Towards Object-Oriented Klaim

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

By its own nature, *mobile code* requires flexibility in order to be adaptive to any execution context it may be run in. In this paper we investigate this flexibility requirement from the design point of view, and propose a solution based on the *mixin* technique to fulfill it. We also propose an extension of the language Klaim with object-oriented features, as an application of this approach.

1 Introduction

Internet provides technologies that allow the sharing of resources and services among computers distributed geographically in wide area networks. The growing use of Internet as a primary environment for developing, distributing and running programs requires new supporting infrastructures. A possible answer to these requirements is the use of *mobile code* [25,12] and in particular of *mobile agents* [19,28], which are software objects made of data and code that can be sent on the net, or can autonomously migrate to a remote computer and execute automatically on arrival.

Mobile code should be rather flexible in order to exploit the power of communication and make a fruitful use of all the features/resources belonging to the net's sites taking part into the communication. In particular, the need of processing tasks for which not all resources are available on the site they reside may be fulfilled (*i*) either by importing additional code from remote sites (*code-on-demand*); (*ii*) or by sending the task code for *remote evaluation* to a remote site [12].

Thus mobile code must be very adaptive to different execution environments, yet it should be structured enough to specify, without ambiguities, which are the needs it expects to fulfill and which are the constraints it is able to respect. With these objectives in mind, it seems appealing to structure mobile code following an object-oriented approach. Moreover, the natural structure of object-oriented languages allows to statically type check the code in each site independently, and we can use this type information to dynamically match code received from a remote site, so that the execution is still safe in a distributed context. However, the standard, essentially static, features of the inheritance mechanism for achieving flexibility do not scale well to a mobile distributed context, as discussed in Section 3. For these reasons, we propose an alternative approach, based on the notion of mixins, and in particular on a *mixin*-based calculus [5], for modelling the dynamism required by mobility in case of object-oriented code.

Mixins (classes parameterized over superclasses) have become a focus of active research both in the software engineering [26,24] and programming language design [7,20,15] communities. Mixin inheritance has been shown to be an expressive alternative to multiple inheritance [6] and a powerful tool for implementing reusable class hierarchies. The key idea of mixin-based inheritance is that it is more oriented to the concept of "completion" (since it is the derived class, the mixin, that is completed with base classes, by means of an "application" operation), as opposed to the concept of extendibility/specialization, that is at the heart of class-based inheritance (where the base class is extended/specialized by subclasses). Thus, for example, the mixin approach would allow a site to provide the base class, typically implementing the high-level behavior of an architecture or of a strategy, and let the mobile code (e.g., a mobile agent), coded as a mixin, provide the particular implementation (this is typical of *design patterns* [16]).

This paper aims at putting at test the mixin approach in a mobile scenario, where object-oriented code is exchanged on the net. As an application, we propose an extension of the language Klaim (*Kernel Language for Agent Interaction and Mobility*) [13] with object-oriented features. The paper is structured as follows. Section 2 briefly recalls the mixin calculus of [5]. In Section 3 we investigate two relevant scenarios for mobile code. Section 4 introduces the object-oriented extension of Klaim, and Section 5 uses this extension to picture two examples of the scenarios presented earlier.

2 The Mixin Calculus

A *mixin* (a class definition parameterized over the superclass) can be viewed as a function that takes a class as a parameter and derives a new subclass from it. The same mixin can be applied to many classes, obtaining a family of subclasses with the same set of methods added and/or redefined. A subclass can be implemented before its superclass has been implemented, thus mixins remove the dependency of the subclass on the superclass, enabling modular development of class hierarchies.

The formal setting of our approach is the mixin calculus presented in [5], which is a standard calculus of functions, records, and imperative features with new constructs to support classes and mixins. There are four main expressions involving classes: $\mathsf{classval}$, mixin, $\diamond$ (mixin *application*), and new

(class *instantiation*). Class values can be created by mixin application, and objects can be created by class instantiation. A class value is essentially a *generator* function, that, at the moment of class instantiation, is applied to the field values and it solves all the hierarchy relations by calling its superclass' generator function.

