Coordinating Mobile Object-Oriented Code

Coordinating Mobile Object-Oriented Code*

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. Standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not seem to scale well to a distributed context with mobility. In this paper, a *mixin*-based approach is proposed for structuring mobile object-oriented code and it is shown to fit in the dynamic and open nature of a mobile code scenario. We introduce MoMi (Mobile Mixins), a coordination language for mobile processes that communicate and exchange object-oriented code in a distributed context. MoMi is equipped with a type system, based on polymorphism by subtyping, in order to guarantee *safe* code communications.

1 Introduction

Internet provides technologies that allow the transmission of resources and services among computers distributed geographically in wide area networks. The growing use of a network 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* [26, 11] and in particular of *mobile agents* [20, 19, 28], which are software objects consisting of data and code that can autonomously migrate to a remote computer and execute automatically on arrival.

On the other hand, the object-oriented paradigm has become established as a well suited technology for designing and implementing large software systems. In particular it provides a high degree of modularity, then of flexibility and reusability, so that it is widely used also in distributed contexts (see, e.g., Java [2] and CORBA [23]).

The new scenario arising from mobility puts at test the *flexibility* of object-oriented code in a wider framework. Object-oriented components are often developed by different providers and may be downloaded on demand for being dynamically assembled with local applications. As a consequence, they have to be strongly adaptive to any local execution environment, so that they can be dynamically reconfigured in several ways: downloaded code can be specialized by locally defined operations, conversely, locally developed software can be extended with new operations available from the downloaded code.

* 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.

Hence a coordination language allowing to program object-oriented components as mobile processes should provide specific mechanisms for coordinating not only the transmission, but also the local dynamic reconfiguration of object-oriented code.

In this paper we address the above issue in the specific context of class-based languages, that "form the main stream of object-oriented programming" [1]. We propose to use a new *mixin*-based approach for structuring mobile object-oriented code, as an alternative to standard inheritance mechanisms (Section 2). 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 (the operation is known as *mixin application*), 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 dependence of the subclass on the superclass, enabling dynamic development of class hierarchies. Mixins have become a focus of active research both in the software engineering [27, 25, 14] and programming language design [7, 21, 15] communities. In our approach, we use mixins and mixin application as the coordination mechanism for assembling mobile components in a flexible and safe way.

The paper proposes a formal calculus, MoMi, that integrates the mixin technology (relying on the calculus of [6]) into a core coordination language for mobile processes in a distributed context, where the main features of both components coexist in a uniform way. MoMi can be seen as a kernel language for programming and coordinating network services that allow remote communication with transmission of object-oriented code and dynamic reconfiguration of class hierarchies. The calculus is equipped with a type system, based on polymorphism by subtyping. We focus our attention on the subtyping relation on mixins, which comes out to be the main tool for coordinating the composition of local and mobile code in a safe way. The mobile code enjoys the benefits deriving from being statically type-checked and remote communications use static types "dynamically" to coordinate themselves (see Section 4).

The paper is structured as follows. In Section 2 we motivate our approach by investigating scenarios of object-oriented mobile code. In Section 3 we define the calculus MoMi whose operational semantics is given in Section 4. The type system of MoMi is presented in Section 5. In Section 6 we show an implementation of an example scenario in MoMi. Section 7 concludes the paper and discusses some related works and future directions.

2 Mobility and Object-Oriented Code

In this section we discuss two different scenarios, where an object-oriented application is received from (sent to) a remote site. In this setting we can assume that the application consists of a piece of code A that moves to a remote site, where it will be composed with a local piece of code B. These scenarios may take place during the development of an object-oriented software system in a distributed context with mobility.

Scenario 1 The local programmer may need to dynamically download classes in order to complete his own class hierarchy, without triggering off a chain reaction of changes over the whole system. For instance, he may want the downloaded class A to be a child class of a local class B. This generally happens in *frameworks* [17]: classes of the framework provide the general architecture of an application (playing the role of the local software), and classes that use the framework have to specialize them in order to provide specific implementations. The downloaded class may want to use operations that depend on the specific site (e.g. system calls); thus the local base class has to provide generic operations and the mobile code becomes a derived class containing methods that can exploit these generic operations.

Scenario 2 The site that downloads the class A for local execution may want to redefine some, possibly critical, operations that remote code may execute. This way access to some sensitive local resources is not granted to untrusted code (for example, some destructive "read" operations should be redefined as non-destructive ones in order to avoid that non-trusted code erases information). Thus the downloaded class A is seen, in this scenario, as a base class, that is locally specialized in a derived class B.

Summarizing, in **1** the base class is the local code while in **2** the base class is the mobile code. These scenarios are typical object-oriented compositions seen in a distributed mobile context. A major requirement is that composing local code with remote code should not affect existing code in a massive way. Namely, both components and client classes should not be modified nor recompiled.

