O'KLAIM: a coordination language with mobile mixins*

Lorenzo Bettini[1] Viviana Bono[2] Betti Venneri[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze,
{bettini,venneri}@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Torino, bono@di.unito.it

Abstract. This paper presents O'KLAIM (Object-Oriented KLAIM), a linguistic extension of the higher-order calculus for mobile processes KLAIM with object-oriented features. Processes interact by an asynchronous communication model: they can distribute and retrieve resources, sometimes structured as incomplete classes, i.e., mixins, to and from distributed tuple spaces. This mechanism is coordinated by providing a subtyping relation on classes and mixins, which become polymorphic items during communication. We propose a static typing system for: (*i*) checking locally each process in its own locality; (*ii*) decorating object-oriented code that is sent to remote sites with its type. This way, tuples can be dynamically retrieved only if they match by subtyping with the expected type. If this pattern matching succeeds, the retrieved code can be composed with local code, dynamically and automatically, in a type-safe way. Thus a global safety condition is guaranteed without requiring any additional information on the local reconfiguration of local and foreign code, and, in particular, without any further type checking. Finally, we present main issues concerning the implementation of O'KLAIM.

1 Introduction

Mixins [14, 22, 1] are (sub)class definitions parameterized over a superclass and were introduced as an alternative to standard class inheritance. A mixin could be seen as a function that, given one class as an argument, produces another class, by adding or overriding specific sets of methods. The same mixin can be used to produce a variety of classes with the same functionality and behavior, since they all have the same sets of methods added and/or redefined. The superclass definition is not needed at the time of writing the mixin definition, thus improving modularity. The uniform extension and modification of classes is instead absent from the classical class-based languages.

Due to their dynamic nature, mixin inheritance can be fruitfully used in a *mobile code* setting [28, 17]. In [8], we introduced MOMI (Mobile Mixins), a coordination model for mobile processes that exchange object-oriented code. The underlying idea

* This work has been partially supported by EU within the FET - Global Computing initiative, project MIKADO IST-2001-32222, DART project IST-2001-33477 and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

motivating MOMI is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not appear to scale well to a distributed context with mobility. MOMI's approach consists in structuring mobile object-oriented code by using mixin-based inheritance, and this is shown to fit into the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that must access some files, can be completed with a base class in order to provide access methods that are specific of the local file system. Conversely, critical operations of a mobile agent enclosed in a downloaded class can be redefined by applying a local mixin to it (e.g., in order to restrict the access to sensible resources, as in a *sand-box*). Therefore, MOMI is a combination of a core coordination calculus and an object-oriented mixin-based calculus.

MOMI highly relies on typing. The most important feature of MOMI's typing is the *subtyping* relation that guarantees safe, yet flexible, code communication. We assume that the code that is sent around has been successfully compiled in its own site (independently from the other sites), and it travels together with its static type. When the code is received on a site (whose code has been successfully compiled, too), it is accepted only if its type is compliant with respect to the one expected, where compliance is based on subtyping. Thus, dynamic type checking is performed only at communication time. This is a crucial matter for mobility, since mobile code and in particular mobile agents are expected to be autonomous: once the communication successfully occurred, transmitted code behaves remotely in a type safe way (no run-time errors due to type violations). This makes the code exchange an *atomic* action.

This paper presents the experimental language O'KLAIM that is obtained by applying MOMI's approach [8] to the language KLAIM [18, 4], which is specifically designed to program distributed systems where mobile components interact through multiple distributed tuple spaces and mobile code. A preliminary design that led to O'KLAIM was introduced in [7]. KLAIM offers a much more sophisticated, complete, and effective coordination mechanism of mobile processes than the toy language of MOMI, where the focus was mainly on the subtyping relation on classes and mixins. O'KLAIM integrates the mixin-based object-oriented features into a mobile code calculus with an asynchronous coordination mechanism. To this aim, the notion of "tuple" is extended to include object-oriented code, therefore KLAIM processes can retrieve from and insert into tuple spaces object-oriented components (in particular, classes, mixins and objects) as well as standard KLAIM processes. A type system is designed for checking statically the extended notion of processes, so that compiled processes contain some static type information which is used dynamically. Therefore, the tuples that are added to a tuple space are decorated with their type, and a process willing to retrieve a tuple from a tuple space will employ an extended pattern matching mechanism that uses also this tuple type information. This matching essentially consists in checking subtyping on object-oriented components. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors) without requiring any further type checking of the whole code. Type safety of the communication results from the static type soundness of local and foreign code and a (global) subject reduction property. In particular, we show that the subject reduction property is based on a crucial property of substitutivity by subtyping. The underlying substitution operation requires specific

methods renaming, in order to avoid name collision problems that arise when classes and mixins are used as first-class data in a mobile code setting where matching relies on subtyping. These new metatheoretical results about the precise concept of substitution to be used extend and improve the results presented for MOMI in [8].

Summarizing, O'KLAIM aims at two complementary goals. Firstly, subtyping on classes and mixins (as designed for MOMI) is successfully experimented as a tractable mechanism to coordinate mobile code exchange within a process calculus with a more sophisticated communication mechanism. Secondly, the language KLAIM is enriched with object-oriented code. This casts some light on how the same approach can be fruitfully used for extending other mobile code languages with safe object-oriented code exchange. Finally, the implementation of O'KLAIM is presented. This consists in a Java package, `momi`, providing the run-time systems for classes and mixins that can be dynamically manipulated and composed. The programming language X-KLAIM (that implements the basic concepts of KLAIM) has been extended in order to be compiled into Java code exploiting the `momi` package.

2 O'KLAIM: an object-oriented KLAIM

O'KLAIM is a linguistic integration of KLAIM with object-oriented features, following the design of MOMI [8]. The coordination part and the object-oriented part are orthogonal, so that, in principle, such an integration would work for any extension/restriction of KLAIM (as discussed in [4]) and also for other calculi for mobility and distribution, such as *DJoin* [23]. We first recall the main features of KLAIM and MOMI and then we show how they are integrated in order to build O'KLAIM.