The syntax of the calculus is summarized in Table 1[1]; we briefly comment the parts that are relevant for our purposes (for the operational details of the calculus we refer to [5]).

- $\mathsf{classval}\langle v_g, [m_i]^{i \in Meth}, [p_\ell]^{\ell \in Prot}\rangle$ is a *class value*, i.e., the result of mixin application. It is a triple, containing one function and two sets of variables. The function v_g is the generator for the class. The $[m_i]$ set contains the names of all methods defined in the class, and the $[p_\ell]$ set contains the names of protected methods.

- mixin
  method $m_j = v_{m_j};\quad ^{(j \in New)}$
  redefine $m_k = v_{m_k};\quad ^{(k \in Redef)}$
  protect $[p_\ell];\quad ^{(\ell \in Prot)}$
  constructor v_c;
  end
  is a *mixin*, in which $m_j = v_{m_j}$ are definitions of new methods, and $m_k = v_{m_k}$ are method redefinitions that will replace methods with the same name in any class to which the mixin is applied. $v_{m_{j,k}}$ are value-expressions defining a method body. Namely, each $v_{m_{j,k}}$ is a function of *self*, which will be bound to the newly created object at instantiation time, and of the private *field*. In method redefinitions, v_{m_k} is also a function of *next*, which will be bound to the old, redefined method from the superclass. The v_c value in the constructor clause is used when evaluating a mixin application to build the generator of the class.

- $e_1 \diamond e_2$ is an application of mixin e_1 to class value e_2. It produces a new class value.

- new e uses generator v_g of the class value, to which e evaluates, to create a function that returns a new object, as briefly explained above.

The mixin calculus also relies on the Wright-Felleisen idea of *store* [29], called *heap* here, in order to evaluate imperative side effects: the expression $\mathsf{H}\langle x_1, v_1\rangle \ldots \langle x_n, v_n\rangle.exp$ associates reference variables $x_1, \ldots x_n$ with values $v_1, \ldots, v_n$. H binds $x_1, \ldots, x_n$ to $v_1, \ldots, v_n$ in *exp*. The set of pairs h in the expression $\mathsf{H}h.exp$ represents the *heap*, where the results of evaluating imperative subexpressions of *exp* are stored.

Finally, the calculus is equipped with a type system, centered around the typing of mixin and of mixin application expressions. Here we only comment

[1] In order to avoid ambiguity with the . used as the action prefix of Klaim, we use $\Leftarrow$ for method selection.

$$
\begin{array}{l}
exp ::= const \mid x \mid \lambda x.exp \mid exp_1\ exp_2 \mid \mathit{fix} \mid \mathsf{ref} \mid \mathsf{deref} \mid := \\
\quad \mid \{x_i = exp_i\}^{i \in I} \mid exp \Leftarrow x \mid \mathsf{H}\ h.exp \mid \mathsf{new}\ exp \\
\quad \mid \mathsf{classval}\langle v_g, [m_i]^{i \in Meth}, [p_\ell]^{\ell \in Prot} \rangle \\
\quad \mid\ \mathsf{mixin} \\
\qquad \mathsf{method}\ m_j = v_{m_j}; \quad ^{(j \in New)} \\
\qquad \mathsf{redefine}\ m_k = v_{m_k}; \quad ^{(k \in Redef)} \\
\qquad \mathsf{protect}\ [p_\ell]; \quad ^{(\ell \in Prot)} \\
\qquad \mathsf{constructor}\ v_c; \\
\quad\ \ \mathsf{end} \\
\quad \mid exp_1 \ \diamond\ exp_2
\end{array}
$$

Table 1: the syntax of the calculus of mixins of [5].

the types of classes and mixins (we refer the interested reader to [5] for the complete type system). In the following $\{x_i : \tau_i\}^{i \in I}$ is a record type and $[p_\ell]$ is a set of names.

