X-KLAIM: a Programming Language for Object-Oriented Mobile Code. User's manual

Lorenzo Bettini

Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
http://www.lorenzobettini.it

July 1, 2004

Contents

1 The programming language X-KLAIM

X-KLAIM (*eXtended* KLAIM) is an experimental programming language specifically designed to program distributed systems composed of several components interacting through multiple tuple spaces and mobile code (possibly object-oriented). It is based on the kernel language KLAIM (*Kernel Language for Agent Interaction and Mobility*) [De Nicola *et al.*, 1998]. X-KLAIM extends KLAIM with a high level syntax for processes: it provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition and object-oriented features based on mixin inheritance. The Java package KLAVA (originally presented in [Bettini *et al.*, 2002c]; see the KLAVA user's manual [Bettini, 2003b]) provides the run-time system for X-KLAIM operations, and a compiler translates X-KLAIM programs into Java programs that use KLAVA. The structure of such framework is depicted in Figure 1. This document is based on chapters of [Bettini, 2003a].

Before describing the basic concepts of X-KLAIM we give a very brief introduction to KLAIM (we refer the interested reader to [De Nicola *et al.*, 1998] and to the KLAIM web page, `http://music.dsi.unifi.it`, for more complete descriptions of the formal model).

KLAIM is based on the notion of *locality* and relies on a Linda-like communication model. Linda [Carriero & Gelernter, 1989b; Gelernter, 1985; Gelernter, 1989] is a coordination language with asynchronous communication and shared memory. The shared space is named *tuple space*, 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.

Tuples are anonymous and content-addressable. *Pattern-matching* is used to select tuples in a tuple space: two tuples match if they have the same number of fields and corresponding fields 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 $(\text{“foo”}, \text{“bar”}, 100 + 200)$ matches with $(\text{“foo”}, \text{“bar”}, !Val)$. 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.

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* (or *sites*), which are part of a *net*. Each node contains a single tuple space and processes in execution, and can be accessed through its *locality*. There are two kinds of localities: *physical localities* are the identifiers through which nodes can be uniquely identified within a net; *logical localities* are symbolic names for nodes. A reserved logical locality, `self`, can be used by processes to refer to their execution node. Physical localities have an absolute meaning within the net, while logical localities have a relative meaning depending on the node where they are interpreted and can be thought as aliases for network resources. Logical localities are associated to physical localities through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the physical locality of the node.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can execute the following operations over tuple spaces and nodes:

- $\mathbf{in}(t)@l$: evaluates tuple t and looks for a matching tuple t' in the tuple space located at l. Whenever a 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.

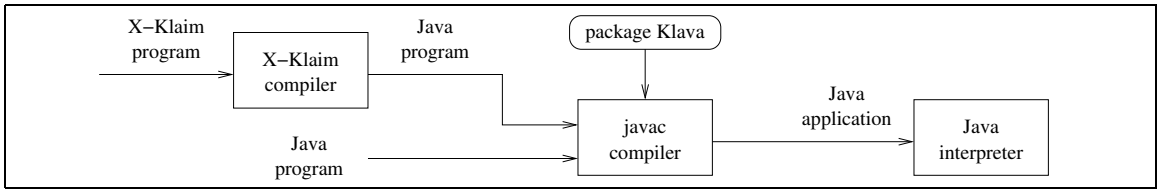


Figure 1: The framework for X-KLAIM.

- **read**(t)@l: differs from **in**(t)@l only because the tuple t' selected by pattern-matching is not removed from the tuple space located at l.
- **out**(t)@l: adds the tuple resulting from the evaluation of t to the tuple space located at l.
- **eval**(P)@l: spawns process P for execution at l.

X-KLAIM has a syntax that is quite similar to *Pascal* syntax; blocks of code are delimited by **begin end** and the character '**;**' is used as a separator for instructions and not as a terminator. This implies that the code

begin instr1 ; instr2 **end**

is syntactically correct, while the following one is not:

begin instr1 ; instr2 ; **end**

X-KLAIM is case-insensitive for keywords, but not for variable and process names. Comments start with the symbol **#** and terminate at the end of the line. An X-KLAIM program is made of some global process definitions and some node definitions.

In the rest of this document we will describe the syntax of X-KLAIM and provide some programming examples. The version of the language presented here differs from previous presentations ([Bettini, 1998; Bettini *et al.*, 1998; Bettini *et al.*, 2000; Bettini *et al.*, 2001b]) in that it relies on the hierarchical model of KLAIM, presented in [Bettini *et al.*, 2002a; Bettini, 2003a]. Thus, it also provides all the primitives for explicitly dealing with node connectivity. X-KLAIM now provides object-oriented features, i.e., object-oriented code mobility, structured via mixin-based inheritance, according to the philosophy of MOMI [Bettini *et al.*, 2002b; Bettini *et al.*, 2003b].

2 Processes

The main computational unit in KLAIM and thus also in X-KLAIM is a process. The syntax of X-KLAIM processes is shown in Table 1. A process is addressable in an X-KLAIM program through its name, and can receive arguments and declare some local variables. Arguments are passed to a process by value, unless **ref** is used for declaring a formal parameter.

Local variables of processes are declared in the **declare** section of the process definition. Standard base types are available (**str**, **int**, **bool**) as well as X-KLAIM typical types, such as **loc**, **logloc** and **phyloc** for locality variables, **process** for process variables and **ts**, i.e., tuple space, for implementing data structures by means of tuple spaces, e.g., lists, that can be accessed through standard tuple space operations.

A formal parameter has simply the form <name>:<type>. A variable (resp. a list of variables with the same type) can be declared as follows:

var <name> : <**type**>
var <name_1>, ..., <name_n> : <**type**>

The same style can be used to declare a variable in the process body. A variable declared in the **declare** section is visible in the whole process body, while a variable declared in the process body is visible only in the code block where it is declared.

Constant variables are declared without specifying the type, since this is automatically inferred from their values:

const s := `"foo"` ; # *a string constant*
const b := **true** ; # *a boolean constant*
const i := 1971 ; # *an integer constant*

Logical locality constants are declared by using the type **locname**; the value of such a constant is represented by the symbol name itself.

A locality variable can be initialized with a string that will correspond to its actual value. Logical localities are basically names, while physical localities must have the form

RecProcDefs	::=	**rec** id formalparams procbody
	|	**rec** id formalparams **extern**
	|	RecProcDefs **;** RecProcDefs
formalParams	::=	**[** paramlist **]**
paramlist	::=	ϵ | id **:** type | **ref** id **:** type | paramlist **,** paramlist
procbody	::=	declpart **begin** proc **end**
declpart	::=	ϵ | **declare** decl
decl	::=	**const** id **:=** expression
	|	**locname** id
	|	**var** idlist **:** type
	|	decl **;** decl
idlist	::=	id | idlist **,** idlist
proc	::=	KAction | **nil**
	|	id **:=** expression | **var** id **:** type | proc **;** proc
	|	**if** boolexp **then** proc **else** proc **endif**
	|	**while** boolexp **do** proc **enddo**
	|	**forall** Retrieve **do** proc **enddo**
	|	procCall | **call** id | **(** proc **)** | **print** exp
KAction	::=	**out(** tuple **)@**id | **eval(** proc **)@**id | Retrieve
	|	**go@**id | **newloc(** id **)**
Retrieve	::=	Block | NonBlock
Block	::=	**in(** tuple **)@**id | **read(** tuple **)@**id
NonBlock	::=	**inp(** tuple **)@**id | **readp(** tuple **)@**id | Block **within** numexp
boolexp	::=	NonBlock | *standard bool exp*
tuple	::=	expression | proc | **!** id | tuple **,** tuple
procCall	::=	id **(** actuallist **)**
actuallist	::=	ϵ | expression | proc | id | actuallist **,** actuallist
expression	::=	$*$ expression | *standard exp*
id	::=	*string*
type	::=	**int** | **str** | **loc** | **logloc** | **phyloc** | **process** | **ts** | **bool**

Table 1: X-KLAIM process syntax. Syntax for other standard expressions is omitted.

<IP_address>:<port>, so a physical locality variable has to be initialized with a string corresponding to an Internet address. The type **loc** represents a general locality, without specifying whether it is logical or physical, while **logloc** (resp. **phyloc**) represents a logical (resp. physical locality). A simple form of subtyping is supplied for locality variables in that

$$\textbf{logloc} <: \textbf{loc} \qquad\qquad \textbf{phyloc} <: \textbf{loc}$$

Here are some examples of locality variable manipulations:

var l : **loc**;
var output : **logloc**;
var server : **phyloc**;
output := `"screen"`;
server := `"150.217.14.10:9999"`;
l := output; # *OK: a logical locality can be assigned to a locality*
l := server; # *OK: a physical locality can be assigned to a locality*

Locality names (logical localities) resolution does not take place automatically (differently from previous versions); instead, it has to be explicitly invoked by putting the operator $*$ in front of the locality that has to be evaluated:

l := $*$output; # *retrieve the physical locality associated to output*
out($*$output)`@self`; # *insert the physical locality associated to output*

However logical localities used as "destination" are still evaluated automatically, i.e., if the locality used after the @ is a logical one, it is first translated to a physical locality.

Apart from standard KLAIM operations, X-KLAIM provides non-blocking version of the retrieval operations, namely **readp** and **inp**; these act like **read** and **in**, but, if no matching tuple

```
if readp(!i, !j)@l and (not in("foo", !k)@self within 3000) then
  out(i, j)@self
else
  out(k)@self
endif
```

Listing 2.1: A more complex retrieval operation.

is found, they do not block the running process and simply return `false`. Thus these operations can be used where a boolean expression is expected. Some versions of Linda also introduce such operations [Carriero & Gelernter, 1989a]. These variants are useful when one wants to search for a matching tuple in a tuple space without running the risk of blocking. For instance, **readp** can be used to test whether a tuple is present in a tuple space.

Furthermore, a timeout (expressed in milliseconds) can be specified for **in** and **read**, through the keyword **within**; the operation is then a boolean expression that can be tested to determine whether the operation succeeded:

```
if in(!x, !y)@l within 2000 then
  # ... success!
else
  # ... timeout occurred
endif
```

Time-outs can be used when retrieving information for avoiding that processes block due to network latency bandwidth or to absence of matching tuples.

These boolean expressions can be combined in order to execute more complex retrieval operations, as in the example in Listing 2.1: the **if** succeeds if a tuple matching $(!i, !j)$ is present at l and no tuple matching $(\texttt{"foo"}, !k)$ is found at `self` within 3 seconds.