2.1 The basics of KLAIM

KLAIM (*Kernel Language for Agent Interaction and Mobility*) [18, 4] is a coordination language inspired by the Linda model [24], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are sequences of information items (called *fields*). There are two kinds of fields: *actual fields* (i.e., expressions, processes, localities, constants, identifiers) and *formal fields* (i.e., variables). Syntactically, a formal field is denoted with !*ide*, where *ide* is an identifier. Tuples are anonymous and content-addressable; *pattern-matching* is used to select tuples in a tuple space:

- two tuples match if they have the same number of fields and corresponding fields have matching values or formals;
- formal fields match any value of the same type, but two formals never match, and two actual fields match only if they are identical.

For instance, tuple (`"foo"`, `"bar"`, 300) matches with (`"foo"`, `"bar"`, !*val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *val* (an integer variable) will contain the value 300.

Tuple spaces are placed on *nodes* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. The distinction between logical and physical locality (and thus the concept of "allocation environment"), and the creation of new nodes and process definitions are not

relevant in the O'KLAIM context, thus, for the sake of simplicity, we omit them in the present formal presentation. Notice, however, that their integration, being orthogonal, is completely smooth.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can perform four basic operations over nodes. The $\mathbf{in}(t)@\ell$ operation looks for tuple t' that matches with t in the tuple space located at ℓ. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. The $\mathbf{read}(t)@\ell$ operation differs from $\mathbf{in}(t)@\ell$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at ℓ. The $\mathbf{out}(t)@\ell$ operation adds the tuple t to the tuple space located at ℓ. The $\mathbf{eval}(P)@\ell$ operation spawns process P for execution at node ℓ.

KLAIM is higher-order in that processes can be exchanged as primary class data. While $\mathbf{eval}(P)@\ell$ spawns a process for (remote) evaluation at ℓ, processes sent with an **out** must be retrieved explicitly at the destination site. The receiver can then execute the received process locally, as in the following process: $\mathbf{in}(!X)@\texttt{self}.\mathbf{eval}(X)@\texttt{self}$.

2.2 MOMI and O'KLAIM

MOMI was introduced in [8], where mixin inheritance is shown to be more flexible than standard class inheritance to fit into the dynamic nature of a mobile code scenario. The key rôle in MOMI's typing is played by a *subtyping* relation that guarantees safe, yet flexible and scalable, code communication. MOMI's subtyping involves not only object subtyping, but also a form of class subtyping and mixin subtyping: therefore, subtyping hierarchies are provided along with the inheritance hierarchies. It is important to notice that we are not violating the design rule of keeping inheritance and subtyping separated, since mixin and class subtyping plays a pivotal role only during the communication, when classes and mixins become genuine run-time polymorphic values.

In synthesis, MOMI consists of:

1. the definition of an object-oriented "surface calculus" containing essential features that an object-oriented language must have to write mixin-based code;
2. the definition of a new subtyping relation on class and mixin types to be exploited dynamically at communication time;
3. a very primitive coordination language consisting in a synchronous send/receive mechanism, to study the communication of the mixin-based code among different site.

O'KLAIM integrates the object-oriented component of MOMI, in particular the subtyping relation on classes and mixins (both described in the next section), within KLAIM, which offers a much more sophisticated, complete, and effective coordination mechanism than the toy one of MOMI.

2.3 O'KLAIM: object-oriented expressions

In this section we present the object-oriented part of O'KLAIM, which is defined as a class-based object-oriented language supporting mixin-based class hierarchies via

exp ::=	v	(value)
	new exp	(object creation)
	$exp \Leftarrow m$	(method call)
	$exp_1 \diamond exp_2$	(mixin appl.)
v ::=	$\{m_i : \tau_{m_i} = b_i \ ^{i\in I}\}$	(record)
	x	(variable)
	class $[m_i : \tau_{m_i} = b_i \ ^{i\in I}]$ end	(class def)
	mixin expect$[m_i : \tau_{m_i} \ ^{i\in I}]$ redef$[m_k : \tau_{m_k}$ with $b_k \ ^{k\in K}]$ def$[m_j : \tau_{m_j} = b_j \ ^{j\in J}]$ end	(mixin def)

Table 1. Syntax of object-oriented terms.

mixin definition and *mixin application* (see Table 1). It is important to notice that specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax of the kernel object-oriented calculus.

Object-oriented expressions offer object instantiation, method call and mixin application; $\diamond$ denotes the mixin application operator. An object-oriented value, to which an expression reduces, is either an object, which is a (recursive) record $\{m_i : \tau_{m_i} = b_i \ ^{i\in I}\}$, or a class definition, or a mixin definition, where $[m_i : \tau_{m_i} = b_i \ ^{i\in I}]$ denotes a sequence of method definitions, $[m_k : \tau_{m_k}$ with $b_k \ ^{k\in K}]$ denotes a sequence of method re-definitions, and I, J and K are sets of indexes. Method bodies, denoted here with b (possibly with subscripts), are closed terms/programs and we ignore their actual structure. A mixin can be seen as an abstract class that is parameterized over a (super)class. Let us describe informally the mixin use through a tutorial example:

```
M = mixin                                C = class
  expect [n : τ]                           [n = ...
  redef [m2 : τ2 with ... next ...]        m2 = ...]
  def [m1 : τ1 = ...n()...]              end
end                                                    (new (M ◇ C))⇐ m1()
```

Each mixin consists of three parts:

1. methods *defined* in the mixins, like m_1;
2. *expected methods*, like n, that must be provided by the superclass;
3. *redefined methods*, like m_2, where *next* can be used to access the implementation of m_2 in the superclass.

The application $M \diamond C$ constructs a class, which is a subclass of C.