A class type has the following form:

$$\mathsf{class}\langle \gamma, \{m_i{:}\tau_i\}, [p_\ell] \rangle^{i \in I, \ell \in L}$$

where γ is the argument type of the constructor, $\{m_i{:}\tau_i\}^{i \in All}$ is the record type of methods (which is also the type of *self*).

A mixin type gives information about the types of the methods already present in the mixin itself and makes type assumption about the methods of the superclass to which the mixin will be applied:

$$\mathsf{mixin}\langle \gamma_b, \gamma_d, \{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}, \{m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\}, [p_\ell] \rangle$$

where

- γ_b is the expected argument type of the superclass constructor.
- γ_d is the exact argument type of the mixin constructor.
- $\{m_i : \tau^{\uparrow}_{m_i}, m_k : \tau^{\uparrow}_{m_k}\}$ are the expected types of the methods that must be supported by any class to which the mixin is applied. In particular, m_i are the methods that are not redefined by the mixin but still expected to be supported by the superclass since they are called by other mixin methods. Analogously, $\tau^{\uparrow}_{m_k}$ are the types assumed for the old bodies of the methods redefined in the mixin.
- $\{m_j{:}\tau^{\downarrow}_{m_j}, m_k{:}\tau^{\downarrow}_{m_k}\}$ are the exact types of mixin methods (new and redefined, respectively).
- $[p_\ell]$ is an annotation listing the names of all methods to be protected, both new and redefined.

The exact relationship between the types expected by the mixin and the actual types of the superclass methods is formalized in the rule for typing mixin application, where constraints are specified and have to be met in order to build a full-fledged type-safe derived class from the application of a mixin to a (base) class.

3 Mobility and Object-Oriented Code

Mobile code has been introducing some new programming paradigms [12] that can be seen as evolved instances of the classical client-server design for distributed applications. Indeed, in order to fulfill a specific task, a distributed application can rely on mobile code in several ways:

- code can be downloaded from remote sites and executed locally (*code-on-demand*);
- code can be sent to a remote site and executed there (*remote evaluation*);
- *mobile agents* can be scattered through the net so that they can autonomously continue their execution on the computers they are visiting.

All these scenarios lead to the requirement of strong flexibility for mobile code. This is evident in some, possible orthogonal, aspects that can be summarized into two main ones:

- architecture and operating system independence (also called *on-line portability* [10]): this allows mobile code to execute in several different heterogeneous computers and operating systems;
- *structural flexibility*: mobile code must be able to adapt itself to the current execution environment and to react accordingly (e.g., when some needed operations and resources are not available on the local computer).

In this paper we address the second issue, and we aim at investigating it under an object-oriented perspective. In order for mobile code to be adaptive, it needs to be "incomplete": those parts that are independent of the execution environment must be abstracted away from the operations that need to be specialized on the execution site (e.g., those that rely on system calls). The execution site is then demanded to provide the implementations of the latter, and so to "complete" the mobile code.

The object-oriented approach, with its mechanisms of inheritance and code specialization, seems to be suitable for dealing with this issue. However, we observe that the classical inheritance operation is essentially static in that it fixes the inheritance hierarchy, i.e., it binds derived classes to their parent class (or to their many parent classes, if multiple inheritance is provided by the language) once for all. If such a hierarchy has to be changed, the program must be modified and then recompiled. Indeed, what we are looking for is a mechanism for providing a dynamic reconfiguration of the inheritance relation between classes, not only a dynamic implementation of some operations.

Let us illustrate this by considering two different scenarios:

Scenario A (*Remote evaluation*). A base class is typically more appropriate for modelling a generic behavior, so it is suitable for the local code. Thus the base class provides generic operations of the execution site (e.g. system calls). On the other hand, mobile code is better seen as the derived class containing methods that can exploit those generic operations.