The compiler also performs some static analysis in order to check whether an identifier is initialized within a specific scope. The retrieval operations in X-KLAIM are binders for the formal fields of their tuples in the sense that after such an operation succeeded, the identifiers used as formal fields can be considered initialized. Thus, in the example in Listing 2.1, the `out(i, j)@self` is correct, since in the **then** branch both `i` and `j` are initialized; on the contrary `out(k)@self` is rejected, since the test of the `if` statement may have failed only because of the `readp(!i, !j)@l`; thus in the **else** branch `k` may not be initialized. If **or** had been used, instead of **and**, in Listing 2.1, then `out(k)@self` would have been correct in the `else` branch, while `out(i, j)@self` would have been rejected in the `then` branch. The evaluation of boolean expressions in X-KLAIM is *lazy*.

It is often useful to iterate over all elements of a tuple space matching a specific template. However, due to the inherent nondeterministic selection mechanism of pattern matching a subsequent **read** (or **readp**) operation may repeatedly return the same tuple, even if several other tuples match. Thus the following piece of code that aims at copying to a different node all tuples matching $(\textbf{int}, \textbf{str})$ after incrementing the first element is destined to fail

```
while readp(!i, !s)@self do
  out(i + 1, s)@l
enddo
```

since it could end up in an infinite loop, always modifying the same tuple. Repeatedly withdrawing such a tuple with **inp** does not solve the problem, since, in order not to be destructive on the original site, it would force to reinsert the withdrawn tuple, thus incurring in the same problem as above.

For this reason X-KLAIM provides the construct **forall** that can be used for iterating actions through a tuple space by means of a specific template. Its syntax is:

```
forall Retrieve do
  proc
enddo
```

We refer the reader to Table 1 for the syntax of "Retrieve". The informal semantics of this operation is that the loop body "proc" is executed each time a matching tuple is available. Even duplicate tuples are repeatedly retrieved by the **forall** primitive; it is however guaranteed that each tuple is retrieved only once. Thus, instead of the while-based code above we write:

```
forall readp(!i, !s)@self do
  out(i + 1, s)@l
enddo
```

Now, if the tuple space contains three matching tuples (of which two are identical): (10, "foo"), (10, "foo"), (20, "bar"), after the execution of the loop instruction the tuple space at `l` will contain the tuples (11, "foo"), (11, "foo"), (21, "bar").

Notice however that the tuple space is not blocked when the execution of the **forall** is started, thus this operation is not atomic: the set of tuples matching the template can change before the command completes. A locked access to such tuples can be explicitly programmed. Our version of **forall** is different from the one proposed in [Butcher *et al.*, 1994] since parallel processes are not created for each retrieved tuple (this would not be consistent with the "iterating" nature of **forall**; a similar functionality could be easily achieved by using **eval** in the loop body). Our **forall** is similar to the **all** variations of retrieval operations in *PLinda* [Anderson & Shasha, 1992].

The **forall** primitive has a different semantics depending on the nature of the retrieval operation: if a blocking action is used, then the process executing **forall** is blocked until another (never retrieved) tuple becomes available; instead, when a nonblocking action is used, the process exits from the **forall** loop and continues its execution.

The KLAVA system automatically assigns a unique identifier to each tuple; such an identifier can be considered as a GUID (*Global Unique Identifier*); after the matching, the identifier of the matching tuple is stored in the template used in the **forall**. The pattern matching procedure checks the list of *already retrieved tuples* of the template, and guarantees that each matching tuple be not retrieved twice. Since the list of tuples that have already been retrieved belongs to a specific template, a subsequent **forall** operation, within the same process, will retrieve the same tuples of previous **forall** loops, if they have not yet been removed.

Data structures can be implemented by means of the data type **ts**; a variable declared with such type can be considered as a tuple space and can be accessed through standard tuple space operations, apart from **eval** that would not make sense when applied to variables of type **ts**. Furthermore **newloc** has a different semantics when applied to a variable of type **ts**: it empties the tuple space.

forall is then useful for iterating through such data structures; for instance the following piece of code transforms a list, stored in the variable `list` of type **ts**, containing data of the form (**str**, **int**) into a list containing data of the form (**int**, **str**):

```
declare
  var s : str;
  var i : int;
  var list : ts;
...
forall inp(!s, !i)@list do
  out(i, s)@list
enddo
```

Notice that we use the non-blocking version of **in**, otherwise the process would be blocked when it finished iterating through the list.

eval(*P*)@*l* starts the process *P* on the node at locality *l*; *P* can be either a process name (and its arguments):

```
eval( P("foo", 10) )@l
```

or the code (i.e., the actions) of the process to be executed:

```
eval( in(!i)@self; out(i)@l2 )@l
```

Processes can also be used as tuple fields, such as in the following code:

out(P("foo", 10), **in**(!i)@self; **out**(i)@l2)@l

However, in this case, these processes are not started automatically at *l*: they are simply inserted in its tuple space. They can be retrieved (e.g., by another process executing at *l*) and explicitly evaluated:

in(!P1, !P2)@self;
eval(P1)@self;
eval(P2)@self

Thus, basically, **eval** provides *remote evaluation* functionalities, while **out** can be used to implement the *code on-demand* paradigm.

According to the requirements made on the run-time support, code mobility may also be classified as follows [Cugola *et al.*, 1997; Hohlfeld & Yee, 1998]:

- *weak mobility*: code coming from a different site can be dynamically linked;
- *strong mobility*: a thread can move its code and execution state to a different site and resume its execution on arrival;
- *full mobility*: in addition to strong mobility, the whole state of the running program is moved, and this includes all threads' stacks, namespaces (e.g., I/O descriptors, file-system names) and other resources, so that migration is completely transparent.

Full mobility can be considered orthogonal to mobile agents and requires a strong support from the operating system layer. Strong mobility is the notion of mobility that best fits in with the classical concept of mobile agent: the execution state of a migrating agent is suspended, and its stack and program counter are sent to the destination site, together with the relevant data; at the destination site, the stack of the agent is reconstructed and the program counter is set appropriately, i.e., to the first instruction after the migration action. Instead, weak mobility does not meet the intuitive idea of mobile agent, because automatic resumption of execution thread is one of the main features of mobile agents (it exalts their autonomy). X-KLAIM provides *strong mobility* by means of the action **go**@*l* (this is obtained through a preprocessing transformation described in [Bettini & De Nicola, 2001]) that makes an agent migrate to *l* and resume its execution at *l* from the instruction following the migration action. Thus in the following piece of code an agent retrieves a tuple from the local tuple space, then it migrates to the locality `l` and inserts the retrieved tuple into the tuple space at locality `l`:

in(!i, !j)@self;
go@l;
out(i, j)@self

Also I/O operations in X-KLAIM are implemented as tuple space operations. For instance the logical locality *screen* can be attached (mapped) to the output device. Hence, operation **out**("foo\n")@*screen* corresponds to printing the string "foo\n" on the screen. Similarly, the locality *keyboard* can be attached to the input device, so that a process can read what the user typed with a **in**(!s)@*keyboard*. Further I/O devices, such as files, printers, etc., can also be handled through the locality abstraction. An example of this usage is shown in Section 6.

However, in order to make programming in X-KLAIM slightly easier, we also supply the instruction **print** that, given a string, prints it to the standard output, followed by a carriage return. String concatenation can be used to compose complex strings. Symbols and constants that do not have type `str` are automatically converted by the compiler:

print "the value of i is " + i + ". Is it < 10? " + (i < 10)

Type casts are supplied by X-KLAIM but only in a safe way. Indeed they are only a means for solving possible ambiguities. For instance the following instruction

out(**inp**(10)@self)@l

NodeDefs	::=	ϵ | **nodes** nodedefs **endnodes**
ProcDefs	::=	ϵ | RecProcDefs
nodedefs	::=	id **::** { environment } nodeoptions nodeprocdefs
	|	nodedefs **;** nodedefs
environment	::=	ϵ | id $\sim$ id | environment **,** environment
nodeprocdefs	::=	procbody | nodeprocdefs || nodeprocdefs
nodeoptions	::=	**class** id | **port** num

Table 2: X-KLAIM node syntax.

inserts the process `inp(10)@self` in the tuple space corresponding to l: since X-KLAIM is a higher-order language, the action `inp(10)@self` is interpreted as a process made only by such action. If on the contrary the programmer wants to actually insert the boolean result of `inp(10)@self`, he can do that by performing an explicit cast to type **bool**:

out((**bool**) **inp**(10)@`self`)@l

This way the program assumes the following semantics:

var b : **bool**;
b := **inp**(10)@`self`;
out(b)@l

Notice that every cast is checked by the compiler to verify that its validity; for instance the instruction

out((**bool**) **in**(10)@`self`)@l

would be rejected by the compiler, because **in** does not return a boolean value.

Apart from the implicit cast to string used for expressions printed with **print**, the compiler also performs other implicit cast when passing arguments to a process; thus, if P is a process that receives a boolean and a process, the process call

P(**inp**(10)@`self`, **inp**(10)@`self`)

is automatically converted to

P((**bool**) **inp**(10)@`self`, **inp**(10)@`self`)

3 Nodes

A process can execute only on a KLAVA node since in KLAIM nodes are the execution engines. The syntax for defining a node in X-KLAIM is in Table 2. A node is defined by specifying its name (*id*), its allocation environment, some options (described later) and a set of processes running on it. An allocation environment contains the mapping from logical localities to physical localities of the form

logical_locality_variable $\sim$ physical_locality_constant

thus it also implicitly declares the logical locality variables for all the processes defined in the node. Processes defined in a node have the same syntax of Table 1 but they do not have a name, since these processes are visible and accessible only from within the node where they were defined and not in the whole program. Basically the processes defined in a node correspond to the `main` entry point in languages such as Java and C.

With the option **class** it is possible to specify the actual Java class that has to be used for this node, and the option **port** can be used to specify the Internet port where the node is listening. Notice that, together with the IP address of the computer where the node will run, the port number defines the physical locality of the node.

3.1 Hello World in X-KLAIM

Usually the first program ever written in a language is the famous "Hello World". We present its version in X-KLAIM:

```
# HelloWorld.xklaim
nodes
hello_world :: {}
 begin
  print "Hello World!"
 end
endnodes
```

After compiling the file `HelloWorld.xklaim` with the X-KLAIM compiler,

```
xklaim HelloWorld.xklaim
```

and after compiling the resulting generated file `HelloWorld.java` with the Java compiler,

```
javac HelloWorld.java
```

the program can be started with the command

```
java HelloWorld
```

This will start the node `hello_world` listening on the standard port (9999) and the process printing `"Hello World"` is started on this node.

Notice: You must have the Java package KLAVA installed in order to compile the Java code generated by the compiler and then to run the Java programs (see [Bettini, 2003b]).

An alternative way of writing the same program is to define a process for printing the string, and then run that process from within the node:

```
# HelloWorld2.xklaim
rec HelloProc[]
 begin
  print "Hello World!"
 end

nodes
hello_world2 :: {}
 begin
  eval(HelloProc())@self
 end
endnodes
```

By compiling this program you will notice that the compiler generates a Java program with the same name of the original source (e.g., containing the class `HelloWorld2` with the `main` method), and a separate Java source for each process (e.g., `HelloProc.java`). Of course you have to compile all the Java sources generated by the compiler in order to run the program.

