A Java package for class and mixin mobility in a distributed setting*

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
bettini@dsi.unifi.it

Abstract. Mixins, i.e., classes parameterized over the superclass, can be the right mechanism for dynamically assembling class hierarchies with mobile code downloaded from different sources. In this paper we present the Java package `momi` that implements the concepts of the language MOMI, which is a calculus for exchanging mobile object-oriented code structured through mixin inheritance. This package can be thought of as an "assembly" language that should be the target of a compiler for a mobile code object-oriented language. In order to show an usage of the package, we illustrate how it is used by the compiler of X-KLAIM, a programming language for mobile code where distributed processes can exchange classes and mixins through distributed tuple spaces.

1 Introduction

Mixins [11, 17] 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. This minimizes the dependences between superclass and its subclasses, as well as between class implementors and end-users, thus improving modularity.

Due to their dynamic nature, mixin inheritance can be fruitfully used in a *mobile code* setting [25, 14], where distributed applications exchange code through the network to be dynamically integrated in the running processes. In [5], we introduced MOMI (Mobile Mixins), a coordination language for mobile processes that exchange object-oriented code. The underlying idea 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 has to 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*).

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

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

(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. If the code is successfully accepted, it can interact with the local code in a safe way (i.e., no run-time errors such as "message-not-understood") without requiring any further type checking of the whole code. 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 failure communication to the sender will ever be required). This makes the code exchange an *atomic* action.

In this paper we present the implementation of MOMI in Java that consists in a package `momi`. 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 MOMI abstracts 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. A concrete example of how to integrate `momi` within a real mobile code language will be shown in Section 4 by illustrating how the package `momi` is used within X-KLAIM [7, 8, 2], a tuplespace based programming language for mobile code.

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 MOMI 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 according to MOMI types. 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).

At the best of our knowledge, this is the first approach that employs mixins in a mobile code environment for flexible and safe code exchange. This is the distinguishing feature w.r.t. to other mixin-based languages and implementations, such as, e.g., [11, 17, 1]. Other works, such as, e.g., [16, 24, 21, 18] are concerned in merging concurrency and object orientation and do not deal explicitly with mobile distributed code, while other OO mobility languages such as [13, 12] do not treat dynamic inheritance. Due to lack of space we will not provide the formal description of MOMI (we refer the interested reader to [5, 3]) and we give only a brief description of the `momi` package; its full presentation can be found in [2]. All the software presented here is freely available at `http://music.dsi.unifi.it`.

2 Main features of the package

Since `momi` is thought of as an assembly language that is target of a compiler for a high level programming language, in order to describe the package we will use a very simple object-oriented programming language (based on the syntax of X-KLAIM). We will not give the formal syntax of this language, since it should be quite intuitive. Let us describe