Scenario B (*Code-on-demand*). A site that downloads code for local execution may want to redefine some, possible critic, operations that the downloaded code may execute. This way the local site can be sure that access to some sensitive resources is not granted to non-trusted code (for example, it might decide to change some destructive "read" operations in non-destructive ones in order to avoid that non-trusted code erases information). Thus the downloaded code is better modelled with a base class, that is specialized locally into a derived class after downloading.

Summarizing, in **A** the base class is the local code, while in **B** the base class is the mobile code.

Clearly, the classical inheritance mechanism cannot structure the mobile object-oriented code in order to reach the dynamism required in both scenarios. Firstly, we could think of a mechanism for dynamically building a class hierarchy when the code-on-demand base class is downloaded, and this would somehow implement scenario B. However this technique would not help implementing scenario A, since it would require a not-so-clear dynamic definition of the base class. Secondly, we are convinced that the two scenarios should be dealt with by the same mechanism, allowing to dynamically use mobile code, in different environments, both as a base class for deriving new classes, and as derived class for being "adopted" by a parent class.

The aim of the present paper is to show, from the point of view of the design, how the mixin approach is a smooth solution for modelling mobility of object-oriented code in any scenario, notably the two described above, since mixin-based inheritance is more oriented to the concept of "completion" than to the concept of extendibility/specialization typical of the classical form of inheritance. Namely, with mixin-based inheritance, the inheritance relation between a derived and a base class is not established through a declaration (e.g., like `extends` in Java); instead it actually takes place through an operation, the mixin application, during the execution of a program, and it is not in its declaration part.

While with standard class-based inheritance the only operation available is the one that allows a class to inherit from a base class, in mixin-based inheritance also the complementary operation is provided: a base class can be applied to a class (a mixin) that becomes a derived class. Since all possibly derived classes (the mixins) are now parameterized over superclasses, the inheritance relation does not strongly couple derived classes and base classes anymore, thus code reusability can be exploited at a higher level.

4 The language O'Klaim (Object Oriented Klaim)

In this section we propose to use the mixin approach for extending the language Klaim[2] (*Kernel Language for Agent Interaction and Mobility*) with object-oriented features.

Firstly, we briefly recall the main features of Klaim, then we show its extended version O'Klaim, and finally we sketch how types are used in order to guarantee the safe transmission and usage of mobile object-oriented expressions.

4.1 An overview of Klaim

Klaim is a language that supports a programming paradigm where both processes and data can be moved over different computing environments. It is specifically centered on the notion of mobile code, implementing mobile agents, and it is inspired by the Linda coordination model [17,11], relying on the concept of *tuple space*; thus, processes can communicate with processes running in other sites.

A tuple space is a multiset of *tuples*; these are containers of information items (called *fields*). There can be *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.

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 match: a formal field matches any value of the same type, and two actual fields match only if they are identical (but two formals never match). For instance, tuple ("foo", "bar", $100 + 200$) 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 integer value 300. The pattern-matching predicate, used in the operational semantics (see [13] for details), is reported in Table 2. We observe that tuple items in Klaim are not typed explicitly: the matching relies on the syntactic form of the identifiers (u for locality variables, x for basic expressions and X for processes).

In Linda there is only one global shared tuple space; Klaim extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* that are part of a *net*. Each node contains a single tuple space and processes in execution; a node can be accessed through its *locality*[3]. A reserved locality, `self`, can be used by processes to refer to their execution node.

[2] The design philosophy and the detailed description of the language are presented in [13]; Klaim prototype implementation is described in [2].

