splits its behavior into two parallel components: one waits for the additional information and the other proceeds. Thus, those activities that do not need the additional information are decoupled from the search activity, which might be complex and expensive.

$$P = \mathbf{newloc}(u).\mathbf{eval}(gatherer(item, u))@\ell_{item}.\,((\mathbf{in}(!x)@u.P_1)\mid P_2)$$

Process *gatherer* can match two alternative tuples. The first one captures the additional information on *item* (e.g., the price). If this is found then it is placed at locality u and *gatherer* terminates. The second tuple is used to obtain the address of the node where the search has to be repeated.

$$\begin{aligned} gatherer(item, u) = {} & \mathbf{read}(item, !x)@\mathtt{self}.\,\mathbf{out}(x)@u.\,\mathbf{nil} \\ & + \mathbf{read}(item, !u')@\mathtt{self}.\,\mathbf{eval}(gatherer(item, u))@u'.\,\mathbf{nil} \end{aligned}$$

Our assumption about the structure of the distributed database guarantees that *gatherer* never deadlocks (because either the associated information or a location where the search can be repeated certainly found), but it does not ensure that the search activity will terminate successfully: *gatherer* might loop indefinitely. This could happen if its second tuple, the one with location information, always finds a match in the tuple spaces.

5.3 An Electronic Marketplace

By means of an example borrowed from [41], we illustrate now how to use KLAIM to program mobile agents.

Assume that a client (process) P wants to buy a specific camera, c. To decide where to purchase the camera, P activates a migrating agent A and passes the following information to it:

1) c, the make and the model of the camera chosen,
2) loc_D, the locality of the directory of the electronic marketplace, and
3) a length measure, which will be used to identify the geographical area of interest.

P expects A to return the name, address and telephone number of the closest (within the chosen area) camera shop with the lowest price for c. The following could be part of the behavior of P:

$$P \overset{def}{=} \ldots\ \mathbf{eval}(A\langle c, loc_D, length\rangle)@\mathtt{self}.\mathbf{in}(c, !x, !y)@\mathtt{self}.\ldots$$

where x will retain the name, address and telephone number of the camera shop from where to buy c at cost y.

The agent A behaves as follows:

1) It obtains the site where P is located, which will be used both to return the outcome of the query and to identify the geographical area which is of interest for pricing information. This is done by putting a tuple containing `self` into a new tuple space u', in order to force the evaluation of `self` within the local tuple space, and by withdrawing the tuple.
2) It migrates to the site of the marketplace directory and asks for (and obtains) the list of all camera shops whose location is close to the site of P. Each item in the list contains the name, address and telephone number of a camera shop. A function l will return the locality information within an item.
3) It visits each camera shop in turn and obtains the local price for c. The agent retains information about the shop only if a lower price than that currently stored is offered.
4) After visiting all the camera shops on the list, it sends back to the site of P the information about the shop that offers the lowest price for c. It then terminates.

For the sake of simplicity, in defining agents we use a conditional construct (which can be programmed by exploiting the dynamic creation of new sites and the choice operator) and a data type *list* (with the usual operators *hd*, *tl*, and *empty*).

$$\begin{aligned} A(x, u, y) \overset{def}{=}\ & \mathbf{newloc}(u').\mathbf{out}(\mathtt{self})@u'.\mathbf{in}(!u'')@u'.\\ & \mathbf{eval}(B\langle x, u'', y\rangle)@u.\mathbf{nil} \end{aligned}$$

$$\begin{aligned} B(x, u, y) \overset{def}{=}\ & \mathbf{out}(cshop, u, y)@\mathtt{self}.\mathbf{in}(cshop, !list)@\mathtt{self}.\\ & \mathbf{if}\ empty(list)\ \mathbf{then}\ \mathbf{out}(x, nocloseshop, -1)@u.\mathbf{nil}\\ & \mathbf{else}\ I\langle c, list, u, l(hd(list))\rangle \end{aligned}$$

$$\begin{aligned} I(x, y, u, u''') \overset{def}{=}\ & \mathbf{eval}(\mathbf{read}(x, !cost)@\mathtt{self}.\\ & R\langle x, y, cost, hd(y), u\rangle)@u'''.\mathbf{nil} \end{aligned}$$