The typing for the object-oriented code refines essentially the typing rules sketched in [8]. The set $\mathcal{T}$ of types is defined as follows.

$$\tau ::= \Sigma \mid \iota \mid \tau_1 \to \tau_2 \mid \mathsf{class}\langle\Sigma\rangle \mid \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \qquad \Sigma ::= \{m_i : \tau_{m_i} \ ^{i\in I}\}$$

ι is a basic type and $\to$ is the functional type operator. Σ (possibly with a subscript) denotes a record type of the form $\{m_i : \tau_{m_i} \ ^{i\in I}\}$. if $m_i : \tau_{m_i} \in \Sigma$ we say that the *subject* m_i

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\ (proj) \qquad \frac{\Gamma \Vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (rec)$$

$$\frac{\Gamma \vdash \{m_i:\tau_{m_i} = b_i\ ^{i\in I}\} : \{m_i:\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \mathsf{class}\ [m_i:\tau_{m_i} = b_i\ ^{i\in I}]\ \mathsf{end} : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}\ (class)$$

$$\frac{\begin{array}{c}\Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k} \vdash \{m_j:\tau_{m_j} = b_j\ ^{j\in J}\} : \{m_j:\tau_{m_j}\ ^{j\in J}\} \\ \Gamma, \bigcup_{i\in I} m_i:\tau_{m_i}, \bigcup_{k\in K} m_k:\tau_{m_k}, \bigcup_{j\in J} m_j:\tau_{m_j}, next:\tau_{m_r} \Vdash b_r:\tau'_{m_r} \quad \tau'_{m_r} <: \tau_{m_r} \quad \forall r\in K \\ Subj(\Sigma_{new}) \cap Subj(\Sigma_{exp}) = \emptyset \quad Subj(\Sigma_{new}) \cap Subj(\Sigma_{red}) = \emptyset \\ Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset\end{array}}{\begin{array}{l}\Gamma \vdash \mathsf{mixin} \\ \quad\ \mathsf{expect}[m_i:\tau_{m_i}\ ^{i\in I}] \\ \quad\ \mathsf{redef}[m_k:\tau_{m_k}\ \mathsf{with}\ b_k\ ^{k\in K}]\ :\ \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \quad\ \mathsf{def}[m_j:\tau_{m_j} = b_j\ ^{j\in J}] \\ \quad\ \mathsf{end}\end{array}}\ (mixin)$$

where $\Sigma_{new} = \{m_j:\tau_{m_j}\ ^{j\in J}\}, \Sigma_{red} = \{m_k:\tau_{m_k}\ ^{k\in K}\}, \Sigma_{exp} = \{m_i:\tau_{m_i}\ ^{i\in I}\}$

Table 2. Typing rules for object-oriented values

$$\frac{\Gamma \vdash exp : \{m_i:\tau_{m_i}\ ^{i\in I}\} \qquad j\in I}{\Gamma \vdash exp \Leftarrow m_j : \tau_{m_j}}\ (lookup) \qquad \frac{\Gamma \vdash exp : \mathsf{class}\langle\{m_i:\tau_{m_i}\ ^{i\in I}\}\rangle}{\Gamma \vdash \mathsf{new}\ exp : \{m_i:\tau_{m_i}\ ^{i\in I}\}}\ (new)$$

$$\frac{\begin{array}{l}\Gamma \vdash exp_1 : \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \Gamma \vdash exp_2 : \mathsf{class}\langle\Sigma_b\rangle \\ \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red}) \\ Meth(\Sigma_b) \cap Meth(\Sigma_{new}) = \emptyset\end{array}}{\Gamma \vdash exp_1 \diamond exp_2 : \mathsf{class}\langle\Sigma_b \cup \Sigma_{new}\rangle}\ (mixin\ app)$$

Table 3. Typing rules for object-oriented expressions.

occurs in Σ (with type τ_{m_i}). $Subj(\Sigma)$ is the set of the *subjects* of Σ and $Meth(\Sigma)$ is the set of all the method names occurring in Σ (e.g., if $\Sigma = \{m:\{n:\tau\}\}$ then $Subj(\Sigma) = \{m\}$ while $Meth(\Sigma) = \{m,n\}$). As we left method bodies unspecified (see above), we must assume that there is an underlying system to type method bodies and records. We will denote this typing with $\Vdash$. Rules for $\Vdash$ are obviously not specified, but $\Vdash$-statements are used as assumptions in other typing rules. The typing rules for values are in Table 2.

Mixin types, in particular, encode the following information:

1. record types Σ_{new} and Σ_{red} contain the types of the mixin methods (new and redefined, respectively);
2. record type Σ_{exp} contains the *expected* types, i.e., the types of the methods expected to be supported by the superclass;
3. well typed mixins are well formed, in the sense that name clashes among the different families of methods are absent (the last three clauses of the (*mixin*) rule).

The typing rules for expressions are in Table 3.

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle}\ (\sqsubseteq \mathit{class})$$

$$\frac{\Sigma'_{new} <: \Sigma_{new} \qquad \Sigma_{exp} <: \Sigma'_{exp} \qquad \Sigma'_{red} = \Sigma_{red}}{\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle}\ (\sqsubseteq \mathit{mixin})$$

Table 4. Subtype on class and mixin types.

Rule (*mixin app*) relies strongly on a subtyping relation $<:$. The subtyping relation rules depend obviously on the nature of the object-oriented language we choose, but an essential constraint is that it must contain the *width subtyping* rule for record types: $\Sigma_2 \subseteq \Sigma_1 \Rightarrow \Sigma_1 <: \Sigma_2$.

We consider $m : \tau_1$ and $m : \tau_2$ ($\tau_1 \not\equiv \tau_2$) as distinct elements, and $\Sigma_1 \cup \Sigma_2$ is the standard record union. Σ_1 and Σ_2 are considered *equivalent*, denoted by $\Sigma_1 = \Sigma_2$, if they differ only for the order of their pairs $m_i : \tau_{m_i}$.