Standard mechanisms of class extension and code specialization would solve these design problems only in a static and local context, but they do not scale well to a distributed context with mobile code. The standard inheritance operation is essentially static in that it fixes the inheritance hierarchy, i.e., it binds derived classes to their parent classes once for all. If such a hierarchy has to be changed, the program must be modified and then recompiled. This is quite unacceptable in a distributed mobile scenario, since it would be against its underlying dynamic nature. 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 go back and look in more details at the above scenarios. We could think of implementing a kind of dynamic inheritance for specifying at run-time the inheritance relation between classes without modifying their code. Such a technique could solve the difficulty raised by scenario **1**. However dynamic inheritance is not useful for solving scenario **2**, that would require a not so clear dynamic definition of the base class. Another solution would be releasing the requirement of not affecting the existing code, and allowing to modify the code of the local class (i.e. the local hierarchy). This could solve the second scenario, but not the first one that would require access to foreign source code. We are also convinced that the two scenarios should be dealt with by the same mechanism, allowing to dynamically use the same code in different environments, either as a base class for deriving new classes, or as derived class for being "adopted" by a

parent class. We remark that a solution based on *delegation* could help solving these problems. However delegation would destroy at least the dynamic binding and the reusability of the whole system [4].

Summarizing, mobile object-oriented code needs to be much more flexible than locally developed applications. To this aim we propose a new solution which is based on a mixin approach and we show that it enables to achieve the sought dynamic flexibility. Indeed, mixin-based inheritance is more oriented to the concept of "completion" than to that of extendibility/specialization. Mixins are incomplete class specifications, parameterized over superclasses, thus the inheritance relation between a derived and a base class is not established through a declaration (e.g., like `extends` in Java), instead it can be coordinated by the operation of *mixin application*, that takes place during the execution of a program, and it is not in its declaration part.

The novelty of our approach is the smooth integration of mobile code with mixins, a powerful tool for implementing reusable class hierarchies, that originated in a classical object-oriented setting as an alternative to class-based inheritance. The above examples hint that the usual class inheritance would not scale that harmoniously to the mobile and distributed context.

3 MoMi: Mobile Mixin Calculus

In this section we present the kernel calculus MoMi, aiming at coordinating distributed and mobile processes that exchange object-oriented code and reuse it in several execution environments. Following motivations discussed in the previous section, object-oriented code is structured via mixins. The mixin-based component is integrated with a core distributed and mobile calculus, where we abstract a few main features for representing distribution, communication and mobility of processes. This way MoMi is intended to represent a general framework for integrating object-oriented features in several calculi for mobility, such as, e.g., Klaim [12] and *DJoin* [16]. Before presenting our calculus we briefly recall the main features of the calculus of mixins of [6].

3.1 Object-Oriented Expressions

The object-oriented core of our language is based on the mixin calculus of [6], whose syntax is shown in Table 1. For the sake of clarity the syntax is slightly different from the one presented in [6], since some of its details are not necessary for our purposes. We recall here the features of the mixin calculus that are pertinent to MoMi.

$$
\begin{array}{rcl}
exp & ::= & x \mid \lambda x.exp \mid exp_1\ exp_2 \mid fix \\
& \mid & \mathsf{ref}\, x \mid \mathsf{deref}\, x \mid x := exp \\
& \mid & \{x_i = exp_i\}^{i \in I} \mid exp \Leftarrow x \mid \mathsf{new}\ exp \\
& \mid & \mathsf{class}\langle exp_g, [m_i]^{i \in Meth} \rangle \\
& \mid & \mathsf{mixin} \\
& & \quad \mathsf{method}\ m_j = exp_{m_j};^{(j \in New)} \\
& & \quad \mathsf{redefine}\ m_k = exp_{m_k};^{(k \in Redef)} \\
& & \quad \mathsf{constructor}\ exp_c; \\
& & \quad \mathsf{end} \\
& \mid & exp_1 \diamond exp_2
\end{array}
$$

Table 1. Syntax of the mixin core calculus.

The mixin calculus is fundamentally class-based and it takes a standard calculus of functions, records, and imperative features and adds new

constructs to support classes and mixins. It relies on the Wright-Felleisen idea of *store* [29], called *heap* here, in order to evaluate imperative side effects. There are four expressions involving classes: class, mixin, $\diamond$ (mixin application), and new. A class can be created by mixin application (via the $\diamond$ operator), and objects (that are records) can be created by class instantiation (using new). Finally, we define the root of the class hierarchy, class *Object*, as a predefined class. The root class is necessary so that all other classes can be treated uniformly, as it is the only class that is not obtained as a result of mixin application.

```
let A =                              let B =
  mixin                                mixin
    method m1 = ... n() ...              method n = ...
    redefine m2 = ... next() ...         method m2 = ...
    constructor c;                       constructor d;
  end                                  end
```

Table 2. Two mixins.

A mixin essentially is an abstract class that is parameterized over a super class. Let us consider the mixin A in Table 2. Each mixin consists of four parts: methods *defined* in the mixins, like m_1; *expected* methods, like n, that must be provided by the superclass, during the completion of the mixin; *redefined* methods, like m_2, where *next* can be used to access the implementation of m_2 in the superclass. a *constructor* that takes care of initializing the fields of the mixin (left implicit here).