[3] For the purposes of this paper, the distinction between *logical localities* and *physical localities* and the notion of *allocation environment* are not relevant, so we skip them; we refer the interested reader to [13] for further details.

$$match(e, e) \qquad match(\ell, \ell) \qquad match(P, P)$$

$$match(!\,x, e) \qquad match(!\,u, \ell) \qquad match(!\,X, P)$$

$$\frac{match(et_2, et_1)}{match(et_1, et_2)} \qquad \frac{match(et_1, et_2) \quad match(et_3, et_4)}{match((et_1, et_3), (et_2, et_4))}$$

Table 2: The Matching Rules

Let us briefly comment the syntax of KLAIM *Processes*, summarized in Table 3. Processes are the active computational units and may be executed concurrently either at the same site or at different sites. They are built from the basic operations by using standard operators borrowed from process algebras [21], such as, e.g., *action prefixing*, *parallel composition* and *nondeterministic choice*.

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

Table 3: KLAIM Process Syntax

KLAIM processes can perform five basic operations over nodes. $\mathbf{in}(t)@\ell$ evaluates the tuple t and looks for a matching tuple 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. $\mathbf{read}(t)@\ell$ 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. $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space located at ℓ. $\mathbf{eval}(P)@\ell$ spawns process P for execution at node ℓ. $\mathbf{newloc}(u)$ creates a new node in the net and binds its site to u. The node can be considered a "private" node that can be accessed by the other nodes only if the creator communicates the value of variable u, which is the only way to access the fresh node.

4.2 O'Klaim

In the extension O'Klaim we propose here, object-oriented mobile code can be exchanged on the net, as well as processes, according to the Linda programming paradigm. O'Klaim processes are formally defined in Table 4. By using the mixin-based approach above introduced, mobile code can be used by the receiving site in several ways. Namely it can be:

- *composed with a class*, if it represents a mixin that needs to be completed, so creating a new class;
- *instantiated*, if it represents a class or a mixin, so giving rise either to an object or to an *incomplete object*, respectively;
- *composed with another object*, if it represents an incomplete object, i.e. an object instantiated from a mixin;

P	$::=$	$\mathbf{nil}$	(null process)
	$\mid$	$act.P$	(action prefixing)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$P_1 + P_2$	(non-deterministic choice)
	$\mid$	X	(process variable)
	$\mid$	$A\langle \widetilde{P}, \widetilde{\ell}, \widetilde{e} \rangle$	(process invocation)
	$\mid$	exp	(object-oriented expression)
	$\mid$	$\mathsf{let}\ x = exp\ \mathsf{in}\ P$	(let)
act	$::=$	$\mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(u)$	
t	$::=$	$f \mid f, t$	
f	$::=$	$e \mid P \mid \ell \mid !\,x \mid !\,X \mid !\,u \mid !\,m : \tau$	

Table 4: O'Klaim Process Syntax

For our purpose, the operations of **in** (and **read**) and **out**, retrieving from and inserting into a tuple space, are particularly significant. In order to obtain O'Klaim, we extend the Klaim syntax of tuples t so including any object-oriented expression. In particular, $!\,m : \tau$ is a formal field for retrieving an object oriented expression of type τ (the details of types are explained in Section 4.3). Then, in the particular case when t is a mixin, the actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ are used to move a piece of object-oriented code from/to a locality ℓ, respectively. Moreover, the set of actions has to be extended in order to include operations on object-oriented expressions, as defined in the mixin calculus presented in Section 2. We introduce the construct $\mathsf{let}\ x = \mathit{exp}\ \mathsf{in}\ P$ as a terminating process in order to pass to the sub-process P the results of an object-oriented computation.

Figure 1 depicts a possible use of the extended syntax.