In the rule (*mixin app*), Σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. The premises of the rule (*mixin app*) are as follows:

i) $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$ requires that the superclass provides all the methods that the mixin expects and redefines;
ii) $\mathit{Meth}(\Sigma_b) \cap \mathit{Meth}(\Sigma_{new}) = \emptyset$ guarantees that name clashes cannot occur during the mixin application.

Notice that the superclass may have more methods than those required by the mixin constraints. Thus, the type of the mixin application expression is a class type containing both the signatures of all the methods supplied by the superclass (Σ_b) and those of the new methods defined by the mixin (Σ_{new}).

The key point is the introduction of a novel subtyping relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtyping relation is used to match dynamically the actual parameter's types against the formal parameter's types during communication. The part of the operational semantics of O'KLAIM, which describes communication formally, is presented in Section 2.6. The subtyping relation $\sqsubseteq$ is defined in Table 4. Rule ($\sqsubseteq$ *class*) is naturally induced by the (width) subtyping on record types, while rule ($\sqsubseteq$ *mixin*): permits the subtype to define more 'new' methods; prohibits to override more methods; and enables a subtype to require less expected methods.

2.4 O'KLAIM: processes and nets

O'KLAIM syntax is defined in Table 5. In order to obtain O'KLAIM, we extend the KLAIM syntax of tuples t to include any object-oriented value v (defined in Table 1). In particular, differently from KLAIM, formal fields are now explicitly typed. Actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ can be used to move object-oriented code (together with the other KLAIM items) from/to a locality ℓ, respectively. Moreover, we add to KLAIM processes the construct $\mathsf{let}\ x = exp\ \mathsf{in}\ P$ in order to pass to the sub-process P the result of computing an object-oriented expression exp (for exp syntax see Table 1). We use the following syntactic convention: x, X and χ are variables representing

$$
\begin{array}{lll}
P ::= \mathbf{nil} & \text{(null process)} & N ::= l :: p \quad \text{(single node)} \\
\quad \mid act.P & \text{(action prefixing)} & \quad \mid N_1 \parallel N_2 \quad \text{(net composition)} \\
\quad \mid P_1 \mid P_2 & \text{(parallel composition)} & p ::= P \mid \langle t \rangle \mid p_1 \mid p_2 \quad \text{(located item)} \\
\quad \mid X & \text{(process variable)} & \\
\quad \mid \mathsf{let}\ x = exp\ \mathsf{in}\ P & \text{(OO expression)} &
\end{array}
$$

$$
act ::= \mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \qquad \ell ::= l \mid \chi \qquad t ::= f \mid f,t
$$

$$
f ::= arg \mid !id : \sigma \qquad id ::= x \mid X \mid \chi \qquad arg ::= id \mid e \mid P \mid l \mid v
$$

Table 5. O'KLAIM syntax (see Table 1 for the syntax of *exp* and *v*, and Section 2.5 for types σ).

$$
\frac{}{\Gamma, id : \sigma \vdash id : \sigma}\ (proj) \qquad \frac{}{\Gamma \vdash l : \mathsf{loc}}\ (loc) \qquad \frac{}{\Gamma \vdash \mathbf{nil} : \mathsf{proc}}\ (nil)
$$

$$
a \equiv \mathbf{in}, \mathbf{read}, \mathbf{out} \qquad \frac{\begin{array}{l}\Gamma \vdash \ell : \mathsf{loc} \\ \Gamma \vdash f_i : \sigma_i \qquad i = 1, \ldots, n \ \wedge\ f_i \equiv arg \\ \Gamma \cup ftypes(f_1, \ldots, f_n) \vdash P : \mathsf{proc}\end{array}}{\Gamma \vdash a(f_1, \ldots, f_n)@\ell.P : \mathsf{proc}}\ (action)
$$

$$
ftypes(f,t) = \begin{cases} \{id : \sigma\} \cup ftypes(t) & \text{if } f \equiv !id : \sigma \\ ftypes(t) & \text{otherwise} \end{cases}
$$

$$
\frac{\Gamma \vdash Q : \mathsf{proc} \qquad \Gamma \vdash \ell : \mathsf{loc}}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P : \mathsf{proc}}\ (eval)
$$

$$
\frac{\Gamma \vdash P_1 : \mathsf{proc} \qquad \Gamma \vdash P_2 : \mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2) : \mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp : \tau \qquad \Gamma, x : \tau \vdash P : \mathsf{proc}}{\Gamma \vdash \mathsf{let}\ x = exp\ \mathsf{in}\ P : \mathsf{proc}}\ (let)
$$

Table 6. Typing rules for processes

object-oriented values, processes and localities, respectively. A constant locality (e.g., IP:port) is denoted by *l*. Moreover, *e* is a basic expression (i.e., not object-oriented).

A *Net* is a finite collection of *nodes*. A node is a pair where the first component is a (constant) locality and the second component is either a process *P* or a tuple $\langle t \rangle$ or a composition of processes and tuples. Thus, a tuple space is represented by the parallel composition of located tuples. Notice that programmers write only located processes, while located tuples are produced at run-time by evaluating **out** actions (see Table 8).

2.5 Typing for O'KLAIM

In order to type processes and nets, we extend the set of types $\mathcal{T}$ to $\mathcal{T}^* = \mathcal{T} \cup \{\mathsf{proc}, \mathsf{loc}\}$. σ will range over $\mathcal{T}^*$; in particular, loc is used to type localities and proc for well-typed processes. Typing rules for processes are defined in Table 6. O'KLAIM type system is not concerned with access rights and capabilities, as it is instead the type system for KLAIM presented in [19]. In the O'KLAIM setting, types serve the purpose of avoiding the "message-not-understood" error when merging local and foreign object-oriented code in a site. Thus, we are not interested in typing actions inside processes:

from our perspective, an O'KLAIM process is well typed when it has type proc, which only means that the object-oriented code that the process may contain is well typed.