This way the code of the process `HelloProc` can be reused by other nodes in the same program. Indeed, it can also be used by other programs: they can import its implementation as follows

```
rec HelloProc[] extern

nodes
hello_world3::{}
 begin
  eval(HelloProc())@self
 end
endnodes
```

Of course, the `HelloProc.java` must have already been generated by the `xklaim` compiler. Notice, however, that in this case the `xklaim` compiler will not actually check that a process `HelloProc` is actually defined in some other source. If this is not the case, when you compile the corresponding generated Java sources, you may get an error by the Java compiler if such a process cannot be found anywhere. Thus, the **extern** keyword is exactly the same as in the language C.

A distributed version of the "Hello World" program can be easily built in X-KLAIM. We can write the sender and the receiver into two separate files:

```
# HelloSender.xklaim, compile it with option −T 1
rec HelloSenderProc[ dest : loc ]
 begin
  out("Hello World!")@dest
 end

nodes
hello_sender::{receiver ~ localhost:11000}
 port 10000
 begin
  eval(HelloSenderProc(receiver))@self
 end
endnodes
```

The sender node maps the logical locality *receiver* to the physical locality `localhost:11000`, and passes the logical locality to the sender process that simply puts a tuple containing the string `"Hello World"` in the tuple space of the remote node[1]. Notice that this node also specifies its physical locality, by declaring its port. Finally, in order to keep the example simple, we rely on the flat network model of KLAIM, where all nodes belong to the same net, and they are all at the same level. For this reason, we have to pass the option `-T 1` to the `xklaim` compiler when we compile this source:

```
xklaim HelloSender.xklaim -T 1
```

The receiver node, that is executing on `localhost` listening on port 11000, simply waits for a tuple made of a string a print the received message (again use the option `-T 1`):

```
# HelloReceiver.xklaim, compile it with option −T 1
nodes
hello_receiver::{}
 port 11000
 declare
  var msg : str
 begin
  in(!msg)@self;
  print "received: " + msg
 end
endnodes
```

Now, after compiling all the programs (and also the generated Java sources), you cannot simply run the two programs, since the two nodes expect to connect to a net server. Thus, you must first run the KLAVA net server (the port number is optional, by default it is 9999):

```
java Klava.Net 9999
```

and then run the receiver node, by specifying the address and the port number of the net server:

```
java HelloReceiver localhost 9999
```

[1] In this simple example, we assume that all nodes run on the same machine — for this reason we use the `localhost` address. Of course, you can experiment by running the two nodes on different machines and in that case you have to substitute `localhost` with the correct IP of the receiver node

and when the receiver node is connected to the net server, you can start the sender:

```
java HelloSender localhost 9999
```

Since the two nodes are connected to the same net server, they will be able to communicate.

Another possibility is to send the `HelloSenderProc` process directly to the receiver site so that it can out the tuple locally:

```
# HelloSender2.xklaim, compile it with option −T 1
rec HelloSenderProc[ dest : loc ] extern

nodes
hello_sender2::{receiver ~ localhost:11000}
 port 10000
 begin
  eval(HelloSenderProc(self))@receiver
 end
endnodes
```

Notice that the process that outs the tuple is just the same (since it is parameterized over the destination locality), and the sender passes `self` as the destination locality to the process and spawns the process for execution at the receiver site. Further mobility examples are shown in the next section.

3.2 Program output

During execution of X-KLAIM programs, many things are written to the standard output (console). These can be useful to see whether the programs behave correctly and also to see whether actions complete successfully. Here we describe only the most relevant information that are written to the standard output.

First of all, when you start a node that connects to a net server, you will see something similar to the following strings, assuming the computer we run the programs has IP `192.168.1.100` (in particular, this is the output of `HelloReceiver` program):

```
<192.168.1.100:11000>Connecting to 192.168.1.100:9999 ...
<192.168.1.100:11000>Socket obtained...
<192.168.1.100:11000>Connected.
<192.168.1.100:11000>Login as 192.168.1.100:11000 ...
<192.168.1.100:11000>Physical Locality: 192.168.1.100:11000
<192.168.1.100:11000>Login successful!
(Klava.LoginNodeCoordinator) Login succedeed to 192.168.1.100:9999
```

This shows that the node of `HelloReceiver` successfully enters the net server (`192.168.1.100:9999`) with physical locality `192.168.1.100:11000`. The connection output of `HelloSender` will be similar (of course with a different physical locality). In particular, when a string is prefixed by a string in parenthesis (like the last line above), it means that this is the output of a specific process (in this case, the process of the KLAVA package that takes care of actually performing the login in a net).

The following lines are written by the process in the node of `HelloReceiver` that performs the **in** operation:

```
(__hello_receiver_1) IN( ( !KString )  )@self
(__hello_receiver_1) > IN( ( !KString )  )@192.168.1.100:11000
```

notice that the second string (prefixed by a >) shows that the logical locality `self` is first translated into the corresponding physical locality (in this case the physical locality of the node itself). Then, the process is blocked waiting to find a matching tuple.

On the sender site we will see these strings:

```
(HelloSenderProc) OUT( ( Hello World! )  )@receiver
(HelloSenderProc) >OUT( ( Hello World! )  )@192.168.1.100:11000
```

Again, the logical locality is firstly translated into the corresponding physical locality and then the tuple is sent to the remote site.

At this point, on the receiver site, we will see these strings:

```
(Node - hello_receiver) OUT( ( Hello World! )  )@192.168.1.100:11000 From 192.168.1.100:10000
(__hello_receiver_1) -> IN( ( Hello World! )  )@192.168.1.100:11000
(__hello_receiver_1) received: Hello World!
```

The first string, output by the node itself, shows that a tuple has been received by a remote site. At this point, the process that was waiting for a matching tuple finds one (the `->` says that the blocking operation has succeeded), and the string received is printed.

Considering the `HelloSender2` program, that directly spawns the `HelloSenderProc` to the receiver site, the output of the `HelloReceiver` node will be as follows:

```
(__hello_receiver_1) IN( ( !KString )  )@self
(__hello_receiver_1) > IN( ( !KString )  )@192.168.1.100:11000
(Node - hello_receiver) EVAL( ( KlavaProcessPacket : HelloSenderProc )  )@192.168.1.100:11000
   From 192.168.1.100:10000
Starting a Remote Process ...
(HelloSenderProc) OUT( ( Hello World! )  )@self
(HelloSenderProc) >OUT( ( Hello World! )  )@192.168.1.100:11000
(__hello_receiver_1) -> IN( ( Hello World! )  )@192.168.1.100:11000
(__hello_receiver_1) received: Hello World!
```

Finally, let us observe that the sequence of these strings may interleave, since they are output by different concurrent threads.

4 Mobility Examples

In this section we show a few programming examples dealing with mobility, implemented in X-KLAIM. The first one is a *news gatherer*, that relies on mobile agents for retrieving information on remote sites. We assume that some data are distributed over the nodes of an X-KLAIM net and that each node either contains the information we are searching for, or, possibly, the locality of the next node to visit in the net.

The agent *NewsGatherer* first tries to read a tuple containing the information we are looking for, if such a tuple is found, the agent returns the result back home; if no matching tuple is found within 10 seconds, the agent tests whether a link to the next node to visit is present at the current node; if such a link is found the agent migrates there and continues the search, otherwise it reports the failure back home.

The first implementation of such an agent is shown in Listing 4.1 and employs **eval** for spawning an instance of the agent to a remote site. Since **eval** implements weak mobility, it is necessary to explicitly spawn a new copy to the new site, passing all the parameters representing the execution state of the agent: the boolean `finish` says whether the agent has visited all the possible sites, and the search is considered successful if `itemVal` is not empty.

Notice that the source of the agent is a little bit complex, since it might not be clear, at first glance, what the agent is supposed to do. One can use strong mobility in order to make the source clearer. The implementation of the agent exploiting strong mobility (by means of the migration operation **go**) is reported in Listing 4.2.

The third example is still an autonomous information retrieval agent in the context of a virtual *market place*: suppose that someone wants to buy a specific product at a market made of geographically distributed shops. To decide at which shop to buy, she/he activates a migrating agent which is programmed to find and return the name of the closest shop (i.e. the shop within the chosen area, determined by a maximal distance parameter) with the lowest price. The implementation of the agent `MarketPlaceAgent` is shown in Listing 4.3.

The `MarketPlaceAgent` takes as parameters the product name, the maximal distance and the locality where the result of the search must be returned. The agent is sent (by means of an **eval** not shown here) for execution at the node containing the marketplace directory, where it asks for

```
rec NewsGatherer[ item : str, itemVal : str, finish : bool, retLoc : loc ]
 declare
  var itemVal : str ;
  var nextLoc : loc
 begin
  if not finish then
   if read( item, !itemVal )@self within 10000 then
    eval( NewsGatherer( item, itemVal, true, retLoc ) )@retLoc
   else
    if readp( item, !nextLoc )@self then
     eval( NewsGatherer( item, "", false, retLoc ) )@nextLoc
    else
     eval( NewsGatherer( item, "", true, retLoc ) )@retLoc
    endif
   endif
  else
   if itemVal != "" then
    print "found " + itemVal
   else
    print "search failed"
   endif
  endif
 end
```

Listing 4.1: X-KLAIM implementation of a news gatherer using **eval**.

```
rec NewsGatherer[ item : str, retLoc : loc ]
 declare
  var itemVal : str ;
  var nextLoc : loc ;
  var again : bool
 begin
  again := true;
  while again do
   if read( item, !itemVal )@self within 10000 then
    go@retLoc;
    print "found " + itemVal;
    again := false;
   else
    if readp( item, !nextLoc )@self then
     go@nextLoc
    else
     go@retLoc;
     print "search failed";
     again := false
    endif
   endif
  enddo
 end
```

Listing 4.2: X-KLAIM implementation of a news gatherer using strong mobility.

```
rec MarketPlaceAgent[ ProductMake : str, retLoc : loc, distance : int ]
 declare
  var shopList : TS ;
  var nextShop, CurrentShop, thisShop : loc ;
  var CurrentPrice, newCost : int ;
  locname screen
 begin
  # ask for a list of shops which are within a certain distance
  out( "cshop", distance )@self;
  in( "cshop", !shopList )@self;
  out( "retrieved list: ", shopList )@screen;
  CurrentPrice := 0 ;
  CurrentShop := self ;
  # while there are shops in the list to be visited
  forall inp( ! nextShop )@shopList do
   thisShop := nextShop ;
   go@nextShop ; # migrate to the next shop ;
   out( "AgentClient: searching for ", ProductMake )@screen ;
   if read( ProductMake, ! newCost )@self within 10000 then
    if ( CurrentPrice = 0 OR newCost < CurrentPrice ) then
     # update the best price
     CurrentPrice := newCost;
     CurrentShop := thisShop
    endif
   endif
  enddo ;
  # OK we're done, let's send the results
  out( ProductMake, CurrentShop, CurrentPrice )@retLoc
 end
```