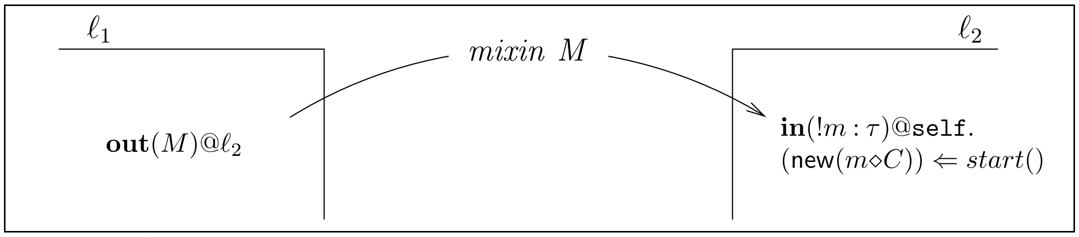
ℓ_1
mixin M
ℓ_2
out$(M)@\ell_2$
in$(!m : \tau)$@`self`.
$(\mathsf{new}(m \diamond C)) \Leftarrow \mathit{start}()$

Fig. 1: An example of remote evaluation based on object-oriented mobile code: the receiver site completes the received mixin m (of type τ) with the local class C, it creates an instance of the new class, and calls method *start* on the newly created object.

4.3 Typing

At this stage we are not interested in typing every kind of tuple items, thus we can associate a type void to them but for the case when they are object oriented expressions. These are typed by the inference rules defined in [5]. We observe that rules for classes are standard, while mixin-types encode type requirements on expected class arguments (Section 2). Thus mixins are type-checked locally, in the site where they are defined. As a consequence, the operational semantics of O'KLAIM will take account type information. Indeed, an **out** operation adds a tuple decorated with a type to a tuple space. Conversely, a process can perform an **in** action by synchronizing with a process which represents a matching typed tuple.

To this aim, the standard matching predicate for tuples, *match* (Table 2), is rewritten as *typedmatch*, as follows:

$$\mathit{typedmatch}(f_1, f_2) = \begin{cases} \mathit{match}(f_1, f_2) \\ \qquad\qquad \text{if (type of } f_1) = (\text{type of } f_2) = \mathsf{void} \\ (\text{type of } f_1) = (\text{type of } f_2) \\ \qquad\qquad \text{otherwise} \end{cases}$$

In particular, the last condition checks whether the object-oriented code, downloaded from a remote site by an **in** (or **read**) operation, is an object-oriented expression that behaves in accordance with the expected features.

The main advantage of this matching is the dynamic use of statically inferred types. The type of the retrieved expression is built statically by the type inference system of [5]. Then *typedmatch* uses this type information, delivered together with the retrieved expression, in order to dynamically check that the received item is correct w.r.t. the type of the formal field, say τ. In a process of the form $\mathbf{in}(!\,m:\tau)@l.P$, the type τ is used to statically type check the continuation P, where m is possibly used.

The type inference system of O'Klaim is obviously connected to the issue of *type safety*, that naturally pops up when object-oriented expressions are moved to different localities and used in different environments. This property can be essentially enforced by the type inference system already defined for the mixin calculus [5]. Then *type matching* guarantees that the moved code complies with the expected code. As a consequence, completing mobile mixins for defining new derived/base classes on the remote site will produce only well-typed classes, so avoiding run-time type errors. Thus *type safety of the communication results from the (static) type soundness of local and mobile code, with no need of further re-compilation and type-checking.* This topic will be widely investigated in a separate paper (work in progress) dealing with operational semantics and type properties for O'Klaim. Moreover, this type discipline can be embedded into the type system defined for Klaim in [14], that aims at enforcing secure access control for processes.

5 Mixin Mobility in O'Klaim

We present in the following two simple examples showing mobility of mixins in O'Klaim with types. They code the remote evaluation and the code-on-demand situations discussed in Section 3. Let us observe that both situations can be seen as examples of mobile agents as well.