If we now consider the mixin B in Table 2, then the application $A \diamond (B \diamond Object)$ will construct a class, which is a subclass of B, that can be instantiated and used:

let C = A $\diamond$ (B $\diamond$ *Object*) in (new C)$\Leftarrow m_1()$

An example follows in Table 3, to show how mixins work in practice (the syntax is further simplified in the example). We define a mixin `Encrypted` that implements encryption functionality on top of any stream class. Note that the class to which the mixin is applied may have more methods than expected by the mixin. For example, `Encrypted` can be applied to `Socket` $\diamond$ *Object*, even though `Socket` $\diamond$ *Object* has other methods besides *read* and *write*.

```
let FileStream =                     let Socket =
    mixin                                mixin
      method write = ...                   method write = ...
      method read = ...                    method read = ...
      ...                                  method hostname = ...
    end in                                 method portnumber = ...
                                           ...
                                         end in

let Encrypted =
    mixin
      redefine write = ... next (encrypt(data, key));
      redefine read = ... decrypt(next (), key);
      constructor (key, arg) = ...
    end in ...
```

Table 3. Example of mixin usage.

From the definition of `Encrypted`, the type system of the mixin calculus infers the constraints that must be satisfied by any class to which `Encrypted` is applied. The class must contain *write* and *read* methods whose types must be supertypes of those given to *write* and *read*, respectively, in the definition of `Encrypted`.

To create an encrypted stream class, one must apply the `Encrypted` mixin to an existing stream class. For example, `Encrypted` $\diamond$ `FileStream` $\diamond$ *Object* is an encrypted file class. The power of mixins can be seen when we apply `Encrypted` to a family of different streams. For example, we can construct `Encrypted` $\diamond$ `Socket` $\diamond$ *Object*, which

is a class that encrypts data communicated over a network. In addition to single inheritance, we can express many uses of multiple inheritance by applying more than one mixin to a class. For example, `PGPSign` $\diamond$ `UUEncode` $\diamond$ `Encrypt` $\diamond$ `Compress` $\diamond$ `FileStream` $\diamond$ *Object* produces a class of files that are compressed, then encrypted, then uuencoded, then signed.

3.2 MoMi syntax

We consider a very simple distributed calculus similar to CCS [22], enriched with localities. This is widely inspired by Klaim [12], in the sense that physical nodes are explicitly denoted as localities. The calculus is higher-order in that processes can be exchanged as first-entity data, and since we are interested in mobility of code this is a meaningful choice.

A node is denoted by its locality, ℓ, and by the processes P running on it. Informally, the semantics of $\mathsf{send}(P, \ell)$ is that of sending (the code of) process P to a process at locality ℓ waiting for it by means of a receive. This calculus is synchronous in that both sender and receiver are blocked until the communication occurs. However, switching to an asynchronous version would be straightforward.

The syntax of the MoMi calculus (Table 4) is obtained by combining this distributed calculus with the expressions of the mixin calculus: object-oriented expressions are seen as special processes. Process *exp* cannot be followed by any continuation process: this is a syntactic simplification that does not harm the expressive power, but it helps in having a simpler operational semantics (see Section 4). On the other hand, the construct $\mathsf{let}\ x = exp\ \mathsf{in}\ P$, which can only occur as last action in a process, allows to pass to the sub-process P the results of an object-oriented computation. The receive action specifies, together with the formal parameter name, the type of the expected actual parameter. In Section 5 a type system is introduced in order to assign a type to each well-behaved process. Only the free identifiers that are arguments of receives are explicitly typed by the programmer. In the processes $\mathsf{receive}(id : \tau).P$ and $\mathsf{let}\ x = exp\ \mathsf{in}\ P$, receive and let act as binders for, respectively, id and x in the process P.

$P ::=$	**nil**	(null process)
$\mid$	$a.P$	(action prefixing)
$\mid$	$P_1 \mid P_2$	(parallel comp.)
$\mid$	X	(process variable)
$\mid$	exp	(OO expression)
$\mid$	$\mathsf{let}\ x = exp\ \mathsf{in}\ P$	(let)
$a ::=$	$\mathsf{send}(P, \ell)$	(send)
$\mid$	$\mathsf{receive}(id : \tau)$	(receive)
$N ::=$	$\ell :: P$	(node)
$\mid$	$N_1 \parallel N_2$	(net composition)

Table 4. MoMi Syntax (see Table 1 for *exp* syntax).

3.3 Mixin Mobility in Action

We present in the following two simple examples showing mobility of mixins in action. They represent a *remote evaluation* and a *code-on-demand* [11] situation, respectively. Let us observe that both situations can be seen as examples of mobile agents as well. A more complex example is presented in Section 6.