$$\begin{aligned} R(x, y, w, z, u) \overset{def}{=}\ & \mathbf{if}\ empty(y)\ \mathbf{then}\ \mathbf{out}(x, z, w)@u.\mathbf{nil}\\ & \mathbf{else}\ C\langle x, tl(y), w, z, u, l(hd(tl(u)))\rangle \end{aligned}$$

$$\begin{aligned} C(x, y, w, z, u, u''') \overset{def}{=}\ & \mathbf{eval}(\mathbf{read}(x, !cost)@\mathtt{self}.\mathbf{if}\ cost < w\\ & \mathbf{then}\ R\langle x, y, cost, hd(y), u\rangle\\ & \mathbf{else}\ R\langle x, y, w, z, u\rangle)@u'''.\mathbf{nil} \end{aligned}$$

The following will be part of the behavior of each camera shop S_i

$$S_i \overset{def}{=} \ldots \mid \mathbf{out}(c, price(c))@\mathtt{self}.\mathbf{nil} \mid \ldots$$

Let D denote the marketplace directory process. The net could be initially structured as follows:

$$s_P ::_{\{s_D / loc_D\}} P \parallel s_D ::_{\{s_1 / cs_1, \ldots, s_n / cs_n\}} D \parallel s_1 ::_{\{\}} S_1 \parallel \ldots \parallel s_n ::_{\{\}} S_n$$

If now we are interested in inferring the type δ of P, we have that:

$$\begin{aligned} \delta\ &: rec\ \nu.(\mathtt{self} \mapsto \langle\{o, i, e, n\}, \nu\rangle, u' \mapsto \langle\{o, i, e, n\}, \nu\rangle,\\ &\quad loc_D \mapsto \langle\{e\}, \delta'\rangle)\\ \delta'\ &: \mathtt{self} \mapsto \langle\{o, i\}, \phi\rangle, u'' \mapsto \langle\{o\}, \phi\rangle, u''' \mapsto \langle\{e\}, \delta''\rangle\\ \delta''\ &: rec\ \nu.(\mathtt{self} \mapsto \langle\{r\}, \phi\rangle, u'' \mapsto \langle\{o\}, \phi\rangle, u''' \mapsto \langle\{e\}, \nu\rangle) \end{aligned}$$

where *rec* is used to represent recursive types.

These types state that P performs any kind of operation both at the site where it is located (addressed by `self`) and at the site it dynamically creates (namely u'). Moreover, when (a process activated by) P migrates to the site of the marketplace directory (addressed by loc_D), it performs both local **out** and **in**, remote **out** at u'' (to return the outcome of the initial query), and migration to u''' (the site of a camera shop). Finally, when running at the site of a camera shop, (a process activated by) P performs local **read** (to read the local price for the camera c), remote **out** at the original site of P, and migrations to the sites of other camera shops.

6 KLAVA: KLAIM IN JAVA

In this section we describe the prototype implementation of KLAIM. To ensure portability over different platforms we choose Java [3] as the implementation language. Of course, here we assume a basic knowledge of Java.

The implementation of KLAIM in Java (JDK 1.1), called KLAVA [5], extends Java packages with two new packages, **`Linda`** and **`KLAIM`**.

The **`Linda`** package implements standard Linda primitives. The main classes of this package are **`Tuples`** and

```
public class NodeClassLoader extends ClassLoader {
    private Hashtable classes = new Hashtable();
    private Hashtable classData = new Hashtable();
    Node thisNode;

    public NodeClassLoader() {
    }

    synchronized public void addClassBytes( String className, byte classBytes[]) {
        if( classData.get( className ) == null && classBytes != null )
                    classData.put( className, classBytes );
        }
    :
    public synchronized Class loadClass(String className, boolean resolveIt)
        throws ClassNotFoundException {
        Class result;
        byte classData[];
        result = (Class)classes.get(className); /* Check local cache of classes */
        if (result != null) {
            return result;
        }
        classData = getClassBytes(className); /* Load the class from the repository */
        if (classData == null) {
            try {
                result = super.findSystemClass(className);
                return result;
            } catch (Exception e) {
                System.err.println(“NodeClassLoader : ” + e);
                e.printStackTrace();
                throw new ClassNotFoundException( className );
            }
        }
        result = defineClass(classData, 0, classData.length); /* Parse the class file */
        if (result == null) {
            throw new ClassFormatError();
        }
        if (resolveIt) {
            resolveClass(result);
        }
        classes.put(className, result);
        return result;
    }
}
```

Fig. 1. `NodeClassLoader.java`.