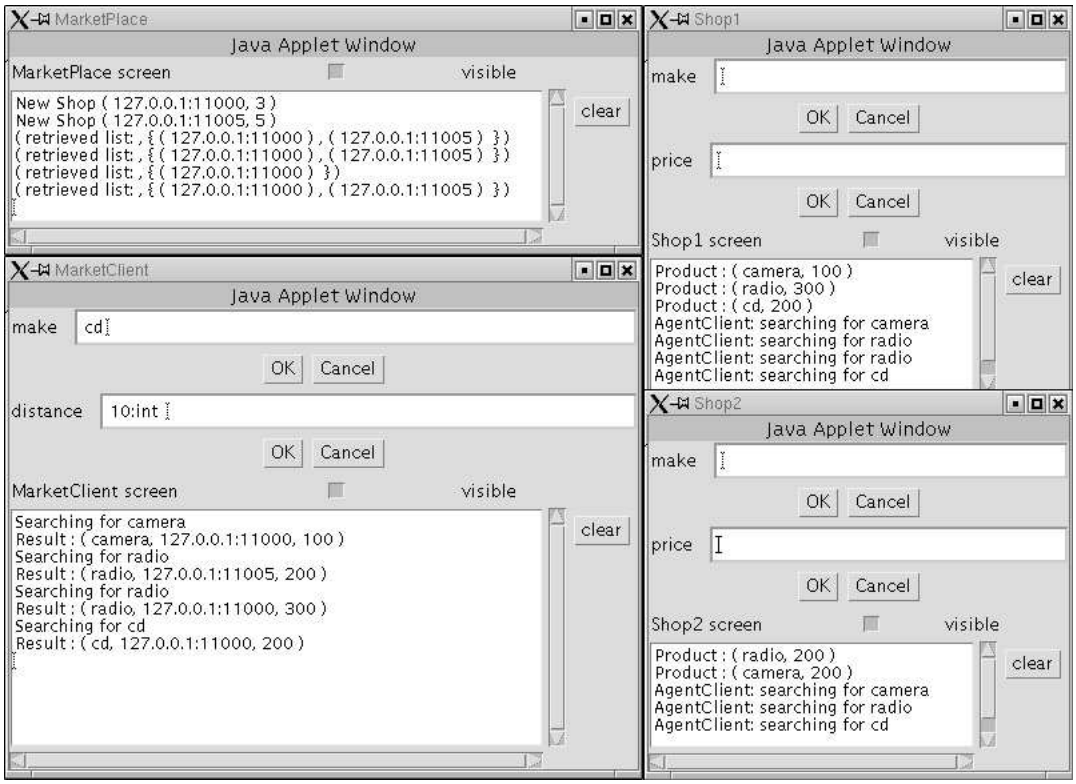
Listing 4.3: X-KLAIM implementation of an agent visiting shops of a virtual market place searching for an item with the lowest price.

the list of the shops in the selected shopping area. Then, `MarketPlaceAgent` migrates to the first shop in the list. At each shop, `MarketPlaceAgent` checks the price of the wanted product, possibly updating the information about the lowest price and the shop that offers it, and migrates to the next shop in the list. If there are no more shops to visit, `MarketPlaceAgent` sends the result of the search back to the locality received as parameter. The list of nodes to visit is stored in a list (implemented through a **ts**) and **forall** is used for iterating over this list.

Screenshot 4.1 shows a client that performs some searches through the `MarketPlaceAgent` in two shops. In this example there are two shops affiliated to the market place: `Shop1` at physical locality `127.0.0.1:11000` with a distance of 3, and `Shop2` at physical locality `127.0.0.1:11005` with a distance of 5; this information is shown in the window of the market place directory (up left). The client sends the agent searching for a `camera` within a distance of 10, so the market place directory provides the agent with a list made of the localities of the two shops, and after visiting both, the agent reports home that the first shops sells the searched item at the lower cost. The second query has basically the same parameters but the agent has to search for a `radio` and this time the second shop sells it at the lower price. Then it still searches for a radio but within a closer distance (e.g., 4) and this time the second shop is not even visited (since its distance is 5, so the market place directory does not put it into the list communicated to the agent). Finally a `cd` is searched for (within a wider distance) and when visiting the second shop a timeout is raised, since that shop does not sell that item.

We conclude this section by presenting an example that uses the remote evaluation paradigm, thus, the code does not to autonomously migrate: it is moved by another process. This example implements a *load balancing system* that dynamically redistributes mobile code among several processors: we suppose that remote clients send processes for execution to a server node that dis-

Screenshot 4.1 The market place directory (up left), the market client (down left) and two shops of the virtual market place.



tributes the received processes among a group of processors by using, each time, the (estimated) idlest one. Each processor sends a number of "credits" to the server (this number corresponds to the processor availability to perform computations on behalf of the server); the server stores the number of credits in a database and, when needed, it chooses the processor with the highest number of credits and decreases this number.

When a processor receives a process, it immediately starts executing the process (in a parallel thread) and sends a credit back to the server. Indeed, the system is based on the heuristic that if a processor is busy, it cannot send a credit back, or at least it does not send a credit immediately.

This example is implemented by the code fragment in Listing 4.4 that shows the server that dispatches the received process to the idlest processor (left) and the processor that receives a process for execution from the server and sends a credit back to it. The code presented here is simplified in order to concentrate on the code mobility related parts (e.g., it does not handle cases such as all credits are exhausted for all processors). Notice that processes are exchanged by means of **out** and **in**. Since in this new version of KLAIM processes are not automatically "closed"[2] when sent with an **out**, then when a process is executed in a processor it will actually use the local resources.

The overall architecture of this load balancing system is based on a *push* model, in that the server delivers the processes to be executed to a chosen processor node. An alternative implementation could be based on a *pull* model: a processor node, when idle, asks the server for a process to be executed. This architecture can be employed to develop systems similar to *SETI@home* [Korpela *et al.*, 2001] that uses Internet-connected computers in the *Search for Extraterrestrial Intelligence* (*SETI*): users that want to help the project can install this software that downloads data to be analyzed from the server when the computer is idle (for instance when the screen saver starts).

[2]In the early version of X-KLAIM, when a process was sent to a remote site with an **out**, its *closure* was actually sent, i.e., the process together with the allocation environment of the node where **out** is performed. This implied that all the logical localities of the sent process were actually translated using the allocation environment of the starting node, instead of the destination node. This corresponded to a *static scoping* discipline for **out**.

```
rec DeliverProcess[ ProcessorDB : ts ]
 declare
  var P : process ;
  var HighestCredit, Credits : int ;
  var Processor, HighestProcessor : loc
 begin
  while ( true ) do
   in( !P )@self ; # wait for a process
   HighestCredit := 0 ;
   forall readp( !Processor, !Credits )@ProcessorDB do
    if ( Credits > HighestCredit ) then
     HighestCredit := Credits ;
     HighestProcessor := Processor
    endif
   enddo ;
   out( P )@HighestProcessor ;
   # update its credits
   in( HighestProcessor, HighestCredit )@ProcessorDB ;
   out( HighestProcessor, HighestCredit − 1 )@ProcessorDB
  enddo
 end
```

```
rec ReceiveProcess[ server : loc ]
 declare
  var P : process ;
  locname screen
 begin
  while ( true ) do
   in( !P )@self ;
   eval( P )@self ;
   out( "SERVER", "CREDIT", self )@server
  enddo
 end
```

Listing 4.4: Load balancing: (left) the server receives a process and dispatches it to the idlest processor; (right) the processor node receives a process and executes it locally and sends a credit back to the server.

NodeCoordinator	::=	**rec** NodeCoordDef
NodeCoordDef	::=	**nodecoord** id formalparams declpart nodecoordbody
	|	**nodecoord** id formalparams **extern**
nodecoordbody	::=	**begin** nodecoordactions **end**
nodecoordaction	::=	*standard process action* | **login(** id **)** | **logout(** id **)**
	|	**accept(** id **)** | **disconnected(** id **)** | **disconnected(** id **,** id **)**
	|	**subscribe(** id **,** id **)** | **unsubscribe(** id **,** id **)** | **register(** id **,** id **)** | **unregister(** id **)**
	|	**newloc(** id **)** | **newloc(** id **,** nodecoordactions **)**
	|	**newloc(** id **,** nodecoordactions **,** num **,** classname **)**
	|	**bind(** id **,** id **)** | **unbind(** id **)**
	|	**dirconnect(** id **)** | **acceptconn(** id **)**

Table 3: X-KLAIM node coordinator syntax. This syntax relies on standard process syntax shown in Table 1

5 Node Connectivity in X-KLAIM

As hinted in the introduction, this new version of the language X-KLAIM differs from previous presentations in that it relies on the hierarchical model of KLAIM(we refer the interested reader to [Bettini *et al.*, 2002a; Bettini, 2003a] for a formal treatment of such model). Thus, it also provides all the primitives for explicitly dealing with node connectivity. Consistently with the hierarchical model of KLAIM such actions can be performed only by *node coordinators*.

The syntax of node coordinators is shown in Table 3, and is basically the same of standard X-KLAIM processes (Table 1) apart from the new privileged actions. We briefly comment these new actions:

- **login**(*loc*), where *loc* is an expression of type **loc**, logs the node where the node coordinator is executing at the node at locality *loc*; **logout**(*loc*) logs the node out from the net managed by the node at locality *loc*. **login** can be used as a boolean expression in that it returns **true** if the login succeeds and **false** otherwise.
- **accept**(*l*) is the complementary action of **login** and indeed, the two actions have to synchronize in order to succeed; thus a node coordinator on the server node (the one at which other nodes want to log) has to execute **accept**. This action initializes the variable *l* to the physical locality of the node that is logging. **disconnected**(*l*) notifies that a node has disconnected from the current node; the physical locality of such node is stored in the variable *l*. **disconnected** also catches connection failures. Notice that both **accept** and **disconnected** are

```
rec nodecoord SimpleLogin[ server : loc ]
 begin
  print "try to login to " + server + "...";
  if login( server ) then
   print "login successful";
   out("logged", true)@self
  else
   print "login failed!"
  endif
 end

rec nodecoord SimpleLogout[ server : loc ]
 begin
  in("logged", true)@self;
  print "logging off from " + server + "...";
  logout(server);
  print "logged off."
 end
```

```
rec nodecoord SimpleAccept[]
 declare
  var client : phyloc
 begin
  print "waiting for clients...";
  accept(client);
  print "client " + client + " logged in"
 end

rec nodecoord SimpleDisconnected[]
 declare
  var client : phyloc
 begin
  print "waiting for disconnections...";
  disconnected(client);
  print "client " + client + " disconnected."
 end
```

Listing 5.1: An example showing **login** and **logout** (left) and the corresponding **accept** and **disconnected**.

blocking in that they block the running process until the event takes place. Instead, **logout** does not have to synchronize with **disconnected**.