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 will provide the specific implementation of the *print* method in the following way:

$$\begin{array}{l} \ell_1 :: \ldots \mid \mathsf{send}(my\_agent, \ell_2) \parallel \\ \ell_2 :: \ldots \mid \mathsf{receive}(mob\_agent : \mathtt{agent}). \\ \qquad \mathsf{let}\ PrinterAgent = mob\_agent \diamond printer \diamond Object\ \mathsf{in} \\ \qquad\quad (\mathsf{new}\ PrinterAgent) \Leftarrow start() \end{array}$$

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.

$$\begin{array}{l} \ell_1 :: \ldots \mid \mathsf{send}(my\_agent, \ell_2) \parallel \\ \ell_2 :: \ldots \mid \mathsf{receive}(mob\_agent : \mathtt{agent}). \\ \qquad \mathsf{let}\ RestrictedAgent = restricted \diamond mob\_agent \diamond Object\ \mathsf{in} \\ \qquad\quad (\mathsf{new}\ RestrictedAgent) \Leftarrow start() \end{array}$$

This example can also 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[1] (Example 2). Indeed, without any change to the code of the examples, one could also dynamically construct a class such as `restricted` $\diamond$ `mob_agent` $\diamond$ `printer` $\diamond$ *Object*. It is important to remark that in these examples we assume that the code sent (argument of send) and the code expected (argument of receive) are "compatible". This will be guaranteed by the type matching of the actual parameter and of the formal one in the communication rule (see Table 6).

4 Operational Semantics

The operational semantics of the MoMi calculus is centered around the distributed calculus, that allows distributed processes to communicate and exchange data (i.e. processes) by means of send and receive. The semantics of the object-oriented expressions is omitted here since it is basically the same of the one presented in [6]. The reduction of an *exp* is denoted by $\rightarrow$ (its closure is $\twoheadrightarrow$) and will produce an *answer* of the form $h.v$, where h is the heap obtained by evaluating the side effects present in *exp*, and v is the value obtained by evaluating *exp*.

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

$$
\begin{array}{c}
N_1 \parallel N_2 = N_2 \parallel N_1 \\
(N_1 \parallel N_2) \parallel N_3 = N_1 \parallel (N_2 \parallel N_3) \\
\ell :: P = \ell :: P \mid \mathbf{nil} \\
\ell :: (P_1 \mid P_2) = \ell :: P_1 \parallel \ell :: P_2
\end{array}
$$

Table 5. Congruence laws

The semantics for MoMi's 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 $\equiv$ (defined as the least congruence relation closed under the rules in Table 5) allows the rearrangement of the syntactic structure of a term so that reduction rules may be applied.

Reduction rules are displayed in Table 6. The crucial rule is (*comm*) that allows to communicate code among different sites. Code exchanged during a communication is a process that is not evaluated, and this is consistent with the higher-order nature of send and receive. The substitution $Q[P/id]$ is to be considered as a *name-capture-avoid substitution* (and so will be all substitutions from now on).

$$
\frac{\vdash \tau_1 <: \tau_2}{N \parallel \ell_1 :: \mathsf{send}(P^{\tau_1}, \ell_2).P' \parallel \ell_2 :: \mathsf{receive}(id : \tau_2).Q \rightarrowtail N \parallel \ell_1 :: P' \parallel \ell_2 :: Q[P^{\tau_1}/id]}\ (comm)
$$

$$
\frac{exp \twoheadrightarrow h.v}{N \parallel \ell :: exp \rightarrowtail N \parallel \ell :: \mathbf{nil}}\ (exp)
$$

$$
\frac{exp \twoheadrightarrow h.v}{N \parallel \ell :: \mathsf{let}\ x = exp\ \mathsf{in}\ P \rightarrowtail N \parallel \ell :: P[h.v/x]}\ (let)
$$

$$
\frac{N \equiv N_1 \qquad N_1 \rightarrowtail N_2 \qquad N_2 \equiv N'}{N \rightarrowtail N'}\ (net)
$$

Table 6. Distributed operational semantics

The key idea of this rule relies on the dynamic checking of the type of the actual parameter in order to guarantee a safe communication of code. Namely, the argument of a send is a process P annotated with its type τ_1, which is produced by the static analysis of process P[2]. In Section 5 we will present a type system that allows to check whether a process is typeable, so that only well-typed processes will be evaluated. Moreover, this static type analysis is assumed to produce an annotated version of the process to be evaluated, where every send's argument is annotated with its type. The (*comm*) rule uses this type information, delivered together with P, in order to dynamically check that the received item P is compliant with the formal argument (of type τ_2) by subtyping (as shown in Section 5.2). Conversely, the type τ_2 has been previously used to statically type check the continuation Q, on site ℓ_2, where id is possibly used. We would like to stress that, except for the dynamic checking required during the communication, type analysis of processes is totally static and performed in each site independently. Thus type safety of the communication results from

[2] Similarly, Java bytecode contains type information, used both by the classloader and the bytecode verifier.

the (static) type soundness of local and mobile code, with no need of further re-compilation and type-checking.

Finally, we require that a process, in order to be executed on a site, must be closed (i.e. without free variables), so it must be well-typed under $\Gamma = \emptyset$. It is easy to verify that if a process P is closed, for any $\mathsf{send}(P', \ell)$ occurring in P, the free variables of P' are bound by an outer let or by an outer receive. This implies that exchanged code is closed, as expected, when a send is executed.