`TupleSpace`. The class `Tuples` provides the methods to build and handle tuples. The class `TupleSpace` provides the mechanisms to build, access and update a tuple space. In particular, the Linda operations **in**, **out**, and **read** are implemented as methods of this class.

The `KLAIM` package supports the implementation of KLAIM. The main classes of this package are `Net`, `Node`, `K-Process`, and `NodeMsg`.

The class `Net` implements KLAIM coordination language, i.e., a KLAIM net is an object of this class. A net object behaves like a server and contains the code to register the sites of a net. In the current implementation, localities are implemented as strings. Sites, on the other hand, are Internet addresses.

An object of the class `Node` implements a KLAIM node. Hence, it encapsulates a tuple space and a set of processes. KLAIM primitives (**in**, **read**, **out**, **eval**) are implemented as methods of this class. One of the parameters of these methods is the locality of the node.

A KLAIM process is an object of the class `K-Process`. The main method of this class is the method `execute()`. This method is invoked to run a process on a node, such as the method `run` of the class `Thread`.

The objects of the class `NodeMsg` are used to implement node communications. A message object contains the sender node, the receiver node, the operation code, and a content field of type `Object`. This feature permits transmission of processes. However, the receiver node may not know the class the process belongs to. Therefore, the process must be sent together with the corresponding `.class` file. Each node has also a specific `NodeClassLoader` which performs the dynamic linkage of the class received from other nodes of the net.

The main method of the class `NodeClassLoader` is `addClassBytes` which is invoked when a node receives a process from the net. The method `addClassBytes` inserts the `.class` files into a local hash table. The method `loadClass` uses the hash table to load the class definitions of remote processes before starting their execution. Note that a similar approach was adopted in the implementation of the AGLETS library [26]. Fig. 1 presents part of our Java code implementing the `NodeClassLoader`.

To give the reader a flavor of KLAVA programming, we report in Fig. 2 the source code of the CameraClient agent of the example presented in Section 5.3.

```
public class CameraClient extends K-Process {
       protected KString CameraMake;
       protected Locality MarketPlaceDir;
       protected KInteger distance;
       protected MarketPlaceAgent mAgent;

       public CameraClient( KString c, Locality m, KInteger d ) {
              CameraMake = c;
              MarketPlaceDir = m;
              distance = d;
       }

       public void execute() {
              PhysicalLocality newLoc;
              PhysicalLocality KLoc = new PhysicalLocality();
              KString ShopName = new KString();
              KInteger CameraPrice = new KInteger();
              newLoc = (PhysicalLocality)newloc();
              out( self, newLoc );
              in( KLoc, newLoc );
              mAgent = new MarketPlaceAgent( CameraMake, KLoc, distance );
              eval ( mAgent, MarketPlaceDir );
              in( CameraMake, ShopName, CameraPrice, self );
              Print( CameraMake + “ at “ + ShopName + “ costs “ + CameraPrice );
       }

       public static void main( String args[] ) throws IOException {
              Node node;
               PhysicalLocality ClientLoc = new PhysicalLocality( “CameraClient“ );
               KString CameraMake = new KString( “CameraX“ );
               Locality MarketLoc = new PhysicalLocality( “MarketPlace“ );
               KInteger distance = new KInteger( 10 );
               if ( args.length > 0 )
                   ClientLoc = new PhysicalLocality( args[0] );
               if ( args.length > 1 )
                   CameraMake = new KString( args[1] );
               if ( args.length > 2 )
                 MarketLoc = new PhysicalLocality( args[2] );
               if ( args.length > 3 )
                   distance = new KInteger(Integer.parseInt( args[3] ) );
               node = new NodeG( “CameraClient“, ClientLoc, “localhost“, 9999 );
               K-Process P = new CameraClient( CameraMake, MarketLoc, distance );
               node.start();
               node.addProcess( P );
       }
}
```

Fig. 2. `CameraClient.java`.