An example of these four operations is shown in Listing 5.1, where the node coordinators executing on the client are presented on the left, and the complementary ones executing on the server are presented on the right. Notice that the process that executes the **login** communicates with the one that has to execute the **logout** by using a tuple. **accept** and **disconnected** are initializers for the corresponding variables.

- **subscribe**(*loc*, *logloc*) is similar to **login**, but it also permits specifying the logical locality (*logloc* is an expression of type **logloc**) with which a node wants to become part of the net coordinated by the node at locality *loc*; this request can fail also because another node has already subscribed with the same logical locality at the same server. **unsubscribe**(*loc*, *logloc*) performs the opposite operation.

- **register**(*pl*, *ll*), where *pl* is a physical locality variable and *ll* is a logical locality variable, is the complementary action of **subscribe** that has to be performed on the server; if the subscription succeeds *pl* and *ll* will respectively contain the physical and the logical locality of the subscribed node. The association $pl \sim ll$ is automatically added to the allocation environment of the server. **unregister**(*pl*, *ll*) records the unsubscriptions. Notice that an alternative version of **disconnected**, namely **disconnected**(*pl*, *ll*) is supplied, in order to detect lost connections with nodes, that also specifies the logical locality with which a node was subscribed. As the other **disconnected** explained above, this action is more powerful in that it is able to catch also connections brutally closed without an **unsubscribe**. Let us observe that **disconnected** catches also the events of **unregister** so if program uses both, it is up to the programmer to coordinate the two notification actions (an example of such a scenario is shown in Section 6).

An example using these actions is presented in Listing 5.2; the processes are basically similar to those presented in Listing 5.1, but they also deal with logical localities.

bind(*logloc*, *phyloc*) allows to dynamically modify the allocation environment of the current node: it adds the mapping $logloc \sim phyloc$. On the contrary, **unbind**(*logloc*) removes the mapping associated to the logical locality *logloc*. These two operations privileged and only node coordinators can execute them.

In this version of X-KLAIM **newloc** has become a privileged action and is supplied in three forms in order to make programming easier: apart from the standard form that only takes a locality variable, where the physical locality of the new created node is stored, also the form **newloc**(*l*, *nodecoordinator*) is provided. Since **newloc** does not automatically logs the new created

```
rec nodecoord
 SimpleSubscribe[ server : phyloc, name : logloc ]
 begin
  print "try to subscribe at " + server +
   " as " + name + "...";
  if subscribe( server, name ) then
   print "subscribe successfull";
   out("subscribed", true)@self
  else
   print "subscribe failed!"
  endif
 end

rec nodecoord
 SimpleUnsubscribe[ server : phyloc, name : logloc ]
 begin
  in("subscribed", true)@self;
  print "now unsubscribing from " + server +
   " as " + name + "...";
  unsubscribe(server, name);
  print "unsubscribed."
 end
```

```
rec nodecoord SimpleRegister[]
 declare
  var clientloc : logloc;
  var client : phyloc
 begin
  print "waiting for clients to subscribe...";
  if register(client, clientloc) then
   print "client " + clientloc + "~" +
    client + " subscribed"
  else
   print "client failed to subscribe"
  endif
 end

rec nodecoord SimpleUnregister[]
 declare
  var client : logloc
 begin
  print "waiting for unsubscription...";
  unregister(client);
  print "client " + client + " unsubscribed."
 end
```

Listing 5.2: An example showing **subscribe** and **unsubscribe** (left) and the corresponding **register** and **unregister**.

node in the net of the creating node, this second form allows to install a node coordinator in the new node that can perform this action (or other privileged actions).

Notice that this is the only way of installing a node coordinator on another node: due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. In order to provide better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`.

Finally the third form of **newloc** takes two additional arguments: the port number where the new node is going to be listening (and this also determines its physical locality, since the IP address will be the same of the creator node), and the (Java) class of the new node. Since I/O devices can be abstracted into nodes, this form of **newloc** enables to construct, for instance, the graphical interface of a node, made up of several I/O sub-nodes. For an example, see Section 6, where *screen*, *keyboard* and *usersList* are logical localities actually mapped into KLAVA nodes that supply and interface for a text area, and input text box and a list.

In this scenario communications among nodes belonging to the same subnet take place, through the gateway node. In case of firewalls or network restrictions the access to a remote node may be permitted only through a server. For instance, an applet can only open a network connection towards the computer it has been downloaded from. If on this computer there is a `NetNode` running that is willing to act as a gateway, the applet is still able to indirectly communicate with all the nodes and, possibly, with applets that are part of that net managed by that gateway; An example of a KLAVA applet is available at `http://music.dsi.unifi.it/klava_applet`. In this sense, a `NetNode` gateway allows nodes to communicate even if they belong to different restricted domains. However, when there are no network restrictions, direct connections can still be established in order to use a direct (probably faster) communication between nodes of the same, or different, subnet.

In this version of X-KLAIM also *direct connections* can be dealt with explicitly, so we provide the complementary privileged action **dirconnect**(*loc*) and **acceptconn**(*l*) that allow to create a unidirectional direct communication channel. Thus if a node n_1 establishes a direct connection with the node n_2 every time n_1 sends a message to n_2 it will do this directly, i.e., without passing through a possible common server. This situation is not symmetric since the direct connection is unidirectional. Should one want a bidirectional *peer to peer* communication, this has to be programmed explicitly so that upon accepting a direct connection from a node, also the other way direction is established.

An example is presented in Listing 5.3; here also the node definitions are shown in order to

```
rec nodecoord SimpleDirConn[ peer : loc ]
 declare
  var test : str
 begin
  print "establishing direct connection to " +
   peer;
  if dirconnect(peer) then
   print "established";
   out("TEST")@peer;
   in(!test)@peer;
   print "sent and receive " + test
  else
   print "direct connection to " +
    peer + " failed."
  endif
 end

rec nodecoord SimpleAcceptConn[]
 declare
  var peer : phyloc
 begin
  print "waiting for direct connections...";
  acceptconn(peer);
  print "accepted direct connection from " + peer
 end

nodes
mandirconnpeer2 :: {}
 port 11000
 class "NetNode"
 start
 declare
  var peer : phyloc
 begin
  eval(SimpleAcceptConn())@self;
  peer := "127.0.0.1:9999";
  eval(SimpleDirConn(peer))@self
 end
endnodes
```

```
rec nodecoord SimpleAcceptConnAndConnect[]
 declare
  var peer : phyloc
 begin
  print "waiting for direct connections...";
  acceptconn(peer);
  print "accepted direct connection from "
   + peer;
  print "now connecting to " + peer;
  if dirconnect(peer) then
   print "established"
  else
   print "direct connection to " +
    peer + " failed."
  endif
 end

nodes
mandirconnpeer1 :: {}
 class "NetNode"
 start
 begin
  eval(SimpleAcceptConnAndConnect())@self
 end
endnodes
```

Listing 5.3: An example showing **dirconnect** and **acceptconn** for establishing a *peer to peer* direct communication.

clarify the scenario: `mandirconnpeer2` wants to engage a peer to peer communication with the node at locality `127.0.0.1:9999`, thus, it executes a node coordinator for establishing the direct connection, and also executes a node coordinator for accepting the corresponding direct connection request (from the other peer). The other peer `mandirconnpeer1` executes the complementary protocol by running a node coordinator that first accepts a direct connection and then establishes a direct connection to the same node.

The node that first tries to establish the direct connection (`mandirconnpeer2` in this example) should execute the **dirconnect** and **acceptconn** in two parallel processes: if it executed the two actions in sequence, the **acceptconn** would not be guaranteed to start before the other peer started its request. This would probably lead to a deadlock. The other peer (`mandirconnpeer1` in this example), instead, can safely execute the complementary **acceptconn** and **dirconnect** in sequence.

6 A Chat System with Connectivity Actions

In this section we present the implementation in X-KLAIM of a chat system; this is based on the one presented in [Bettini *et al.*, 2004a] that, however, did not rely on the new features. The chat system we present in this section is simplified, but it implements the basic features that are present in several chat systems. The system consists of a `ChatServer` and many `ChatClients`.

The system is dynamic because new clients can enter the chat and existing clients may dis-

```
rec nodecoord HandleLogin[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname users, screen, server
 begin
  while ( true ) do
   if register( client, nickname ) then
    out( nickname, client )@usersDB ;
    out( true )@client ;
    SendUserList( client, usersDB ) ;
    out( (str)nickname )@users ;
    out( "Entered Chat : " )@screen ;
    out( nickname, client )@screen ;
    BroadCast( "USER", "ENTER",
          nickname, server, usersDB )
   endif
  enddo
 end

rec SendUserList[ newEnter : phyloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var userLoc : phyloc ;
  var userList : ts
 begin
  newloc( userList ) ;
  forall readp( !nickname, !userLoc )@usersDB do
   if ( userLoc != newEnter ) then
    out( nickname )@userList
   endif
  enddo ;
  out( userList )@newEnter
 end

rec nodecoord HandleDisconnected[ usersDB : ts ]
 declare
  var nickname : logloc ;
  var client : phyloc ;
  locname screen
 begin
  while ( true ) do
   disconnected(client, nickname);
   out("disconnected: ", nickname, client)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec nodecoord HandleUnregister[ usersDB : ts ]
 declare
  var nickname : logloc ;
  locname screen
 begin
  while ( true ) do
   unregister(nickname);
   out("unsubscription: ", nickname)@screen;
   RemoveClient(nickname, usersDB)
  enddo
 end

rec RemoveClient[ nickname : logloc, usersDB : ts ]
 declare
  var client : phyloc ;
  locname screen, users, server
 begin
  if inp( nickname, !client )@usersDB and
      inp( (str)nickname )@users then
   out( "Left Chat : " )@screen ;
   out( nickname, client )@screen ;
   BroadCast( "USER", "LEAVE",
         nickname, server, usersDB )
  endif
 end
```

Listing 6.1: Node coordinators of the chat server dealing with clients' subscriptions.

connect. The server represents the gateway through which the clients can communicate, and the clients logs in the chat server by specifying their "nickname", represented here by a logical locality. A client that wants to enter the chat must subscribe at the chat server. The server must keep track of all the registered clients and, when a client sends a message, the server has to deliver the message to every connected client. If the message is a private one, it will be delivered only to the clients in the list specified along with the message.

6.1 The Chat Server

When a new client issues a subscription request, the server accepts it only if there is no other client with the same nickname, and in case the access is granted, every client is notified about the new client; moreover the new client is also provided with the list of the clients currently in the chat (Listing 6.1). The server keeps a database of all connected clients in a variable `usersDB` of type **ts** where there is a tuple of the shape `(nickname, locality)` for each client, where `nickname` is a logical locality and `locality` is a physical one. Notice that all the processes running on the chat server share this database.