O'KLAIM requires that every process is statically type-checked separately on its site and annotated with its type. The annotation process, not formally presented here, can be performed by the compiler during type checking: namely, every tuple item t_i that takes part in the information exchange (which may be an object-oriented value) must be decorated with its type information, denoted by $t_i{}^{\sigma_i}$. The types of the tuples are built statically by the compiler, while the types of tuple formal fields must be written explicitly by the programmer. In a process of the form $\mathbf{in}(!\,id : \sigma)@\ell.P$, the type σ is used to statically type check the continuation P, where *id* is possibly used. More generally, concerning (*action*) rule, in a process performing an operation with a tuple (i.e., **out**, **read** and **in**), the actual fields of the tuple are type checked, and the types of formal fields (collected by the function *ftypes*) are used to type check the continuation.

We observe that the typing rules for object-oriented expressions are syntax-driven and do not contain an explicit subsumption rule. Thus, they define an algorithm to assign a principal type to each expression, in a given environment Γ. Analogously, both subtyping and typing rules for processes are in an algorithmic shape.

2.6 Operational semantics for O'KLAIM

The operational semantics of O'KLAIM involves two sets of rules. The first set of rules describes how object-oriented expressions reduce to values and is denoted by $\twoheadrightarrow$. We omit here most of the rules because they are quite standard; they can be found in [6, 3]. However, we want to discuss the rule concerning mixin application, that produces a new class containing all the methods which are added and redefined by the mixin and those defined by the superclass. The rule (*mixinapp*) is presented in Table 7. The function *override*, defined below and used by rule (*mixinapp*), takes care of introducing in the new class the overridden methods. In particular, in the body of a mixin's overridden method m_i, the reserved variable *next* can be used to denote m_i's implementation in the superclass: this "old" implementation is given a fresh name $m_{i'}$. Dynamic binding is then implemented for redefined methods, and old implementations from the super class are basically hidden in the derived class, since they are given a fresh name.

Definition 1. *Given two method sets, ρ_1 and ρ_2, the result of override(ρ_1, ρ_2) is the method set ρ_3 defined as follows:*

- *for all $m_i : \tau_{m_i} = b_i \in \rho_2$ such that $m_i \neq m_j$ for all $m_j : \tau_{m_j} = b_j \in \rho_1$, then $m_i : \tau_{m_i} = b_i \in \rho_3$;*
- *for all $m_i : \tau_{m_i} = b_i \in \rho_1$ such that $m_i : \tau_{m_i} = b'_i \in \rho_2$, let $m_{i'}$ be a fresh method name: then $m_{i'} : \tau_{m_i} = b'_i \in \rho_3$ and $m_i : \tau_{m_i} = b_i[m_{i'}/next] \in \rho_3$.*

Notice that name clashes among methods during the application will never take place, since they have already been solved during the typing of mixin application.

The second set of rules for O'KLAIM, shown in Table 8, concerns processes and it is an extension of the operational semantics of KLAIM. Notice that the O'KLAIM's operational semantics must be defined on typed *compiled* processes, i.e., processes where each object-oriented value and tuples are decorated with their types, as explained in

$$\frac{exp_1 \twoheadrightarrow \left(\begin{array}{l}\mathsf{mixin}\\ \quad \mathsf{expect}[m_i : \tau_{m_i}{}^{i\in I}]\\ \quad \mathsf{redef}[m_k : \tau_{m_k} \text{ with } b_k{}^{k\in K}]\\ \quad \mathsf{def}[m_j : \tau_{m_j} = b_j{}^{j\in J}]\\ \mathsf{end}\end{array}\right) \qquad exp_2 \twoheadrightarrow \mathsf{class}\ [m_l : \tau_{m_l} = b_l{}^{l\in L}]\ \mathsf{end}}{exp_1 \diamond exp_2 \twoheadrightarrow \left(\begin{array}{l}\mathsf{class}\\ \quad [m_j : \tau_{m_j} = b_j{}^{j\in J}] \cup\\ \quad override([m_k : \tau_{m_k} = b_k{}^{k\in K}], [m_l : \tau_{m_l} = b_l{}^{l\in L}])\\ \mathsf{end}\end{array}\right)}$$

Table 7. The (*mixinapp*) operational rule

$$\frac{}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \rightarrowtail l :: P \parallel l' :: P' \mid \langle t \rangle}\ (\textsc{out})$$

$$\frac{match(t,t')}{l :: \mathbf{in}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \mathbf{nil}}\ (\textsc{in})$$

$$\frac{match(t,t')}{l :: \mathbf{read}(t)@l'.P \parallel l' :: \langle t' \rangle \rightarrowtail l :: P[t'/t] \parallel l' :: \langle t' \rangle}\ (\textsc{read})$$

$$\frac{}{l :: \mathbf{eval}(P)@l'.P' \parallel l' :: P'' \rightarrowtail l :: P' \parallel l' :: P' \mid P}\ (\textsc{eval})$$

$$\frac{exp \twoheadrightarrow v}{l :: \mathsf{let}\ x = exp\ \mathsf{in}\ P \rightarrowtail l :: P[v/x]}\ (\textsc{let}) \qquad \frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2 \qquad N_2 \equiv N'}{N \rightarrowtail N'}\ (\textsc{net})$$

Table 8. O'KLAIM operational rules

Section 2.5, because the crucial point is the dynamic matching of types. In fact, an **out** operation adds a tuple decorated with a (static) type to a tuple space, and a process can perform an **in** action by synchronizing with a process which represents a matching typed tuple. The rule for let $x = exp$ in P relies on the reduction relation for object-oriented expressions $\twoheadrightarrow$.