Any time an *exp* is met (rules (*exp*) and (*let*)), this is reduced under the rules of the operational semantics of the mixin calculus. The rule for evaluating (*let*) says that *exp* is evaluated and then the resulting value is replaced for x in P. As a consequence, one can use the let construct to send evaluated code. In this case the code will be implicitly delivered together with its heap (containing the results of evaluating the side effects present in *exp*), according to reduction rules for object-oriented expressions as defined in [6] (see Section 3.1). Thus, all the heap references will be known also at the destination site. This is typical of mobile code and mobile agent systems, where the state is transmitted together with the code. Rule (*net*) is standard for the evolution of nets.

5 Typing

In this section we present the type system for the MoMi calculus. At this stage we are not interested in typing processes in details, so a process starting with an action will simply have the constant type action. A process like $\mathsf{receive}(X : \mathsf{action}).X$ means that it is willing to receive any process starting with an action and to execute it.

5.1 Type Rules

We extend the type assignment system of [6] with rules to type processes. Here we concentrate on the typing of processes, classes and mixins.

Type syntax is defined in Table 7, where ι is a constant type; $\rightarrow$ is the functional type operator; τ ref is the type of locations containing a value of type τ; $\{x_i : \tau_i\}^{i \in I}$ is a record type; and $I, J, K \subset \mathbb{N}$. A class type, $\mathsf{class}\langle \gamma, \sigma \rangle$ includes the type γ of the argument of the class generator, and the type σ of *self*, consisting of a record type $\{m_i : \tau_{m_i}\}^{i \in I}$. In class and mixin types, $\gamma\_$ is the type of the argument of a generator, and $\sigma\_$ is a record type. action is the above mentioned special constant type.

$$
\begin{array}{l}
\tau ::= \iota \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathsf{ref} \mid \{x_i : \tau_i\}^{i \in I} \\
\quad \mid \mathsf{class}\langle \gamma, \sigma \rangle \\
\quad \mid \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{exp}, \sigma_{old}, \sigma_{new}, \sigma_{red} \rangle \\
\quad \mid \mathsf{action}
\end{array}
$$

Table 7. Syntax of types.

Table 8 illustrates the shape of mixin types. Notice that mixin methods make typing assumptions about methods of the superclass to which the mixin will be applied. We refer to these types as *expected* types since the actual superclass methods may have different types. The exact relationship between the types expected by the mixin and the actual types of the superclass methods is formalized below in the rule for mixin application. We mark types that come from the

superclass with $\uparrow$ and those that will be redefined or added in the *mixin* (which acts as the subclass) with $\downarrow$.

$$\mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{exp}, \sigma_{old}, \sigma_{new}, \sigma_{red} \rangle$$

$$\begin{array}{ll} & \sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\}^{i \in I}, \sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\}^{k \in K}, \\ \text{where} & \sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\}^{j \in J}, \sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\}^{k \in K} \\ & m_i, \tau^{\uparrow}_{m_i}, \tau^{\uparrow}_{m_k} \text{ are inferred from method bodies} \end{array}$$

Table 8. The mixin type.

Both new and redefined methods in the mixin may call superclass methods (i.e. methods that are expected to be supported by any class to which the mixin will be applied). We refer to these methods as m_i. Their types $\tau^{\uparrow}_{m_i}$ are inferred from the mixin definition. The mixin type encodes the following information about the mixin:

- γ_b is the expected argument type of the superclass generator.
- γ_d is the argument type of the mixin generator.
- $\sigma_{exp} = \{m_i : \tau^{\uparrow}_{m_i}\}^{i \in I}$, $\sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\}^{k \in K}$ are the expected types of the methods that must be supported by any class to which the mixin is applied. 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, and $\tau^{\uparrow}_{m_k}$ are the types assumed for the old bodies of the methods redefined in the mixin.
- $\sigma_{new} = \{m_j : \tau^{\downarrow}_{m_j}\}^{j \in J}$, $\sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\}^{k \in K}$ are the types of mixin methods (new and redefined, respectively).

For further details we refer the reader to [6]. The type system of the calculus of mixins is extended with rules in Table 9. The rule (*send*) basically states that a process performing a send is well-typed if both its argument and the continuation are well typed. For typing a process performing a receive we type the continuation with the information about the type of *id* (rule (*receive*)). The form for rule (*let*) is standard (first the type of the *exp* is inferred and then P is typed considering this information). For parallel composition (rule (*comp*)) we require that both processes have the same type[3].

$$\frac{}{\Gamma, id : \tau \vdash id : \tau}\ (proj)$$

$$\frac{\Gamma \vdash P : \tau \quad \Gamma \vdash P' : \tau'}{\Gamma \vdash \mathsf{send}(P, \ell).P' : \mathsf{action}}\ (send) \qquad \frac{\Gamma, id : \tau \vdash P : \tau'}{\Gamma \vdash \mathsf{receive}(id : \tau).P : \mathsf{action}}\ (receive)$$

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

Table 9. Type rules for processes.