REMARK 6.1. Java has also been used to implement a dialect of Linda called Jada [12]. Jada supports a version of Linda with multiple tuple spaces. Tuple spaces are the key notion of Jada; they are autonomous entities, distributed over the nodes of a net and identified by the internet address of the nodes where they are placed. In Jada there is no distinction between logical and physical addresses. Processes use tuple spaces by connecting to the nodes where they are placed and by invoking their methods. Jada does not support process mobility, namely the **eval** primitive is not implemented and processes cannot be exchanged in communications.

7 CONCLUDING REMARKS

In this paper, we have presented a kernel programming language which supports mobile applications. An operational semantics, which focuses on the coordination of mobile agents, is provided. A type system that permits one to statically detect violations of security properties related to capabilities and access control has been developed. Programming examples have been presented that illustrate how mobile applications can be expressed in KLAIM. Finally, a prototype implementation in Java has been described.

The KLAIM type system provides a first step towards the ambitious goal of demonstrating that typing information can be systematically used to guarantee that well–typed processes enjoy security properties. We plan to extend the type system by introducing:

- user-defined capabilities,
- allowance capabilities (e.g., maximum life-time in seconds, maximum size in bytes, etc.),
- multilevel security (e.g., structuring localities into levels of security), and
- dynamic transmission of access rights.

KLAIM can also be equipped with cryptographic primitives as done in spi-calculus [1].

We plan to develop observational semantics as a foundation for programming logics and verification techniques. To this end, our starting point will be the testing framework developed for a process calculus based on Linda in [18], [37].

We are currently exploring the possibility of allowing nets to communicate and move processes and tuples between them. The current KLAVA implementation appears to be well-suited also to program this feature, that will lead to providing KLAIM and KLAVA with hierarchical nets.

KLAIM has been implemented via Java packages, hence programmers have to adopt the Java (object-oriented) programming discipline to use KLAIM. A compiler from KLAIM extended with Pascal-like primitives in KLAVA is under development, together with the implementation of the typed version of KLAVA.

ACKNOWLEDGMENTS

We are grateful to Luca Cardelli and Betti Venneri for stimulating discussions about global programming and type systems, and to the anonymous referees, whose useful comments helped us to improve the paper. We also thank Lorenzo Bettini and Emilio Tuosto for discussions about the implementation of KLAIM. This work has been partially supported by Esprit Working Groups *CONFER2* and *COORDINA*, HCM project EXPRESS, by CNR Progetti Speciali *Modelli e Metodi per la Matematica e l'Ingegneria*, and by *Metodologie e Strumenti di Analisi, Verifica e Validazione di Sistemi Software Affidabili*.

REFERENCES

[1] M. Abadi and A. Gordon, "A Calculus for Cryptographic Protocols: the Spi-Calculus," *Proc. Fourth ACM Conf. Computer and Comm. Security*, 1997.

[2] R. Amadio and S. Prasad, "Localities and Failures," *FCT&TCS 14, Proc.*, P.S. Thiagarajan, ed., *Lecture Notes in Computer Science 880*, pp. 205-216. Springer-Verlag, 1994.

[3] K. Arnold and J. Gosling, *The Java Programming Language*. Addison-Wesley, 1996.

[4] G. Berry and G. Boudol, "The Chemical Abstract Machine," *Theoretical Computer Science*, vol. 96, pp. 217-248, 1992.

[5] L. Bettini, "Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile," in Italian, Tesi di Laurea, Dipartimento di Sistemi e Informatica, Università di Firenze, 1998.

[6] G. Boudol, I. Castellani, M. Hennessy, and A. Kiehn, "Observing Localities," *Theoretical Computer Science*, vol. 114, 1993.

[7] L. Cardelli, "A Language with Distributed Scope," *Computing Systems*, vol. 8, no. 1, pp. 27-59. MIT Press, 1995.