The predicate for tuples, *match*, is presented in Table 9. The matching rules exploit the static type information, delivered together with the tuple items, in order to dynamically check that the received item is correct with respect to the type of the formal field, say τ. Therefore, an item is accepted if and only if it is subtyping-compliant with the

$$match(e,e) \qquad match(l,l) \qquad \frac{match(t_2,t_1)}{match(t_1,t_2)}$$

$$\frac{match(t_1,t_2) \quad match(t_3,t_4)}{match((t_1,t_3),(t_2,t_4))} \qquad \frac{match(\sigma,\sigma_i)}{match(!id : \sigma, t_i{}^{\sigma_i})}$$

$$match(\sigma_1,\sigma_2) = \begin{cases} \sigma_1 \sqsubseteq \sigma_2 & \text{if } \sigma_1 \text{ and } \sigma_2 \text{ are mixin or class types} \\ \sigma_1 <: \sigma_2 & \text{otherwise} \end{cases}$$

Table 9. Matching rules (with $\mathsf{proc} <: \mathsf{proc}$ and $\mathsf{loc} <: \mathsf{loc}$)

expected type of the formal field. Informally speaking, one can accept any class containing more resources than expected, and any mixin with weaker requests about methods expected from the superclass can be accepted. This subtyping checking is analogous to the one we would perform in a sequential language where mixins and classes could be passed as parameters to methods. In a sequential setting, this dynamic checking might look as a burden (for example, in [13], mixins and classes are first-order entities, i.e., they can be passed as parameters in methods, but the matching among formal and actual parameters is by syntactic equality on types and not by subtyping), but in a distributed mobile setting the burden seems well-compensated by the added flexibility in communications.

Finally, the semantics for the distributed part is based on structural congruence and reduction relations. Reduction represents individual computation steps, and is defined in terms of structural congruence. The structural congruence $\equiv$ allows the rearrangement of the syntactic structure of a term so that reduction rules may be applied. It is defined as the least congruence relation closed under the following rules.

$$N_1 \parallel N_2 = N_2 \parallel N_1 \qquad (N_1 \parallel N_2) \parallel N_3 = N_1 \parallel (N_2 \parallel N_3) \qquad l :: P = l :: P \mid \mathbf{nil}$$
$$l :: (P_1 \mid P_2) = l :: P_1 \parallel l :: P_2$$

As a final remark, let us observe that we do not define a matching predicate for actual fields containing object-oriented values and processes since this would require to decide equalities on classes, mixins and objects (e.g., equality on their interfaces) and on processes (e.g., a bisimulation). This issue is out of the scope of the present work, since matching between two actual fields does not involve any substitution and then does not cause problems w.r.t. typing.

3 Typing issues and subject-reduction property

The important point in O'KLAIM semantics is that if a process P (statically well-typed) retrieves a tuple by the subtyping matching mechanism and the retrieved value is merged in the continuation of P, then the evaluation proceeds without any additional type-checking. Thus, in order to obtain the subject-reduction theorem, we need to prove that substitution preserves well-typedness, in particular when classes and mixins are replaced to variables inside object-oriented expressions. In the following, we address this issue and we outline the main technical steps, skipping proofs and details for lack of space.

The crucial case concerns mixin application expressions; namely if class/mixin variables are replaced by classes/mixins having a subtype, accidental overrides can occur because of names of the new methods added by the replacing value (see the definition of $\sqsubseteq$). This matter is related to the "width subtyping versus method addition" problem (well known in the object-based setting, see for instance [21]), that in our case boils down to a careful management of these *dynamic name clashes*. Thus, we must define a suitable capture-avoid-substitution, $[\ ]$, requiring possible renaming of methods with fresh names.

Definition 2 (Substitution by refresh). *If x is a class variable of type $\mathsf{class}\langle\Sigma\rangle$ and C is a class value of type $\mathsf{class}\langle\Sigma'\rangle$ such that $\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle$, then $[C/x]$ denotes the*

replacement of C' to x, where C' is obtained from C by renaming all methods belonging to $Meth(\Sigma') - Meth(\Sigma)$ with fresh names. For mixins variables and values, the renaming acts on all and only the methods belonging to $Meth(\Sigma'_{new}) - Meth(\Sigma_{new})$.

With our solution, new methods added by a class or a mixin value during substitution are hidden by renaming, for each occurrence of the variable to be replaced (this is very similar to the "privacy via subsumption" of [27]). On the other hand, we only rename methods that do not appear in the type of the variable x. This second constraint ensures a basic property: the refreshed version C' of C has a type τ' such that $\tau' \sqsubseteq \mathsf{class}\langle\Sigma\rangle$. The same holds for refreshed mixins.

Now, using this definition, we can prove that substitution is type safe. For simplicity, in order to deal with $<:$ and $\sqsubseteq$ at the same time, we introduce the meta notation:

$$\tau_1 \preccurlyeq \tau_2 = \begin{cases} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ is a mixin or a class type} \\ \tau_1 <: \tau_2 & \text{otherwise} \end{cases}$$

Lemma 1 (Substitution by narrowing). *Let v, exp and P be an object-oriented value, an object-oriented expression and a process, respectively,*

1. *if $x : \tau_1 \vdash exp : \tau$ and $\Gamma \vdash v : \tau_2$ where $\tau_2 \preccurlyeq \tau_1$, then $\Gamma \vdash exp[v/x] : \tau'$ with $\tau' \preccurlyeq \tau$;*
2. *if $x : \tau_1 \vdash P : \mathsf{proc}$ and $\Gamma \vdash v : \tau_2$ where $\tau_2 \preccurlyeq \tau_1$, then $\Gamma \vdash P[v/x] : \mathsf{proc}$.*

Sketch of proof:

1. By induction on typing rules for expressions. The only crucial case is when *exp* is a mixin application and v is a class value or a mixin. Notice that, *exp* is well-typed and no method occurring in the type x is renamed; then the last condition in (*mixin app*) rule is preserved and ensures that no name clash can occur after substitution.
2. By induction on typing rules for processes using the previous point.

Summarizing, the type safety of the communication results from two main properties: (i) static type soundness of local and foreign code; (ii) the preservation of well-typedness under substitution by subtyping. It is standard to verify that all the other rules concerning $\twoheadrightarrow$ preserve well typedness and so we obtain the *subject-reduction theorem*. Thus, the local evaluation of a process cannot produce errors like "message-not-understood" even if it retrieves data from foreign sites and merges it in the local configuration. In other words, a *well-typed net* (i.e., a net where each process in each site is well-typed) remains well typed during its evolution (*global safety condition*).