The server uses two (node coordinator) processes for intercepting clients' disconnections: `HandleUnregister` and `HandleDisconnected`. The second one would be useless if the network communications are reliable (i.e., no communication suddenly crashes without further notice); however, this assumption may be too strong in a realistic scenario. Thus `HandleDisconnected` intercepts also this kind of disconnections. As we said above the **disconnected** action returns even after an ordinary unsubscription, so the process `RemoveClient` has to further check whether

```
rec HandleMessage[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc
 begin
  while ( true ) do
   in( "MESSAGE", !message, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCast( "MESSAGE", "ALL",
          message, sender, usersDB )
   endif # ignore errors
  enddo
 end

rec HandlePersonal[ usersDB : ts ]
 declare
  var message : str ;
  var sender : logloc ;
  var from : phyloc ;
  var to : ts
 begin
  while ( true ) do
   in( "PERSONAL", !message, !to, !from )@self ;
   if readp( !sender, from )@usersDB then
    BroadCastTo( "MESSAGE", "PERSONAL",
          message, to, sender, usersDB )
   endif
  enddo
 end
```

```
rec BroadCast[ communication_type : str, message_type : str,
        message : str, from : logloc, usersDB : ts ]
 declare
  var nickname : logloc ;
  var user : phyloc
 begin
  forall readp( !nickname, !user )@usersDB do
   out( communication_type, message_type,
      message, from )@user
  enddo
 end

rec BroadCastTo[ communication_type : str, message_type : str,
        message : str, to : ts, from : logloc, usersDB : ts ]
 declare
  var nickname : str ;
  var user : phyloc
 begin
  forall inp( !nickname )@to do
   # recipients are specified as strings in the "to" list
   # so we have to convert them first
   if readp( (logloc) nickname, !user )@usersDB then
    out( communication_type, message_type,
      message, from )@user
   endif
  enddo
 end
```

Listing 6.2: Processes on the server dealing with message dispatching.

a client has already been removed from the database.

The broadcasting of messages to clients is managed by two processes running on the `ChatServer` node: `BroadCast` and `BroadCastTo` (Listing 6.2): the former sends a message to all connected clients while the latter sends a message only to the clients specified in the list `to`. This second version is useful when delivering *personal* messages.

All messages have the following tuple shape:

```
(communication_type, message_type, message, from)
```

where `communication_type` and `message_type` specify the type of message (e.g., the values `"USER"` together with `"ENTER"` indicate that a user entered the chat, while `"MESSAGE"` and `"ALL"` indicate a chat message that is destined to every client). `message` is the content of the message (e.g., the nickname of the user that entered the chat or the body of a chat message) and `from` is the nickname (logical locality) of the client that originated the message.

Messages are received by the chat server by means of two processes `HandleMessage` and `HandlePersonal` (respectively for standard chat messages and for personal messages) also shown in Listing 6.2. When a client wants to send a personal message it has to specify also a list (a **ts** tuple field) containing the nicknames of the clients it is destined to). These processes are responsible for delivering a message to all the recipient clients.

6.2 The Chat Client

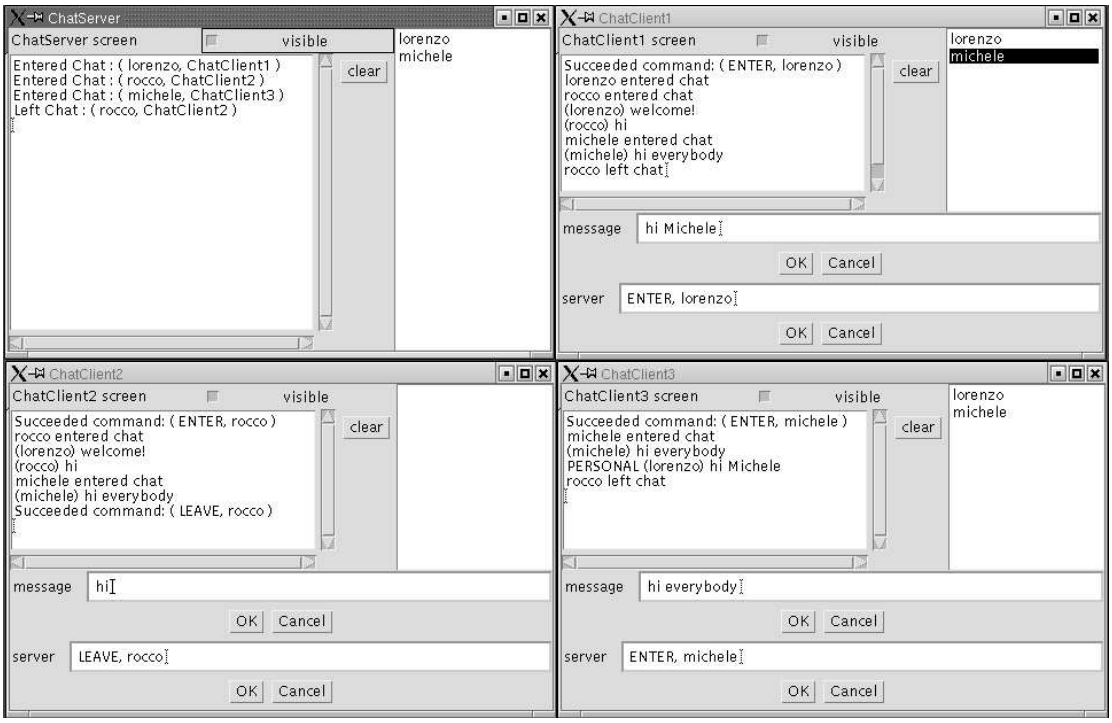
A chat client executes two processes for handling messages dispatched by the server (Listing 6.3): `HandleMessages` takes care of processing chat messages and `HandleServerMessages` handles server messages informing of new clients joining the chat or existing clients leaving (the list of connected clients is updated accordingly). This information is printed on the screen of the client (attached to the locality `screen`).

The user can insert messages for the server (i.e., commands for entering and exiting from the chat) and standard chat messages in two text fields that are attached, respectively, to the

```
rec HandleMessages[]
 declare
  locname screen ;
  const standard_message := "MESSAGE";
  var message, message_type : str ;
  var from : logloc
 begin
  while ( true ) do
   in( standard_message, !message_type,
      !message, !from )@self ;
   if message_type = "PERSONAL" then
    out( "PERSONAL " )@screen
   endif;
   out( "(" )@screen ;
   out( (str)from )@screen ;
   out( ") " )@screen ;
   out( message )@screen ; out( "\n" )@screen
  enddo
 end

rec HandleServerMessages[]
 declare
  locname screen, usersList ;
  const user_message := "USER" ;
  var command, nickname : str;
  var from : logloc
 begin
  while ( true ) do
   in( user_message, !command,
      !nickname, !from )@self ;
   if command = "ENTER" then
    out( nickname )@screen ;
    out( " entered chat\n" )@screen ;
    if not readp(nickname)@usersList then
     out( nickname )@usersList
    endif
   else
    if command = "LEAVE" then
     out( nickname )@screen ;
     out( " left chat\n" )@screen ;
     inp( nickname )@usersList
     # ignore non existing names
    endif
   endif
  enddo
 end

rec nodecoord HandleServerKeyboard[]
 declare
  locname server, screen, serverKeyb, usersList;
  var command, nick : str ;
  var nickname : logloc ;
  var response : bool ;
  var chat_server : phyloc ;
  var userList : ts
 begin
  chat_server := *server;
  while ( true ) do
   in( !command, !nick )@serverKeyb ;
   if ( command != "ENTER" and command != "LEAVE" ) then
    out( "Unknown command: " )@screen ;
    out( command )@screen ;
    out( "\n" )@screen
   else
    # nick was entered as a string
    nickname := (logloc) nick;
    if command = "ENTER" then
     if subscribe( chat_server, nickname ) then
      out( "Succeeded command: " )@screen ;
      in( !userList )@self ;
      UpdateUserList( userList )
     else
      out( "Failed command: " )@screen
     endif
    else # it is a LEAVE
     unsubscribe( chat_server, nickname ) ;
     out("command", "removeAll")@usersList
    endif ;
    out( command, nickname )@screen
   endif
  enddo
 end

rec HandleMessageKeyboard[]
 declare
  const ID := "messageKeyboard" ;
  var message : str ;
  var selected : str ;
  var selectedUsers : ts ;
  locname messageKeyb, usersList, server
 begin
  while ( true ) do
   in( !message )@messageKeyb ;
   # is there someone selected?
   out( "command", "getSelectedItem", ID )@usersList ;
   in( "command", "getSelectedItem", ID, !selected )@usersList ;
   if ( selected != "" ) then
    newloc( selectedUsers ) ;
    out( selected )@selectedUsers ;
    # there's some one selected
    out( "PERSONAL", message, selectedUsers, *self )@server
   else
    out( "command", "getSelectedItems", ID )@usersList ;
    in( "command", "getSelectedItems",
       ID, !selectedUsers )@usersList ;
    if readp( !selected )@selectedUsers then
     # there's some one selected
     out( "PERSONAL", message, selectedUsers, *self )@server
    else
     # no one selected: broadcast
     out( "MESSAGE", message, *self )@server
    endif
   endif
  enddo
 end
```

Listing 6.3: Node coordinators and processes running on a chat client.

Screenshot 6.1 Three chat clients and the chat server.



localities `serverKeyb` and `messageKeyb`. For each of these localities there is a process, respectively `HandleServerKeyboard` and `HandleMessageKeyboard` (also in Listing 6.3) that read the input of the user and communicate with the server. When `HandleServerKeyboard` reads a tuple of the shape `("ENTER", nickname)` it tries to subscribe at the chat server with that specific nickname. On the contrary, if the tuple contains `"LEAVE"` it unsubscribes.

A user can specify that a chat message is destined only to a restricted number of clients by selecting them from the list of connected clients. Such list is indeed attached to the locality `usersList` that, in turn, is a special tuple space that provides a sort of interface for accessing the items of such list (in the KLAVA implementation this tuple space is an interface for a `java.awt.List` object). Thus a process can access the elements of such a list through tuples that start with the string `"command"` and consist of a specific command and its arguments. For each command the template of the tuple is different. If the result of a command has to be retrieved the request is issued with an **out** and the response retrieved with an **in**. An identifier has to be provided so that a process does not retrieve the result of the request of another process. For instance the following two lines retrieve multiple selected items in the list (the result is stored in the **ts** variable `selected`):

out(`"command"`, `"getSelectedItem"`, ID)@usersList ;
in(`"command"`, `"getSelectedItem"`, ID, !selected)@usersList ;

If there is some client selected in this list, the message is sent as `"PERSONAL"` and the list of recipients is sent along with the message; otherwise the message is considered destined to all connected clients.

Screenshot 6.1 shows three chat clients and the chat server.