[3] At this stage, it is meaningful to consider only parallel composition of processes that perform actions, so we require both P_1 and P_2 to have an action type.

In order to facilitate the understanding of the type system of the MoMi calculus, we report in Table 10 the type rule for mixin application, taken from [6].

$$\frac{\begin{array}{l}\Gamma \vdash exp_1 : \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{exp}, \sigma_{old}, \sigma_{new}, \sigma_{red}\rangle \\ \Gamma \vdash exp_2 : \mathsf{class}\langle \gamma_c, \sigma_b\rangle \\ \Gamma \vdash \sigma_d <: \sigma_b <: (\sigma_{exp} \cup \sigma_{old}) \\ \Gamma \vdash \gamma_b <: \gamma_c\end{array}}{\Gamma \vdash exp_1 \diamond exp_2 : \mathsf{class}\langle \gamma_d, \sigma_d\rangle} \quad (mixin\ app)$$

$$\text{where} \quad \begin{array}{ll} \sigma_{exp} & = \{m_i : \tau^{\uparrow}_{m_i}\},\ \sigma_{old} = \{m_k : \tau^{\uparrow}_{m_k}\} \\ \sigma_{new} & = \{m_j : \tau^{\downarrow}_{m_j}\},\ \sigma_{red} = \{m_k : \tau^{\downarrow}_{m_k}\} \\ \sigma_b & = \{m_k : \tau_{m_k}, m_i : \tau_{m_i}, m_l : \tau_{m_l}\} \\ \sigma_d & = \{m_i : \tau_{m_i}, m_l : \tau_{m_l}, m_j : \tau^{\downarrow}_{m_j}, m_k : \tau^{\downarrow}_{m_k}\} \end{array}$$

Table 10. Rule for mixin application.

In the rule definition, σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. In particular, m_k are the superclass methods redefined by the mixin, m_i are the superclass methods called by the mixin methods but not redefined, and m_l are the superclass methods not mentioned in the mixin definition at all. Note that the superclass may have more methods than required by the mixin constraint.

Type σ_d contains the signatures of all methods supported by the subclass created as a result of mixin application. Methods $m_{i,l}$ are inherited directly from the superclass, methods m_k are redefined by the mixin, and methods m_j are the new methods added by the mixin. We are guaranteed that methods m_j are not present in the superclass by the construction of σ_b and σ_d: σ_d is defined so that it contains all the labels of σ_b plus labels m_j.

The premises of the rule are as follows:

- The $\sigma_d <: \sigma_b$ constraint requires that the types of the methods redefined by the mixin (m_k) be subtypes of the superclass methods with the same name. This ensures that all calls to the redefined methods in m_i and m_l (methods inherited intact from the superclass) are type-safe.
- The $\sigma_b <: (\sigma_{exp} \cup \sigma_{old})$ constraint requires that the actual types of the superclass methods m_i and m_k be subtypes of the expected types assumed when typing the mixin definition.
- The $\gamma_b <: \gamma_c$ constraint requires that the actual argument type of the superclass generator be a supertype of the type assumed when typing the mixin definition. Since class generators are functions, their argument types are in contravariant position, so this justifies the supertype requirement.

In the type of the class created as a result of mixin application, γ_d is the argument type of the generator, and σ_d (see above) is the type of objects that will be instantiated from the class.

Finally, let us remark that rules of Tables 9 and 10 are syntax-driven, so they can define an algorithm for deciding whether a given process P, on a site

ℓ, is typeable. In particular, since we require P to be closed, then typability of P means that $\emptyset \vdash P : \tau$, where each subterm of P is assigned a type (even when τ is action). Thus the reconstruction of the deduction $\emptyset \vdash P : \tau$ allows to statically decorate any send's argument, occurring in P, by its type, as required by (*comm*) rule in the semantics (Table 6). For instance, let $x = exp$ in $\mathsf{send}(x, \ell)$ has type action, and its compiled version is let $x = exp$ in $\mathsf{send}(x^{\tau_1}, \ell)$ if exp has type τ_1.

5.2 Subtyping Relation

The main novelty of the MoMi type system is the extension of the subtyping relation to class and mixin types. In [6], subtyping exists only at the object level, to keep the inheritance and the subtyping hierarchies completely separated. Here, the key idea is to deal with classes and mixins as polymorphic entities that are exchanged among distributed sites. So we extend the subtyping relation to classes and mixins, in order to achieve more flexibility in the communication. Observe that, from the formal point of view, we have chosen to use subtyping in type matching at communication time, instead of explicitly define a *subsumption* rule in the type system. Namely, any term of type τ is implicitly assumed to have also any type greater than τ. So, in particular, in the rule (*comm*), the formal parameter of a receive, which is explicitly typed, matches with any received item whose type is a subtype of the one expected.

The starting point is the basic system of subtyping for arrow and ref types and other standard subtyping rules (they can be found in [6]). Concerning record types, we use the standard width subtyping. This is not a crucial simplification[4] with respect to the more complete width-depth subtyping, which would require more technicalities to be dealt with, technicalities that are not within the purpose of this paper. Width-depth subtyping on records is introduced in a forthcoming foundational work on MoMi's type system.