We remark that, from the point of view of the implementation, the above treatment of "global" fresh names can be solved with static binding for the mentioned methods. The technique of using the static types of variables and the actual types of substituted mixin and class definitions may recall the approach of [22] of allowing overriding, i.e., dynamic binding, only for methods declared in the mixin's *inheritance interface*.

4 The implementation

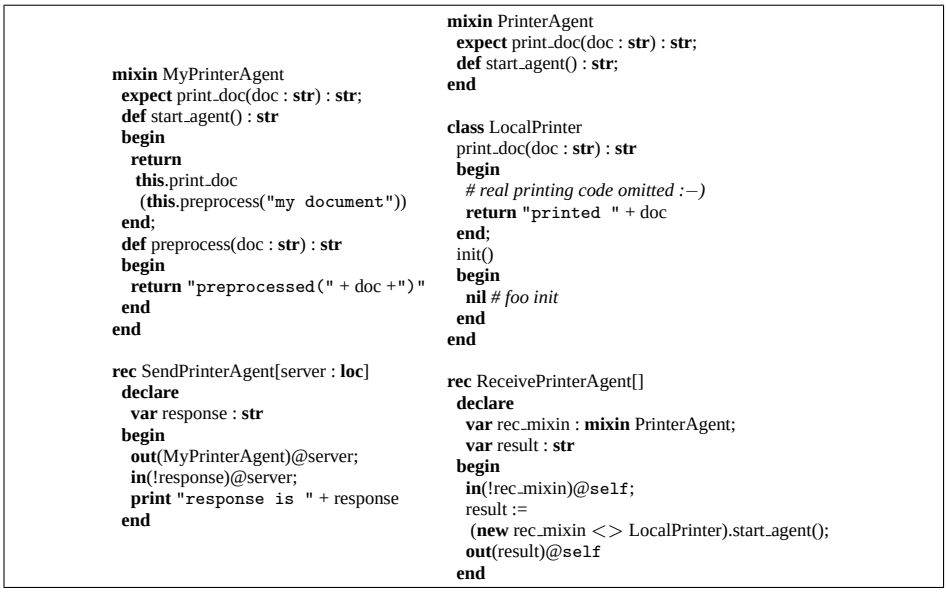
We recall that the implementation we present is based on X-KLAIM [10, 11] (extended with the proper object-oriented mixin-based primitives), used both as the "surface"

object-oriented calculus and as the coordination language, with the added bonus of being able to write methods that can perform KLAIM actions, all the same guaranteeing absence of message-not-understood run-time errors, as shown in the previous section.

The implementation of the O'KLAIM object-oriented component in Java consists in a package `momi` presented in [2] and described in details in [3]. This package provides the run-time system, or the virtual machine, for classes, mixins and objects that can be downloaded from the network and dynamically composed (via the mixin application operation). It thus provides functionalities for checking subtyping among classes and among mixins and for building at run-time new subclasses. Since we abstract from the specific communication and mobility features, this package does not provide means for code mobility and network communication, so that `momi` can be smoothly integrated into existing Java mobility frameworks. We would like to stress that this package should be thought of as an "assembly" language that is the target of a compiler for a high-level language (in our case the language is X-KLAIM). If `momi`, as it is, was used for directly writing object-oriented expressions, the programmer would be left with the burden of writing methods containing Java statements dealing with `momi` objects, classes and mixins, and to check manually that they are well typed. Basically these are the same difficulties a programmer has to face when using an assembly language directly, instead of a high-level language. We could say that `momi` enhances the functionalities of the Java virtual machine: while the latter already provides useful mechanisms for dynamically loading new classes into a running application, the former supplies means for dynamically building class hierarchies (based on mixins) and for inserting new subclasses into existing hierarchies (which is not possible in Java).

In order to implement O'KLAIM we extended the KLAIM programming framework that consists in the programming language X-KLAIM [10, 11], which extends KLAIM with high-level programming constructs, and KLAVA [12] a Java package that implements the run-time system for X-KLAIM operations (X-KLAIM programs are compiled into Java programs that use KLAVA). The package KLAVA already provided all the primitives for network communication, through distributed tuple spaces, and, in particular, for code mobility, not supplied by `momi`. Thus the package has been modified in order to be able to exchange code that is based on `momi`, and for performing subtyping on `momi` elements during pattern matching by relying on the `MoMiType` classes and the associated subtyping. On the other hand, the X-KLAIM compiler generates code that uses both the KLAVA package and `momi`. Obviously, before generating code, it also performs type checking according to the type system defined by MOMI. All this software is freely available at `http://music.dsi.unifi.it`.

The programming example shown in this section involves mixin code mobility, and implements "dynamic inheritance" since the received mixin is applied to a local parent class at run-time. We assume that a site provides printing facilities for local and mobile agents. Access to the printer requires a driver that the site itself has to provide to those that want to print, since it highly depends on the system and on the printer. Thus, the agent that wants to print is designed as a mixin, that expects a method for actually printing, `print_doc`, and defines a method `start_agent` through which the site can start its execution. The actual instance of the printing agent is instantiated from a class dynamically generated by applying such mixin to a local superclass that provides the



```
mixin MyPrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str
 begin
  return
   this.print_doc
    (this.preprocess("my document"))
 end;
 def preprocess(doc : str) : str
 begin
  return "preprocessed(" + doc +")"
 end
end

rec SendPrinterAgent[server : loc]
 declare
  var response : str
 begin
  out(MyPrinterAgent)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
mixin PrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str;
end

class LocalPrinter
 print_doc(doc : str) : str
 begin
  # real printing code omitted :–)
  return "printed " + doc
 end;
 init()
 begin
  nil # foo init
 end
end

rec ReceivePrinterAgent[]
 declare
  var rec_mixin : mixin PrinterAgent;
  var result : str
 begin
  in(!rec_mixin)@self;
  result :=
   (new rec_mixin <> LocalPrinter).start_agent();
  out(result)@self
 end
```