Example 1. Let `agent` represent the type of a mixin defining a mobile agent that has to print some data by using the local printer on any remote site where it is shipped for execution. Obviously, since the *print* operation highly depends on the execution site (even only because of the printer drivers), it is sensible to leave such method to be defined. The mixin can be applied, on the remote site, to a local class *printer* which provides the specific implementation of the *print* method in the following way:

$$\begin{array}{l}\mathbf{in}(!\mathit{mob\_agent}:\texttt{agent})@\texttt{self}.\\ \mathsf{let}\ \ \mathit{PrinterAgent} = \mathit{mob\_agent} \diamond \mathit{printer}\ \ \mathsf{in}\\ \quad(\mathsf{new}\ \mathit{PrinterAgent}) \Leftarrow \mathit{start}()\end{array}$$

Example 2. Let `agent` be a class defining a mobile agent that has to access the file system of a remote site. If the remote site wants to execute this agent while restricting the access to its own file system, it can locally define a mixin *restricted*, redefining the methods accessing the file system according to specific restrictions. Then the arriving agent can be composed with the local mixin in the following way.

$$\begin{array}{l}\mathbf{in}(!\mathit{mob\_agent}:\texttt{agent})@\texttt{self}.\\ \mathsf{let}\ \ \mathit{RestrictedAgent} = \mathit{restricted} \diamond \mathit{mob\_agent}\ \ \mathsf{in}\\ \quad(\mathsf{new}\ \mathit{RestrictedAgent}) \Leftarrow \mathit{start}()\end{array}$$

This example can be seen as an implementation of a “sandbox”.

The above examples highlight how an object-oriented expression ($!mob\_agent$) can be used by the receiver site both as a mixin (Example 1) and as a base class[4] (Example 2). We remark that a solution based on *delegation* would destroy at least the dynamic binding and the reusability of the whole system [3]. Instead, the main feature of this approach is that new classes, obtained by composing local and remote code, are directly embedded into class hierarchies locally developed, without changing existing code while preserving polymorphism by subtyping.

6 Conclusions and Related work

In the literature, there are several proposal for combining objects with processes and/or mobile agents. *Obliq* [9] is a lexically-scoped interpreted language providing distributed object-oriented computation. Mobile code maintains network references and provides transparent access to remote resources. In [8], a general model for integrating object-oriented features in calculi of mobile agents is presented: agents are extended with method definitions and constructs for remote method invocations. In [4], concurrency is introduced in a object-based language, through the identification of objects and processes. In [23], concurrent object-based programming is considered, using the PICT language. [18] extends the Abadi and Cardelli's imperative object calculus [1] with operators for concurrency from the π-calculus [22] and with other operators for synchronization. Apart from [9,8], all these works are then concentrated on synchronization features of concurrent objects.

In this paper we are concerned about exchange of mobile code. Central in our approach is the fact that we chose a class-based calculus, instead of being centered on a object-based one, and the whole calculus seems to shed some light on how the mixin-based approach can be fruitfully used to see classes as mobile (possibly incomplete) processes. In our calculus, objects are not made distributed explicitly, thus no remote method call functionality is considered. Instead of formalizing remote procedure calls (like most of the above mentioned approaches), we are interested in the introduction of safe and scalable distribution of object-oriented code, in a calculus where communication facilities are already provided. To this aim we add object-oriented expressions to a mobile code kernel language and establish the foundations for their integration in a flexible and uniform setting.

References

[1] Abadi, M. and L. Cardelli, “A Theory of Objects,” Springer, 1996.

[4] Every mixin can be formally made into a class value by applying it to the empty top class *Object*, as explained in [5].

[2] Bettini, L., R. De Nicola, G. Ferrari and R. Pugliese, *Interactive Mobile Agents in* X-Klaim, in: P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)* (1998), pp. 110–115.

[3] Bettini, L., M. Loreti and B. Venneri, *On Multiple Inheritance in Java*, in: *Proc. of TOOLS EASTERN EUROPE, Emerging Technologies, Emerging Markets*, 2002, to appear.

[4] Blasio, P. D. and K. Fisher, *A Calculus for Concurrent Objects*, in: U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th Int. Conf.*, LNCS **1119** (1996), pp. 655–670.