[8] L. Cardelli, "Global Computation Manuscript," 1996. Available at URL http://www.luca.demon.co.uk.

[9] L. Cardelli and A. Gordon, "Mobile Ambients," FoSSaCS'98, M. Nivat, ed., *Lecture Notes in Computer Science 1378*, pp. 140-145. Springer-Verlag, 1998.

[10] N. Carriero and D. Gelernter, "Linda in Context," *Comm. ACM*, vol. 32, no. 4, pp. 444-458, 1989.

[11] N. Carriero, D. Gelernter, and J. Leichter, "Distributed Data Structures in Linda," *Proc. ACM Symp. Principles of Programming Languages*, New York: ACM, pp. 236-242, 1986.

[12] P. Ciancarini and D. Rossi, "Jada: Coordination and Communication for Java agents," *Mobile Object Systems: Towards the Programmable Internet*," J. Vitek and C. Tschudin, eds., *Lecture Notes In Computer Science 1222*, pp. 213-228. Springer-Verlag, 1997.

[13] F. Corradini and R. De Nicola, "Locality Based Semantics for Process Algebras," *Acta Informatica*, vol. 34, pp. 291-324, 1997.

[14] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna, "Analyzing Mobile Code Languages," J. Vitek and C. Tschudin, eds., *Mobile Object Systems, Lecture Notes in Computer Science 1222*. Springer-Verlag, 1997.

[15] R. De Nicola, G. Ferrari, and R. Pugliese, "Locality Based Linda: Programming with Explicit Localities," *TAPSOFT '97, Proc.*, M. Bidoit and M. Dauchet, eds., *Lecture Notes in Computer Science 1214*, pp. 712-726. Springer-Verlag, 1997.

[16] R. De Nicola, G. Ferrari, and R. Pugliese, "Coordinating Mobile Agents via Blackboards and Access Rights," *Coordination '97, Proc.*, D. Garlan and D. Le Metayer, eds., *Lecture Notes in Computer Science 1282*, pp. 220-237, Springer-Verlag, 1997.

[17] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri, "Types for Access Control," submitted for publication, 1998. URL http://di.unipi.it/~giangi/papers

[18] R. De Nicola and R. Pugliese, "A Process Algebra Based on Linda. *Coordination '96, Proc.*, P. Ciancarini and C. Hankin, eds., *Lecture Notes in Computer Science 1061*, pp. 160-178. Springer-Verlag, 1996.

[19] C. Fournet, G. Gonthier, J.-L. Lévy, L. Maranget, and D. Rémy, "A Calculus of Mobile Agents," *Concur '96, Proc.*, U. Montanari and V. Sassone, eds., *Lecture Notes In Computer Science 1119*, pp. 406-421. Springer-Verlag, 1996.

[20] D. Gelernter, "Generative Communication in Linda," *ACM Trans. on Programming Languages and Systems*, vol. 7, no. 1, pp. 80-112, 1985.

[21] D. Gelernter, "Multiple Tuple Spaces in Linda," PARLE'89, *Proc.*, G. Goos and J. Hartmanis, eds., *Lecture Notes in Computer Science 365*, pp. 20-27, 1989.

[22] D. Gelernter, N. Carriero, and S. Chandran, et al., "Parallel Programming in Linda," *Proc. Int'l Conf. Parallel Programming*, IEEE, pp. 255-263, 1985.

[23] A. Giacalone, P. Mishra, and S. Prasad, "Facile: A Symmetric Integration of Concurrent and Functional Programming," *Int'l J. Parallel Programming*, vol. 18, no. 2, 1989.

[24] M. Hennessy and H. Lin, "Symbolic Bisimulations," Theoretical Computer Science, vol. 138, pp. 353-389, 1995.

[25] C.A.R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985.

[26] IBM Aglets Workbench—Home Page, 1996. URL address: http://www.trl.ibm.co.jp/ aglets/

[27] A. Ingolfsdottir, "Semantic Models for Communicating Processes with Value–Passing," PhD thesis, Univ. of Edinburgh, 1994.