The subtype relation concerning mixins and classes is in Table 11. In the rule ($<:$ *mixin*) the subtype can define more new methods and require less methods, but it cannot override more methods ($|\sigma|$ is the number of methods in σ); the contravariance of the mixin (subclass) generator parameter is as expected ($\gamma_d <: \gamma'_d$), while for the superclass generator parameter covariance is required ($\gamma'_b <: \gamma_b$).

$$\frac{\Gamma \vdash \gamma <: \gamma' \quad \Gamma \vdash \sigma' <: \sigma}{\Gamma \vdash \mathsf{class}\langle \gamma', \sigma' \rangle <: \mathsf{class}\langle \gamma, \sigma \rangle} \quad (<: \ class)$$

$$\frac{\begin{array}{ccc} \Gamma \vdash \sigma'_{new} <: \sigma_{new} & |\sigma'_{red}| = |\sigma_{red}| & \Gamma \vdash \sigma'_{red} <: \sigma_{red} \\ \Gamma \vdash \sigma_{exp} <: \sigma'_{exp} & |\sigma_{old}| = |\sigma'_{old}| & \Gamma \vdash \sigma_{old} <: \sigma'_{old} \\ \multicolumn{3}{c}{\Gamma \vdash \gamma'_b <: \gamma_b \qquad \Gamma \vdash \gamma_d <: \gamma'_d} \end{array}}{\Gamma \vdash \mathsf{mixin}\langle \gamma'_b, \gamma'_d, \sigma'_{exp}, \sigma'_{old}, \sigma'_{new}, \sigma'_{red} \rangle <: \mathsf{mixin}\langle \gamma_b, \gamma_d, \sigma_{exp}, \sigma_{old}, \sigma_{new}, \sigma_{red} \rangle} \quad (<: \ mixin)$$

Table 11. Subtype relation for classes and mixins.

[4] This is typical, for instance, of popular languages such as C++ and Java.

The new subtype relation on mixins is consistent with $<:$ constraints of the rule for mixin application (Table 10); thus the type system guarantees that what is statically type-checked on a site can be communicated to a different site without producing run-time errors when executed, as long as the above constraints are respected. Conversely, local code remains well typed even when remote code is merged in it, via a well typed communication. Thus, polymorphism by subtyping for classes and mixins guarantees type-safe communications, in the sense that errors like "message-not-understood" cannot occur, without requiring to type check neither the whole system nor the local code again.

6 A scenario for mixin mobility

We will use here a slightly simplified syntax: (i) we will list the methods' parameters in between "()" instead of using explicit λ-abstractions; (ii) $exp_1; exp_2$ is interpreted as $(\lambda x.exp_2)\,exp_1,\ x \notin FV(exp_2)$, in a call-by-value semantics.

The example is about a client and a server, executing on two different nodes, that want to communicate, e.g., by means of a common protocol. They both use a *Socket* to this aim, however the server is willing to abstract from the implementation of such socket, by allowing the client to provide a custom implementation. This can be useful, for instance, because the client may decide to use a customized socket; in this example the client implements a socket that sends and receives compressed data (alternatively it could implement a *multicast* socket, or even a combination of the two). However, the code sent by the client may rely on some low-level system calls, that may be different on the server's site: indeed, the two sites may run different operating systems and have different architectures. These low-level system calls are then to be provided by each site (the client's and the server's sites). The customized socket of the client is then a mixin requiring the existence of such system calls, that will be provided by two different (yet compliant) superclasses, one resident on each site. The code executed in the two nodes (`client` and `server`) is in Listing 1.

Both `ZipSocket` and `Socket` rely on a superclass that provides (at least) methods `write_to_net` and `read_from_net`. The client, in its site, completes its mixin `ZipSocket` with `NetChannel` that provides these two methods for writing data on the net, by using its operating system low-level system calls. Sending the class `ZipSocket` $\diamond$ `NetChannel` directly to the server may be nonsense, since the server may use a different operating system (or a different version of the same operating system). Instead, only the mixin `ZipSocket` is sent to the remote server. In the server this mixin will be received as a `Socket` mixin (and this succeeds since `ZipSocket` $<:$ `Socket`) and it will be completed with `NetFile`, which corresponds to the `NetChannel` of the client. The server will then use such socket independently from the particular client's implementation. Notice that the use of subtyping in the communication (instead of a simpler type equality) completely relieves the receiver (and especially its programmer) from the real complete interface of the clients' code.