[5] Bono, V., A. Patel and V. Shmatikov, *A Core Calculus of Classes and Mixins*, in: R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LCNS (1999), pp. 43–66.

[6] Boyen, N., C. Lucas and P. Steyaert, *Generalised Mixin-based Inheritance to Support Multiple Inheritance*, Technical Report vub-prog-tr-94-12, Vrije Universiteit Brussel (1994).

[7] Bracha, G. and W. Cook, *Mixin-based inheritance*, in: *Proc. OOPSLA '90*, 1990, pp. 303–311.

[8] Bugliesi, M. and G. Castagna, *Mobile Objects*, in: *Proc. of FOOL*, 2000.

[9] Cardelli, L., *A Language with Distributed Scope*, Computing Systems **8** (1995), pp. 27–59.

[10] Cardelli, L., *Mobile computation*, in: Vitek and Tschudin [27], pp. 3–6.

[11] Carriero, N. and D. Gelernter, *Linda in Context*, Comm. of the ACM **32** (1989), pp. 444–458.

[12] Carzaniga, A., G. Picco and G. Vigna, *Designing Distributed Applications with Mobile Code Paradigms*, in: R. Taylor, editor, *Proc. of the 19th Int. Conf. on Software Engineering (ICSE '97)* (1997), pp. 22–33.

[13] De Nicola, R., G. Ferrari and R. Pugliese, Klaim*: a Kernel Language for Agents Interaction and Mobility*, IEEE Transactions on Software Engineering **24** (1998), pp. 315–330.

[14] De Nicola, R., G. Ferrari, R. Pugliese and B. Venneri, *Types for Access Control*, Theoretical Computer Science **240** (2000), pp. 215–254.

[15] Flatt, M., S. Krishnamurthi and M. Felleisen, *Classes and mixins*, in: *Proc. POPL '98*, 1998, pp. 171–183.

[16] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[17] Gelernter, D., *Generative Communication in Linda*, ACM Transactions on Programming Languages and Systems **7** (1985), pp. 80–112.

[18] Gordon, A. D. and P. D. Hankin, *A Concurrent Object Calculus: Reduction and Typing*, in: U. Nestmann and B. C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, ENTCS **16.3** (1998).

[19] Harrison, C., D. Chess and A. Kershenbaum, *Mobile agents: Are they a good idea?*, Research Report 19887, IBM Research Division (1994).

[20] Limberghen, M. V. and T. Mens, *Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems*, Object Oriented Systems **3** (1996), pp. 1–30.

[21] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[22] Milner, R., J. Parrow and J. Walker, *A Calculus of Mobile Processes, I and II*, Information and Computation **100** (1992), pp. 1–40, 41–77.

[23] Pierce, B. C. and D. N. Turner, *Concurrent Objects in a Process Calculus*, in: T. Ito and A. Yonezawa, editors, *Proc. Theory and Practice of Parallel Programming (TPPP 94)*, LNCS **907** (1995), pp. 187–215.

[24] Smaragdakis, Y. and D. Batory, *Implementing layered designs with mixin layers*, in: *Proc. ECOOP '98*, 1998, pp. 550–570.

[25] Thorn, T., *Programming Languages for Mobile Code*, ACM Computing Surveys **29** (1997), pp. 213–239, also Technical Report 1083, University of Rennes IRISA.

[26] VanHilst, M. and D. Notkin, *Using role components to implement collaboration-based designs*, in: *Proc. OOPSLA '96*, 1996, pp. 359–369.

[27] Vitek, J. and C. Tschudin, editors, "Mobile Object Systems - Towards the Programmable Internet," Springer, 1997.

[28] White, J. E., *Mobile Agents*, in: J. Bradshaw, editor, *Software Agents* (1996).

[29] Wright, A. and M. Felleisen, *A syntactic approach to type soundness*, Information and Computation **115** (1994), pp. 38–94.