7 Mobility and Object-Oriented Code

Before describing the object-oriented features of X-KLAIM, we have to introduce the programming philosophy for object-oriented code mobility and in particular we have to sketch the basic features of MOMI a calculus that allows to exchange object-oriented mobile code structured

through mixin-based inheritance. The MOMI programming philosophy has been adopted to add object-oriented features to KLAIM, thus obtaining O'KLAIM (*Object-Oriented* KLAIM first presented in [Bettini *et al.*, 2001a] and refined in [Bettini, 2003a; Bettini *et al.*, 2004b]). Indeed, the object-oriented features of X-KLAIM are based on O'KLAIM.

7.1 Design Issues

First, let us consider 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* [Gamma *et al.*, 1995]: 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 at compile time. 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 some 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 [Bettini *et al.*, 2003a].

Summarizing, mobile object-oriented code needs to be much more flexible than locally developed applications. To this aim we propose a novel solution which is based on a mixin approach with subtyping 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.

7.2 MOMI and O'KLAIM basic concepts

MOMI was introduced in [Bettini *et al.*, 2002b] and extended in [Bettini *et al.*, 2003b]. The underlying motivating idea is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not scale well to distributed contexts with mobility. MOMI's approach consists in structuring mobile object-oriented code by using mixin-based inheritance (a mixin is an incomplete class parameterized over a superclass, see [Bracha & Cook, 1990; Flatt *et al.*, 1998]); this fits with the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that must access some files, can be completed with a base class in order to provide access methods 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 access to sensible resources, as in a *sand-box*). Therefore, MOMI is a combination of a core coordination calculus and an object-oriented mixin-based calculus equipped with types. The key rôle in MOMI's typing is played by a *subtyping* relation that guarantees safe, yet flexible and scalable, code communication, and lifts type soundness of local code to a global type safety property. In fact, we assume that the code that is sent around has been successfully compiled and annotated with its static type. When the code is received on a site (under the hypothesis that the local code has been successfully compiled, too), it is accepted only if its type is subtyping-compliant with the expected one. If the code is accepted, it can be integrated with the local code under the guarantee of no run-time errors, and without requiring any further type checking of the whole code. MOMI's subtyping relation involves not only object subtyping, but also a form of class subtyping and mixin subtyping: therefore, subtyping hierarchies are provided along with the inheritance hierarchies. It is important to notice that we are not violating the design rule of keeping inheritance and subtyping separated, since mixin and class subtyping plays a pivotal role only during the communication, when classes and mixins become genuine run-time polymorphic values.

In synthesis, MOMI consists of:

1. the definition of an object-oriented "surface calculus" with types called SOOL (*Surface Object-Oriented Language*), that describes the essential features that an object-oriented language must have to write mixin-based code;

2. the definition of a subtyping relation on the class and mixin types of the above calculus, to be exploited dynamically at communication time;

3. a very primitive coordination language based on a synchronous send/receive mechanism, to put in practice the communication of the mixin-based code among different site.

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

Table 4: Syntax of SOOL.

O'KLAIM [Bettini *et al.*, 2004b] is the integration of SOOL and its subtyping (both described in the next section), within KLAIM, which offers a much more sophisticated, complete, and effective coordination mechanism than the toy one of MOMI.

SOOL is defined as a standard class-based object-oriented language supporting mixin-based class hierarchies via *mixin definition* and *mixin application*. It is important to notice that specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax of the kernel calculus SOOL (shown in Table 4), including the essential features a language must support to be O'KLAIM's object-oriented component.

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

```
M = mixin
  expect [n : τ]
  redef [m2 : τ2 with ... next() ...]
  def [m1 = ... n() ...]
end

C = class
  [n = ...
  m2 = ...]
end

(new (M ◇ C))⇐ m1()
```

Each mixin consists of three parts:

1. methods defined in the mixins, like m_1;
2. *expected methods*, like n, that must be provided by the superclass;
3. *redefined methods*, like m_2, where *next* can be used to access the implementation of m_2 in the superclass. The application $M \diamond C$ constructs a class, which is a subclass of C.

The key idea of SOOL's typing is the introduction of a novel subtyping relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtyping relation is used to match dynamically the actual parameter's types against the formal parameter's types during communication. The subtyping relation $\sqsubseteq$ is informally defined as follows: for classes, it is naturally induced by the (width) subtyping on record types, while for mixins, subtyping permits the subtype to define more 'new' methods; prohibits to override more methods; and enables a subtype to require less expected

methods. Notice that in O'KLAIM and in X-KLAIM only *subtyping-in-width*, i.e., subtype records can define more methods, but cannot change the signatures of already existing methods[3].

In order to obtain O'KLAIM, the syntax of tuples is extended in order to include any object-oriented value v (defined in Table 4). Actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ can be used to move object-oriented code (together with the other KLAIM items) from/to a locality ℓ, respectively. The object-oriented subtyping, described above, will be used during pattern matching.

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 above. 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:

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

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.

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

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.

8 Object-Oriented Features

The syntax of object-oriented features of X-KLAIM are shown in Table 5. So, the syntax of X-KLAIM processes is extended with object-oriented operations, and basically the syntax of method bodies is the same of the one of an X-KLAIM process (apart from the `return` statement). Notice that, consistently with the nature of the language X-KLAIM, methods are explicitly typed (so the compiler does not have to perform type inference, but only type checking). Furthermore, since in this version of the implementation depth subtyping is not dealt with, a method can only redefine the implementation and not the signature.

The keyword `self` is already a reserved word in X-KLAIM, so we must use `this` in its place to refer to the host object. Furthermore, in method invocation, the receiver object must always be specified, also when a method is called from a method itself, to solve possible ambiguities with process call in X-KLAIM. Finally, in X-KLAIM, class and mixin definitions are allowed to declare also fields. These fields are always considered *private*. For simplicity, in SOOL, we did not

[3] An extension of SOOL with *subtyping-in-depth* can be found in a preliminary form in [Bettini *et al.*, 2003b]. Subtyping-in-depth offers a much more flexible communication pattern, but it complicates the object-oriented code exchange for problems similar to the "subtyping-in-depth versus override" matter of the object-based languages (see [Abadi & Cardelli, 1996; Bettini *et al.*, 2003b] for examples).

typename	::=	**type class** id methoddecls **end**
	|	**type mixin** id mixinmethoddecls **end**
class	::=	**class** id { **declare** fields } methods **end**
mixin	::=	**mixin** id { **declare** fields } mixinmethods **end**
field	::=	**const** id **:=** expression
	|	**locname** id
	|	**var** idlist **:** type
type	::=	xklaimtype | **object** id | **class** id | **mixin** id
method	::=	methoddecl methodbody
methodbody	::=	{ localvars } **begin** methodactions **end**
methoddecl	::=	id **(** parameters **)** { **:** type }
mixinmethod	::=	defredefdecl methodbody
	|	expectedmethod
defredefdecl	::=	(**def** | **redef**) methoddecl
expectedmethod	::=	**expect** methoddecl
mixinmethoddecl	::=	defredefdecl | expectedmethod
methodaction	::=	processaction | **return** exp
exp	::=	xklaimexp | **new** exp | exp <> exp | methodcall | **this** | **this**.id
processaction	::=	xklaimaction | methodcall
methodcall	::=	exp **.** id **(** arguments **)** | **next (** arguments **)**

Table 5: X-KLAIM syntax for MOMI features. Symbols of the shape *xxxs*, such as "parameters" and "arguments", are intended as (possibly empty) lists of *xxx*, separated by the appropriate separator.

treat fields. However, in a real object-oriented language, fields are a very useful feature; in our implementation, as fields are private, they have a minimal impact on the theory of MOMI since they do not appear in the exported interface of classes, mixins and objects.

In the extended version of X-KLAIM class and mixin names, that are used for specifying a type (e.g., **object** id, **class** id, **mixin** id) in a variable or parameter declaration, are only a shortcut for their actual interface. Thus, when performing type checking and structural subtyping, internally, the compiler replaces a class (resp. mixin) name with the corresponding class (resp. mixin) type. This is consistent with one of the principal aims of MOMI, i.e., flexibility in mobile object-oriented code exchange: a class name would be meaningless in a remote site where that name is not known, and if that name would be known also its code would be available at the remote site, and code exchange would be useless. Obviously there must be some types on which all nodes that want to exchange object-oriented code have to agree upon, and by default these are the basic types. In order to defined interfaces, one can use the syntax of *typename*, which is basically similar to a class or mixin definition, apart from the fact that only method signatures can be specified.

This enables a remote site, for instance, to ask for a class providing specific methods with specific types, without any requirements on the name of such a class. The same obviously holds for mixins. This also respects the "open" nature of mixins that are incomplete subclasses, which allow an easier dynamic construction of class hierarchies. In a sense, this structural subtyping philosophy tends to merge the benefits of inheritance with the flexibility of *generic programming* [Musser & Stepanov, 1989; Backhouse *et al.*, 1999].

An object can be declared as follows:

var my_obj : object C

that declares `my_obj` as an object of a class with the interface of `C`. Thus it can be assigned also an object of a class whose interface is a subtype of the one of `C`, as hinted above. For instance, suppose to have the following code:

```
type class C
  m(i : int) : str;
  n()
end
;
class CImpl
```

```
  m(i : int) : str begin ... end;
  n() begin ... end;
  p() begin ... end
end
...
var my_obj : object C;
my_obj := new CImpl
```

then `my_obj` can be assigned an instance of class `CImpl` since its interface is a subtype of `C`.

Class and mixins names can be used as expressions for creating objects, for specifying a type (as in the above declaration) and for delivering code to a remote site. Higher-order variables of kind class and mixin can be declared similarly:

```
var my_class : class C;
var my_mixin : mixin M
```

The above declarations state that `my_class` (`my_mixin`) represents a class (mixin) with the same interface of `C` (`M`). Once initialized, these variables can be used where class and mixin names are expected:

```
my_class := C;
my_obj := new my_class; # same as new C
my_obj := new (my_class <> M); # provided that the application is well−typed
my_mixin := M;
my_obj := new (my_class <> my_mixin); # same as above
```

An object declaration such as

```
var my_obj : object M
```

is correct even when `M` refers to a mixin definition. This does not mean that an object can be instantiated from a mixin directly; indeed the following code is rejected by the compiler:

```
my_obj = new M   # ERROR: class expression required
```

However such a declared object can be instantiated with a class created via the application of `M` to a (correct) superclass:

```
my_obj = new (M <> C)   # OK, provided the application is well typed
```

Indeed, the declared object will be considered as having the same interface of the mixin `M`, that is the interface of a class having all the defined, redefined and expected methods of `M`.

A simplification adopted in this release of the prototype implementation is that recursive types are not handled; thus, when compiling a class of the shape

```
class C
  clone_c() : object C
  begin ... end
end
```

the `xklaim` compiler issues an error when compiling the method `clone_c` saying that the class `C` is not defined. This also shows that classes and mixins have to be defined in the order in which they are needed.