Listing 1: The printer agent example.

method `print_doc` acting as a wrapper for the printer driver. However the system is willing to accept any agent that has a compatible interface, i.e., any mixin that is a subtype of the one used for describing the printing agent. Thus any client wishing to print on this site can send a mixin that is subtyping compliant with the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 1, where **rec** is the X-KLAIM keyword for defining a process, presents a possible implementation of the printing client node (on the left) and of the printer server node (on the right). The printer client sends to the server a mixin `MyPrinterAgent` that complies with (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent`. In particular `MyPrinterAgent` mixin will print a document on the printer of the server after preprocessing it (method `preprocess`). On the server, once the mixin is received, it is applied to the local (super)class `LocalPrinter`, and an object (the agent) is instantiated from the resulting class, and started so that it can actually print its document. The result of the printing task is then retrieved and sent back to the client.

We observe that the sender does not actually know the mixin name `PrinterAgent`: it only has to be aware of the mixin type expected by the server. Furthermore, the sent mixin can also define more methods than those specified in the receiving site, thanks to the mixin subtyping relation. This adds a great flexibility to such a system, while hiding these additional methods to the receiving site (since they are not specified in the receiving interface they are actually unknown statically to the compiler).

5 Conclusions and related work

We have presented the kernel language O'KLAIM, which extends the higher-order calculus KLAIM for mobile processes with mixin-based object-oriented code. The novel

contributions of this paper, with respect to [8] (where we firstly presented design motivations for a mixin-based approach in a mobile context), can be summarized as follows:

1. we integrate the basic ideas of [8] into a mobile process calculus with an asynchronous and more sophisticated communication mechanism;
2. we refine the typing for the object-oriented component, we define a new typing system for KLAIM processes, and we study main typing concerns (in particular, a notion of substitution with renaming) in order to demonstrate the soundness of the proposed solution;
3. we present an implementation of O'KLAIM.

Keeping the O'KLAIM object-oriented calculus and the O'KLAIM processes separated may appear a limitation, but in fact this is not true. Our system consists of three components: the "surface" object-oriented component, a mixin/class subtyping relation, and a coordination calculus. If the object-oriented component is chosen to be an object-oriented concurrent/mobile language, the two components (object-oriented and concurrent/mobile) may interleave in a deeper way. A good example is the O'KLAIM implementation presented in Section 4: in there, X-KLAIM (extended with the proper object-oriented mixin-based primitives) is both the "surface" object-oriented calculus and the coordination language, so that method bodies can perform KLAIM actions. The matching mechanism that allows safe interactions during code exchange is based on the subtyping relation that acts as a general glue to glue together the two language components, of whichever nature they are (as long as the object-oriented one implements classes and mixins).

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [16] is a lexically-scoped language providing distributed object-oriented computation. Mobile code maintains network references and provides transparent access to remote resources. In [15], a general model for integrating object-oriented features in calculi of mobile agents is presented where agents are extended with constructs for remote method invocations. Other works, such as, e.g., [20, 26, 25] do not deal explicitly with mobile distributed code. Our approach is more related to papers, as [29], where properties of distributed systems are enforced by a typing system equipped with subtyping. In our case the property we address is a flexible and type-safe coordination for exchanging code among processes, up- and down-loading classes and mixins from different sites.

Further extensions of O'KLAIM are topics for future developments: subtyping can be extended to *depth subtyping*, which offers a more flexible communication pattern (see [9] for a preliminary discussion) and the object-oriented component can be enriched with *incomplete objects*, i.e., objects instantiated from mixins [5].

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a java extension with mixins. *ACM Transaction on Programming Languages and Systems*, 2003. To appear.
2. L. Bettini. A Java package for class and mixin mobility in a distributed setting. In *Proc. of FIDJI'03*, LNCS, 2003. To appear.
3. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.

4. L. Bettini, V. Bono, R. De Nicola, G. Ferrari, D. Gorla, M. Loreti, E. Moggi, R. Pugliese, E. Tuosto, and B. Venneri. The KLAIM Project: Theory and Practice. In *Global Computing – Trento*, LNCS. Springer, 2003. To appear.
5. L. Bettini, V. Bono, and S. Likavec. A core calculus of mixin-based incomplete objects. In *FOOL 11*, 2004.
6. L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Manuscript.
7. L. Bettini, V. Bono, and B. Venneri. Towards Object-Oriented KLAIM. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
8. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In *Proc. of Coordination Models and Languages*, volume 2315 of *LNCS*, pages 56–71. Springer, 2002.
9. L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *Proc. of FOOL*, 2003.
10. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of WETICE*, pages 110–115. IEEE Computer Society Press, 1998.
11. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
12. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, 32(14):1365–1394, 2002.
13. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer, 1999.
14. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA*, pages 303–311, 1990.
15. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
16. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
17. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. of ICSE*, pages 22–33. ACM Press, 1997.
18. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
19. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
20. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *Proc. of CONCUR*, volume 1119 of *LNCS*, pages 655–670, Pisa, Italy, 26–29 Aug. 1996. Springer.
21. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *LNCS*, pages 42–61. Springer, 1995.
22. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
23. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. of CONCUR*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.
24. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
25. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Proc. of HLCL*, volume 16.3 of *ENTCS*. Elsevier, 1998.
26. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Proc. of TPPP*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
27. J. Riecke and C. Stone. Privacy via Subsumption. *Information and Computation*, 172:2–28, 2002. 3rd special issue of Theory and Practice of Object-Oriented Systems (TAPOS).
28. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.
29. N. Yoshida and M. Hennessy. Subtyping and Locality in Distributed Higher Order Mobile Processes (extended abstract). In *Proc. of CONCUR'99*, volume 1664 of *LNCS*, pages 557–572. Springer-Verlag, 1999.