Let us now consider an alternative implementation of the same scenario, in order to show other features of MoMi: suppose that on the client `ZipSocket`

```
client:: let ZipSocket =
  mixin
    method zip = ...
    method unzip = ...
    method write = write_to_net(zip(data))
    method read  = unzip(read_from_net())
  end in
let NetChannel =
  mixin
    method send =
      // <send through the net>
    method receive =
      // <receive from the net>
    method write_to_net = send(data)
    method read_from_net = receive()
  end in
let channel = ref new
    (ZipSocket ◇ (NetChannel ◇ Object)) in
  (
    send( ZipSocket, server ).
    ( (deref channel)⇐write("hello") ;
      (deref channel)⇐read() )
  )
```

```
server:: let Socket =
  mixin
    method write = write_to_net(data)
    method read  = read_from_net()
  end in
let NetFile =
  mixin
    method write_to_net =
      // <send through the net>
    method read_from_net =
      // <receive from the net>
  end in
(
  receive( sock : Socket ).
  let client_channel = ref new
      (sock ◇ (NetFile ◇ Object)) in
  (
    (deref client_channel)⇐read() ;
    (deref client_channel)⇐write("welcome")
  )
)
```

Listing 1: Example code for client and server communication.

is written like in Listing 2 on the left. In this case the class does not rely on `write_to_net` and `read_from_net` (instead it expects the superclass to provide methods `write` and `read` that the mixin redefines), and thus it is not a subtype of `Socket` in the server. In the server, the code would be like in Listing 2 on the right. Since also `Channel` relies on a super class that provides `write` and `read`, we have that `ZipSocket` <: `Channel`. So the server receives a `ZipSocket` (as a `Channel`) that it completes with `Socket` completed, in turn, with `NetFile`.

```
ZipSocket =
  mixin
    method zip = ...
    method unzip = ...
    redefine write = next(zip(data))
    redefine read  = unzip(next())
  end
```

```
Channel =
  mixin
    redefine write = next(data)
    redefine read  = next()
  end
  ...
  receive( chan : Channel ).
  let client_channel =
    ref new (chan ◇
      (Socket ◇ (NetFile ◇ Object))) in
  (
    (deref client_channel)⇐read() ;
    (deref client_channel)⇐write("welcome")
  )
```

Listing 2: An alternative implementation.

Finally, as hinted in Section 3.1, other implementations of such a socket can be created, simply by using more than one mixin, such as `UUEncode`, `Encrypt`, and so on.

7 Conclusions and Related Work

In the literature, there are several proposals of combining objects with processes and/or mobile agents. *Obliq* [9] is a lexically-scoped 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. Other works, such as, e.g., [5, 24, 18] do not deal explicitly with mobile distributed code. In our calculus no remote method call functionality is considered, and, instead of formalizing remote procedure calls (like most of the above mentioned approaches), MoMi provides the introduction of safe and scalable distribution of object-oriented code, in a calculus where communication and coordination facilities are already provided.

MoMi results from the integration of two calculi, a simple coordination language for mobile code and the mixin calculus of [6]. Since it looks like in MoMi the two core calculi cooperate quite smoothly, this gives us some confidence about the modularity of this approach, so this is meant as a first step towards a general framework to experiment the mixin-based approach also with other different mobile calculi, such as the *DJoin* [16], the Ambient Calculus [10], and in particular KLAIM [12]. The type system of MoMi is currently being extended and completely formalized in order to prove main properties such as subject-reduction and type safety.

We are also designing a mixin-oriented version of KLAIM (and we are planning to extend its implementation, X-KLAIM [3], along the same line). The type system presented in this paper can be modified in order to refine action types, so that finer types can be assigned to processes, e.g. according to the capability-based type system for access control developed for KLAIM [13].

Acknowledgments We thank Rocco De Nicola and the anonymous referees for helpful suggestions. We are very grateful to Luca Cardelli for pointing out crucial questions in the early version of the paper.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.
2. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.
3. L. Bettini, 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)*, pages 110–115. IEEE Computer Society Press, 1998.
4. L. Bettini, M. Loreti, and B. Venneri. On Multiple Inheritance in Java. In *Proc. of TOOLS EASTERN EUROPE, Emerging Technologies, Emerging Markets*, 2002. To appear.
5. P. D. Blasio and K. Fisher. A Calculus for Concurrent Objects. In U. Montanari and V. Sassone, editors, *CONCUR '96: Concurrency Theory, 7th Int. Conf.*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996.
6. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In R. Guerraoui, editor, *Proceedings ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.
7. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

8. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
9. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
10. L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, number 1378 in LNCS, pages 140–155. Springer, 1998.
11. A. Carzaniga, 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)*, pages 22–33. ACM Press, 1997.
12. 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.
13. R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.
14. R. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ICFP '98*, pages 94–104, 1998.
15. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
16. C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
17. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
18. A. D. Gordon 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*, volume 16.3 of *ENTCS*. Elsevier, 1998.
19. C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.
20. F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verification of Multiple - Agent Languages*, number 1192 in LNCS. Springer-Verlag, 1996.
21. M. V. Limberghen and T. Mens. Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3(1):1–30, 1996.
22. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
23. Object Management Group. Corba: Architecture and specification. `http://www.omg.org`, 1998.
24. B. C. Pierce 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)*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
25. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. ECOOP '98*, pages 550–570, 1998.
26. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.
27. M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proc. OOPSLA '96*, pages 359–369, 1996.
28. J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents.* AAAI Press and MIT Press, 1996.
29. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.