informally mixin usage through a tutorial example (for simplicity, instance variables are considered *private*, methods are considered *public* and no method overloading is employed):

```
mixin M
  expect n() : int;
  redef m2(a : int) : str ...
  def m1(b : bool) : int
end

class C
  n() : int ...
  m2(a : int) : str ...
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 mixin application $M <> C$ constructs a class, which is a subclass of C. This operation is type correct only if the class C provides all the methods requested by the mixin (i.e., n and m_2). The type checker will also guarantee that there is no conflict among methods defined by the mixin and methods defined by the class. Mixin types encode the information about the types of mixin methods that are defined, redefined and expected.

In a mixin application class and mixin expressions can also be class and mixin variables that will be replaced at run-time with actual classes and mixins downloaded from the network. In this crucial case, it is safe to replace these variables with classes and mixins that have a subtype w.r.t. the corresponding types of variables. A class C' is a *subtype* of a class C if the type of C' has *at least* the methods of C. A mixin M' is a subtype of a mixin M if the type of M' has *at least* the methods *defined* by M, *at most* the methods *expected* by M and *the same* methods *redefined* by M. For objects, the subtype relation is the same as for classes. For motivations about these subtype relations we refer to [5]. Notice that we consider only *width* subtyping, i.e., the methods with the same name must have the same type, not a subtype; for a formal treatment of *depth* subtyping in MOMI we refer to [6]. Finally, due to lack of space, we do not consider here possible run-time name clashes among methods *defined* by the mixin and also by the class in a mixin application (for such methods we use static binding instead of dynamic binding as illustrated in [2]).

The package `momi` provides means for creating new classes by applying a mixin to an existing class. This is similar to the task performed by a compiler for an object-oriented language when a class derives from a base class, but the main difference is that this operation will take place at run-time. The main aim of the package is that of providing this mechanism in a transparent way: once a class is generated after a mixin application it can be used to generate objects, but it can also take part in other mixin applications as a base class. The objects created through class instantiation are fully fledged and, so, ready for method invocations.

The crucial feature of MOMI is that classes and mixins are themselves mobile code, i.e., code that can be dynamically downloaded from a remote source. This implies that, at mixin application time, the actual implementation of mixins and classes may differ from their expected (static) interface, provided that the former is subtyping-compliant with the latter in order to guarantee the absence of run-time errors. This highlights the main difference between our approach and other mixin based languages in the literature (see, e.g., [17, 1]): in a mixin-based language, when the mixin application takes place

in a specific part of a program, both the code of the mixin and the one of the classes are available for the compiler. This does not hold in MOMI since mixin application can act also on mixin and class (polymorphic) variables.

So the task of generating a new class by applying a mixin to a class must be done in MOMI at run-time, when the class and mixin variables have been actually instantiated dynamically with effective code (possibly downloaded from the network). Basically this is the same difference that distinguishes a language with dynamic class loading, such as Java, and one with static loading, such as C++. This is similar to what happens in calculi where classes and mixins are "first-class citizens", i.e., they can be passed as parameters to functions (see, for example, [10]). It is important to notice, though, that in such calculi the matching between actual and formal parameters is always based on equality (at least at the best of our knowledge), because the burden of checking extra constraints at parameter-passing time, to avoid inheritance/subtyping conflicts, is not worthwhile in a sequential scenario. In a mobile distributed scenario, instead, where flexibility is a must because the nature of the downloaded code cannot always be estimated a priori, the use of such extra subtyping constraints at communication time is a small price to pay, especially since this allows to combine local and foreign code without any recompilation.

Dynamic type checking is minimized (i.e., only during the exchange of code); thus, in implementing the package `momi`, a particular importance was given to this matter: the actual content (the methods) of classes and mixins is only examined during mixin application in order to make any method call on objects completely untyped. This reflects, as we will remark during the description of objects and methods, in the fact that no further type-checking is performed when invoking a method on an object (and when performing casts in the Java-code corresponding to MOMI method bodies), and it is also consistent with the idea of having a virtual machine, that basically executes untyped code as in an assembly language. Of course this relies on the assumption that the compiler generating code that uses `momi` is correct. How to have a formal proof of such a soundness property may be the subject of future studies.

3 The main classes of the package

The package `momi` contains the classes that abstract the concepts of the MOMI language: objects, mixins, classes, types and methods. Indeed also methods are low level structures in the package: they are manipulated in order to assemble new classes at run-time and to build object method tables. Given a class *C* (resp. a mixin *M*) in the high level language, the compiler is supposed to generate a subclass of `MoMiClass` (resp. `Mixin`), a subclass of `MoMiObject` and a subclass of `MoMiMethod` for each methods defined in the class *C* (resp. *defined*, *redefined* and *expected* in the mixin *M*). All the functionalities provided by the package that concern manipulation of these generated classes (i.e., object instantiation and mixin application) are based on the assumption that the compiler has already successfully type-checked the source language.

The class `MoMiMethod` represents a method defined in a MOMI class or mixin and it is the base class from which all the generated methods have to derive. Every `MoMiMethod` owns a private stack where parameters are pushed before calling the method, popped from within the method body, and where the return value is pushed from within the method. Classes derived from `MoMiMethod` have to implement the Java method `invoke`. This method can throw a `NoSuchMethodException` in case of a "message-not-understood" error; however, this exception should never be raised as long as the compiler produces correct code. Consistently with standard object-oriented

language implementations (see, e.g., [23, 22]) the pointer to the current target object `self` (or `this`) is passed to the method at invocation time.

Thus, the steps for calling a method consist in pushing the parameters on the stack of the method (`pass_argument`), invoking the method passing the object on which the method is called and, in case, retrieving the result from the stack (`get_result`). These basic instructions are to be generated by the compiler, that also has the responsibility of pushing the parameters on the stack in the right order and of trying to retrieve the result only if the method returns a value (in this case it has also to cast the returned value to the right type).

Example 1. Let us assume that we have to call a method that takes as argument an integer and a string (passing the integer `100` and the string `"foo"`) and returns a boolean. The code that the compiler should generate, if the reference to the method is stored in `meth` (how to obtain such a reference and how to pass the "right" `self` pointer will be shown in the following), is similar to the following one:

```
meth.pass_argument(new Integer( 100 ));
meth.pass_argument(new String( "foo" ));
meth.invoke(_pass_self);
b = (Boolean) meth.get_result() ;
```

The body of `invoke` has to be generated accordingly: it must pop the arguments from the stack and assign them to the parameters, and cast the pointer `self` to the actual type. A return statement will be translated to a push of the value in the stack followed by an exit statement for leaving the method (i.e., a standard Java `return`).

Example 2. Going back to the previous example, let's assume that the body of the method we are calling is as follows

```
m(i : int, s : str) : bool
begin
 if (i < 10) then
  return i > my_field # my_field is an instance variable
 endif;
 return i >= 0
end
```

The compiler should produce code similar to the following one:

```
public void invoke(MoMiObject _self) throws NoSuchMethodException {
   MyObject self = (MyObject) _self;
   String s = (String) stack.pop();
   Integer i = (Integer) stack.pop();

   if ( i.intValue() < 10 ) {
      stack.push( new Boolean( i.intValue() > self.my_field.intValue() ) );
      return ;
   }
   stack.push( new Boolean( i.intValue() >= 0 ) );
   return ;
}
```

Indeed the `self` pointer is casted to the actual (expected) type (say `MyObject`), and the arguments are popped from the stack and assigned to the formal parameters (once again after casting). Notice how the `my_field` instance variable is explicitly prefixed with the object `self` in the generated code.

If the compiler for our language correctly type-checked the source program, the type casts in the generated code will be type-safe.

The base class for all MOMI objects is `MoMiObject`. The derived classes only have to add the fields (instance variables) declared in the class this object is instantiated from, and this is basically the only task the compiler has to perform when generating code for MOMI objects. The class `MoMiObject` relies on the base class `WithMethods` that includes a table of `MoMiMethods`, `methods`, i.e., an `Hashtable`, where the key is the method name (the untyped nature of the package is proved by methods being searched only by name and not by their types). Such table corresponds to the table used for method dynamic binding in C++ and Java. `WithMethods` is also the base class for `Mixin` and `MoMiClass` representing, respectively, MOMI mixins and classes (shown later).

An object contains also the structure of objects of superclasses. In languages such as C++, this boils down to memory offset careful management: the instance variables of an object of a derived class start, in the memory layout of the object, where the instance variables of an object of the parent class end. This enables objects of derived classes to be used in place of objects of superclasses. Once again, in such languages, this management can be performed statically by the compiler since the structure of the subclass and of the superclass are available at compile time.

This cannot be done in MOMI since the structures of objects are known only at run-time. Thus the relation between the instance variables of the mixin (subclass) and those of the superclass has to be established dynamically through a reference; this reference is represented by the field `next` in the `MoMiObject` class. This is also consistent with the "compositional" nature of mixins: mixin-based inheritance is more similar to object composition, than to object extension. Thus, when an object of the derived class has to be passed to a method inherited by the superclass (i.e., a method expected by a mixin, or the inherited implementation of a redefined method), the object pointed to by `next` has to be passed, instead of `self`. This is a crucial matter, since a method can access the fields of the `self` object and indeed it expects an object of the class where the method is defined (notice the cast to `MyObject` in Example 2).

When the `self` pointer has to be explicitly passed at method invocation time, the "right" `self` has to be passed, i.e., the one that is instance of the class the called method is defined in. Since dynamic binding is used for method invocation, the "right" `self` depends on the specific method version that is called. That is why we need to use, apart from the method table, also a *self table* indexed by method names. Thus, first the reference to the method to be called has to be retrieved by using `get_method` and then the pointer to the right `self` has to be retrieved by using `get_self`. The compiler has not to take care of building the `self_table`: this task is performed during object creation by the package `momi`.

The technique shown above allows to implement dynamic binding for method invocation: in particular, if the method that is being called is an expected method, then the implementation provided by the base class will be called, passing the self inherited from the base class. If a method of the base class is calling a method that has been redefined by the mixin, then the new implementation is called, passing as self the object of the derived class.

Example 3. If the method `m` of the Example 1 is invoked on the object `obj` the complete invocation sentence is as follows:

```
meth = obj.get_method("m");
_pass_self = obj.get_self("m");
meth.pass_argument(new Integer( 100 ));
```

```
meth.pass_argument(new String( "foo" ));
meth.invoke(_pass_self);
b = (Boolean) meth.get_result() ;
```

`_pass_self` may be different from `obj`: if `m` is an expected method, then `_pass_self` will be a `self` contained in `obj` (since such method is inherited from the superclass); alternatively, if `m` is a redefined method and it is being invoked from a method of a superclass, since dynamic binding is employed, `_pass_self` will be an object actually containing `obj`.

The class `Mixin` of the package `momi` is the most complex one since it provides means for performing the mixin application operation and for performing object instantiation. Notice that this class will also be the base class for `MoMiClass`; this makes sense from the design point of view, since a class can be seen as a mixin where there are no expected methods and no redefinitions.

The crucial part of `Mixin` is the (static) method `apply` that applies a mixin to a class and returns a new derived class. This method basically copies in the new class' method table all the methods defined and redefined by the mixin and all the methods expected by the mixin taking them from the superclass. The method `apply` only creates a new `MoMiClass` object, with the most specialized version of each method, but it does not perform any operation concerning dynamic and static bindings: indeed, this will be performed at object instantiation time, carried on by the method `new_instance`. First of all this method calls a *factory method* [19], `_new_instance`, that classes deriving from `Mixin` and `MoMiClass` have to redefine in order to return an instance of the corresponding class deriving from `MoMiObject`. Finally, the `self` tables of the new created object is initialized.

A crucial feature of MOMI is that object-oriented values (i.e., objects, classes and mixins) are explicitly typed when they are exchanged among distributed sites. The compiler has to statically decorate these values when they are sent to a remote site. This annotation will consist in inserting the statically built type in a message containing an object-oriented value. The package `momi` supplies the classes representing the types for the following items: methods, objects, classes and mixins, apart from basic types (that depends on the run-time systems). The interface for MOMI types is `MoMiType`, containing methods `equals` and `subtype`.

The class `RecordType` stores method types in an hashtable, and performs comparison on record types, not considering the order of method types. `ObjectType` and `ClassType` basically rely on this class for performing comparisons. The class for mixin types, `MixinType` keeps a different record type for defined, expected and redefined methods. The comparison is consistent with the subtyping rule presented at the beginning of Section 2: for redefined methods the equality is required, while for expected methods the subtyping relation is inverted. Let us observe that in the previously described classes all these classes for types are never used. This shows the untyped nature of the run-time environment. Types will be used only during the communication.

4 Object-Oriented mobility in X-KLAIM and KLAVA

X-KLAIM [7, 8, 2] is a mobile code programming language where communication takes place through multiple tuple spaces (as in Linda [20]) distributed on sites; sites are accessible through their localities (e.g., IP addresses). The reserved locality `self` can be used to access the local execution site (in order to avoid confusion with the object-oriented `self`, we use `this` in object-oriented expressions instead of `self`). A tuple *t*

```
public class MyMixin extends Mixin {
  public MyMixin() {
    set_method("myinit", new MyMixin_myinit());
    set_method("print_fields", new MyMixin_print_fields());
    set_method("init", new MoMiMethod("init", MoMiMethod.EXPECTED));
    set_method("get_i", new MoMiMethod("get_i", MoMiMethod.EXPECTED));
  }