[28] N. Kobayashi, B. Pierce, and D. Turner, "Linearity and the π-Calculus," *Proc. POPL'96*, 1996.

[29] J. Leitcher, Shared Memories, Buses and LANs—Linda Implementations Across the Spectrum of Connectivity," Research Report YALEU/DCS/TR-714, Dept. of Computer Science, Yale Univ., 1989.

[30] R. Milner, *Communication and Concurrency*. Prentice Hall International, 1989.

[31] R. Milner, "The Polyadic π-Calculus: A Tutorial," Technical Report, ECS-LFCS-91-180, 1991.

[32] R. Milner, J. Parrow, and D. Walker, "A Calculus of Mobile Processes, Part I and II, *Information and Computation*, vol. 100, pp. 1-77, 1992.

[33] B. Pierce and D. Turner, "Concurrent Objects in a Process Calculus," *Theory and Practice of Parallel Programming*, T. Ito and A. Yonezawa, eds., *Lecture Notes in Computer Science 907*, pp. 186-215, 1994.

[34] B. Pierce and D. Sangiorgi, "Typing and Subtyping for Mobile Processes," *Proc. LICS '93*, IEEE Press, 1993. A full version appears in *Mathematical Struction in Computer Science*.

[35] G.D. Plotkin, "A Structural Approach to Operational Semantics," Technical Report DAIMI FN-19, Dept. of Computer Science, Aarhus Univ., 1981.

[36] G.D. Plotkin, "Lectures Notes in Domain Theory," Univ. of Edinburgh, 1983.

[37] R. Pugliese, "Semantic Theories for Asynchronous Languages," PhD thesis VIII-96-6, Dip. Scienze dell'Informazione, Univ. di Roma 'La Sapienza', 1996.

[38] J. Reppy, "Higher Order Concurrency," PhD thesis, Cornell Univ., Tr-92-1285, 1992.

[39] B. Thomsen, L. Leth, and A. Giacalone, "Some Issues in the Semantics of Facile Distributed Programming," *Proc. REX Workshop Semantics: Foundations and Applications*, J.W. de Bakker, W-P. de Roever, and G. Rezenberg, eds., *Lecture Notes in Computer Science 666*, pp. 563-593. Springer-Verlag, 1992.

[40] D. Volpano and G. Smith, "A Typed-Based Approach to Program Security," *Proc. TAPSOFT '97, Lecture Notes in Computer Science 1214*, pp. 607-621. Springer-Verlag, 1997.

[41] J.E. White, "Mobile Agents," *Software Agents*, J.M. Bradshaw, ed., pp. 437-471, 1996.

Rocco De Nicola received the Laurea degree in computer science from the University of Pisa, Italy, in 1978 and the PhD degree in computer science from the University of Edinburgh, United Kingdom, in 1985. He is a full professor in the Dipartimento di Sistemi e Informatica at the University of Florence, Italy, since 1995. From 1990 until 1995, he has been a full professor in the Dipartimento di Scienze dell'Informazione, at the University of Rome 'La Sapienza'. Italy. Before that he was a full time research at IEI–CNR, Pisa and worked at Edinburgh University and for Italtel in Milano. Dr. De Nicola is a member of the IFIP W.G. 2.2. His research interests are centered around design methods and formal specifications and their use for the verification of distributed concurrent systems.

Gian Luigi Ferrari received the Laurea degree in computer science, and the PhD degree in computer science from the University of Pisa, Italy, in 1984 and 1989, respectively. He is an assistant professor of computer science in the Department of Computer Science at the University of Pisa. His research interests include formal specification and verification of concurrent and mobile systems, programming languages for network computing, tool support for mobile systems, and theoretical aspects of distributed computing.

Rosario Pugliese received the Laurea degree in computer science from the University of Pisa, Italy, in 1991, and the PhD degree in computer science from the University of Rome 'La Sapienza' Ital, in 1996. He currently holds a postdoctoral position in the Dipartimento di Sistemi e Informatica, at the University of Florence, Italy. His research interests include semantics of concurrent programming and specification languages, and models and tools supporting specification and verification of concurrent and distributed systems.