  protected MoMiObject _new_instance() { return new MyMixinObject(); }

  public static MixinType create_type() {
    MixinType new_type = new MixinType();
    new_type.addMethod(MyMixin_myinit.create_type(), MoMiMethod.DEFINED);
    new_type.addMethod(MyMixin_print_fields.create_type(), MoMiMethod.REDEFINED);
    new_type.addMethod(MyMixin_init.create_type(), MoMiMethod.EXPECTED);
    new_type.addMethod(MyMixin_get_i.create_type(), MoMiMethod.EXPECTED);
    return new_type;
  }
}
```

Listing 1: The Java class generated by the compiler for the mixin `MyMixin`.

can be inserted in a tuple space located at locality ℓ with the operation **out**$(t)@\ell$ and removed (resp. read) with **in** (resp. **read**). Pattern matching is used for selecting tuples. X-KLAIM programs are compiled into Java programs that use the package KLAVA [9] that provides the run-time system for X-KLAIM operations. Both X-KLAIM and KLAVA have been extended in [2] in order to use `momi` for object-oriented code mobility. In particular, if x is a class variable, the operation **in**$(!x)@\ell$ retrieves a tuple from ℓ containing a class with a subtype of x. X-KLAIM is based on the kernel language KLAIM [15]; the formal extension of the KLAIM model with the MOMI features is presented in [4].

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 relies 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 typing system defined by MOMI. Let us now consider a simple mixin definition:

```
mixin MyMixin
  expect init(a : int, b : str);
  expect get_i() : int;
  redef print_fields() begin ... end;
  def myinit(x : int, y : str) begin ... end
end
```

that expects from the super class two methods, redefines one, and defines a new method. The Java class generated by the X-KLAIM compiler for this mixin definition is shown in Listing 1. For each method of the mixin, a subclass of `MoMiMethod` is generated by the compiler. The generated constructor of `MyMixin` builds the method table. Notice that, for each expected method, objects of the base class `MoMiMethod` are inserted. Instead, the generated classes for expected methods are only useful for retrieving the types of these methods. The compiler generates an appropriate `create_type` method that allows to retrieve the static type (e.g., for decorating mobile code that has to be sent to a remote site). For instance, concerning `MoMiMethods`, such a type will contain the entire method signature, while for mixins it will contain three record types. Classes generated for `MyMixinObject` and the other methods are similar.

```
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 2: The printer agent example.

4.1 A mobile printer agent in X-KLAIM

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 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 to the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 2, 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 `MyPrintedAgent` 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).

Acknowledgments I would like to thank the two co-authors of MoMi, Viviana Bono and Betti Venneri. The anonymous referees provided helpful suggestions for clearing up some aspects in the paper.

References

1. D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP 2000*, number 1850 in LNCS, pages 145–178, 2000.
2. 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`.
3. L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Manuscript.
4. L. Bettini, V. Bono, and B. Venneri. O'KLAIM: a coordination language with mobile mixins. In *Proc. of Coordination 2004*. Springer. To appear in LNCS.
5. L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, number 2315 in LNCS, pages 56–71. Springer, 2002.
6. L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Classes and Mixins. In *FOOL 10*, 2003.
7. L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In *Proc. of the 7th IEEE WETICE*, pages 110–115. IEEE Computer Society Press, 1998.
8. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
9. 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.
10. V. Bono, A. Patel, and V. Shmatikov. A Core Calculus of Classes and Mixins. In *Proc. ECOOP'99*, number 1628 in LCNS, pages 43–66. Springer-Verlag, 1999.
11. G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.
12. M. Bugliesi and G. Castagna. Mobile Objects. In *Proc. of FOOL*, 2000.
13. L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
14. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. ICSE '97*, pages 22–33. ACM Press, 1997.
15. 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.
16. P. Di Blasio and K. Fisher. A Calculus for Concurrent Objects. In *CONCUR '96*, volume 1119 of *LNCS*, pages 655–670. Springer, 1996.
17. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.
18. C. Fournet, C. Laneve, L. Maranget, and D. Rémy. Inheritance in the Join Calculus. In *FSTTCS 2000*, volume 1974 of *LNCS*, pages 397–408. Springer, 2000.
19. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
20. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
21. A. Gordon and P. Hankin. A Concurrent Object Calculus: Reduction and Typing. In *Proc. HLCL '98*, volume 16.3 of *ENTCS*. Elsevier, 1998.
22. T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
23. S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.
24. B. C. Pierce and D. N. Turner. Concurrent Objects in a Process Calculus. In *Proc. TPPP 94*, volume 907 of *LNCS*, pages 187–215. Springer, 1995.
25. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.