Fields declared in class and mixin definitions are always considered as `private` so they are accessible only by the methods of the same definition. This is not a limitation, since a definition can easily export their values by means of *get/set* methods. Constructors are not dealt with explicitly in this release, so a definition has to provide specific methods for field initializations that are to be called explicitly after object instantiation.

During type-checking, the compiler checks that no name clash among method names occur either in mixin (and class) definitions and in mixin applications. Actually, this check is not limited to names of method defined in the interface of a class or a mixin, but extends to all method names occurring also in types of methods (e.g., a method receiving or returning an object with a specific interface). The theoretical and design motivations behind this choice can be found in [Bettini *et al.*,

2004c], we only hint here that, without rejecting these kind of programs, possible ambiguities (and thus, possible type errors) can otherwise occur at run-time when substituting a mixin (resp. a class) variable with an actual mixin (resp. a class) received from the network.

For instance, consider these declarations:

```
1  class C
2   s() begin ... end
3  end
4  ;
5  mixin M
6   def s() begin ... end ;
7   expect t(a: object C)
8  end
```

the compiler will report the following error (the other lines try to help in figuring out where the problem comes from):

```
6: s, method name occurs in other method type
2: s, declared here
7: occurs in the type of a
7: used from here
```

Infact, there's a name clash between the method `s` defined in the mixin, and the method name `s` that occurs in the type of the expected method `t` that takes a parameter of an object type with the same interface of the class `C` that defines a method `s`.

The same checks is performed also when type checking mixin applications, and can detect name clashes that would be quite hard to discover manually, as in the following example:

```
1  mixin CC
2   def m() begin ... end
3  end;
4  class C
5   f( b : object CC ) begin ... end
6  end;
7  mixin M
8   def m() begin ... end
9  end;
10 class WithNameClashApp
11  bang()
12   begin
13    var o : object M;
14    o := new (M <> C)
15   end
16 end
```

where a name clash is correctly discovered by the compiler:

```
14: method name clash in mixin application
8: m, method name occurs in other method type
2: m, declared here
5: occurs in the type of b
5: used from here
```

8.1 Programming examples

In this section we present the implementations of the examples (sketched in O'KLAIM in Section 7) in X-KLAIM extended with MOMI features. We would like to point out that these are still toy examples since we do not concentrate on the operations performed in method bodies, but on the code exchange and integration.

```
# this is the mixin actually sent to the remote site
# MyPrinterAgent <: PrinterAgent
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;
  var sent_mixin : mixin MyPrinterAgent
 begin
  print "sending printer agent to " + server;
  sent_mixin := MyPrinterAgent;
  out(sent_mixin)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
# generic interface with which a printer agent is received
type mixin PrinterAgent
 expect print_doc(doc : str) : str;
 def start_agent() : str
end

# the following class provides print_doc, so a PrinterAgent can be
# applied to it.  Notice that it also provides another method, init()
# that is ignored by the mixin
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
  print "waiting for a PrinterAgent mixin...";
  in(!rec_mixin)@self;
  print "received " + rec_mixin;
  result := (new rec_mixin <> LocalPrinter).start_agent();
  print "result is " + result;
  out(result)@self
 end
```

Listing 8.1: The printer agent example (left: the sender site - the printer client, right: the receiver site - the printer server).

A mobile printer agent

The programming example shown in this section involves mixin code mobility, and implements a sort of "dynamic adoption" since the received mixin si then applied to a local parent class at run-time.

We assume that a site provides printing facilities for local and mobile agents. As hinted in Section 7 the 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 execution engine 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, thus any mixin that is a subtype to 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 8.1 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 respects (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent`. In particular this mixin will print a document on the printer of the server after preprocessing it. 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.

```
# FSAccessAgentRemote <: FSAccessAgent
class FSAccessAgentRemote
 declare
  var result : str
 copy(src : str) : str
 begin
  return src
 end;
 move(src : str) : str
 begin
  # code for removing the file omitted
  return src
 end;
 start_agent() : str
 begin
  result := "copied " + this.copy("foo");
  result := result + ", moved " + this.move("bar");
  result := result + ", moved " + this.move("system");
  return result
 end
end

rec SendFSAgent[server : loc ]
 declare
  var response : str;
  var sent_agent : class FSAccessAgentRemote
 begin
  print "sending fs agent to " + server;
  sent_agent := FSAccessAgentRemote;
  out(sent_agent)@server;
  in(!response)@server;
  print "response is " + response
 end
```

```
# generic interface for accessing a file system
type class FSAccessAgent
 copy(source : str) : str;
 move(source : str) : str;
 start_agent() : str
end

# redefine move so that on critical files performs only a copy
mixin FSSandBox
 expect start_agent() : str;
 expect copy(source : str) : str;
 redef move(source : str) : str
 begin
  if source != "system" then
   return next(source)
  else
   return "not moved(" + this.copy(source) + ")"
  endif
 end
end

rec ReceiveFSAgent[]
 declare
  var rec_agent : class FSAccessAgent;
  var redefined_agent : class FSAccessAgent;
  var result : str
 begin
  print "waiting for a FSAgentAgent class...";
  in(!rec_agent)@self;
  redefined_agent := FSSandBox <> rec_agent;
  result := (new redefined_agent).start_agent();
  print "result is " + result;
  out(result)@self
 end
```

Listing 8.2: The sand-box example (left: the sender site, right: the receiver site).

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 (remember that in X-KLAIM class and mixin definition names are only shortcut for their actual types). Furthermore, the sent mixin can also define more methods than those specified in the receiving site, thanks to the mixin subtyping relation (described in Section 7.2). 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), and also avoiding dynamic name clashes.

The code for `MyPrinterAgent` is actually unknown at the server site, and it is actually sent during the communication, together with statically built type for the definition of `MyPrinterAgent`. This code mobility feature is guaranteed by the KLAVA framework and it is transparent to the package `momi`.

A sand-box for mobile agents

This example involves class mobility and "dynamic inheritance" since such a class is used in a mixin application at the remote site, and some method redefinitions take place.

In this example a site allows mobile agents to access its own file system; however it wants to enforce some form of restrictions, so that sensible files (e.g., system files) are not modified or erased. In order to do that the class code of mobile agents that is accept by the site has to respect a specific interface; then, a local mixin is applied to the received class in order to redefine the `move` operation so that, if it acts on a system file, the operation is still allowed but it is transformed in an innocuous `copy` operation.

The implementation of this example is in Listing 8.2. Notice that the receiver site redefines method `move` trasforming it in `copy` when it accesses `"system"`, while for all the other files it simply relies on the previous implementation, by calling **next**. As expected, the result that is

delivered to the sender site is:

```
response is copied foo, moved bar, moved not moved(system)
```

Let us remark once again, that these are toy examples that basically aim at showing class and mixin code exchange and dynamic sublcass creations. For instance in a real implementation, the mixin `FSSandBox` should not simply rely on `copy`: a malicious agent could actually remove a file after copying it from within method `copy`.

9 Installation

`xklaim` comes with sources being under the GPL license, thus you have to first unpack the tarball file `xklaim-x.x.x.tar.gz`, where `x.x.x` is the release version, in an appropriate directory. Then, once you entered that directory, it can be compiled and installed like any other GNU programs, i.e., with the typical command sequence:

```
./configure
make
make install
```

remember that by default this will install the program and all its files starting from the directory `/usr/local`, thus you have to be super user in order to do that. Otherwise, should you want to perform the installation in a different (possibly personal) directory, say `/myhome/usr`, you have to pass this option to the `configure` script:

```
./configure --prefix=/myhome/usr
```

Optionally, before `make install`, you may want to run `make check`, that tries to compile some programs preprocessed with `xklaim`. Notice that you will experience problems if you have a version of `gcc` earlier than `3.x`, due to a non-standard treatment of `using` clause for explicitly inheriting overloaded methods from a super class. So you need to install a more recent version of `gcc`.

References

ABADI, M., & CARDELLI, L. 1996. *A Theory of Objects*. Springer.

ANDERSON, B. G., & SHASHA, D. 1992. Persistent Linda: Linda + Transactions + Query Processing. *Pages 93–109 of:* BANATRE, J. P., & LE METAYER, D. (eds), *Proc. of Research Directions in High–Level Parallel Programming Languages*. LNCS, vol. 574. Springer.

BACKHOUSE, R., JANSSON, P., JEURING, J., & MEERTENS, L. 1999. Generic Programming –An Introduction–. *Pages 28–115 of:* SWIERSTRA, S.D., HENRIQUES, P.R., & OLIVEIRA, J.N. (eds), *Revised Lectures 3rd Int. School on Advanced Functional Programming, AFP'98, Braga, Portugal, 12–19 Sept. 1998*. LNCS, vol. 1608. Springer.

BETTINI, L. 1998 (April). *Progetto e Realizzazione di un Linguaggio di Programmazione per Codice Mobile*. Master thesis, Dip. di Sistemi e Informatica, Univ. di Firenze.

BETTINI, L. 2003a. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. Ph.D. thesis, Dip. di Matematica, Università di Siena. Available at `http://music.dsi.unifi.it`.

BETTINI, L. 2003b. KLAVA*: a Java package for distributed and mobile applications. User's manual*. 1 edn. Dip. di Sistemi e Informatica, Univ. di Firenze. Available at `http://music.dsi.unifi.it/klava`.

BETTINI, L., & DE NICOLA, R. 2001. Translating Strong Mobility into Weak Mobility. *Pages 182–197 of:* PICCO, G. P. (ed), *Mobile Agents*. LNCS, no. 2240. Springer.

BETTINI, L., DE NICOLA, R., FERRARI, G., & PUGLIESE, R. 1998. Interactive Mobile Agents in X-KLAIM. *Pages 110–115 of:* CIANCARINI, P., & TOLKSDORF, R. (eds), *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. IEEE Computer Society Press.

BETTINI, L., DE NICOLA, R., FERRARI, G., & PUGLIESE, R. 2000. Mobile Applications in X-KLAIM. *In:* CORRADI, A., OMICINI, A., & POGGI, A. (eds), *Proc. of WOA 2000*.

BETTINI, L., BONO, V., & VENNERI, B. 2001a. Towards Object-Oriented KLAIM. *In:* [Lenisa & Miculan, 2001].

BETTINI, L., DE NICOLA, R., & PUGLIESE, R. 2001b. X-KLAIM and KLAVA: Programming Mobile Code. *In:* [Lenisa & Miculan, 2001].

BETTINI, L., LORETI, M., & PUGLIESE, R. 2002a. An Infrastructure Language for Open Nets. *Pages 373–377 of: Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*. ACM.

BETTINI, L., BONO, V., & VENNERI, B. 2002b. Coordinating Mobile Object-Oriented Code. *Pages 56–71 of:* ARBARB, F., & TALCOTT, C. (eds), *Proc. of Coordination Models and Languages*. LNCS, no. 2315. Springer.

BETTINI, L., DE NICOLA, R., & PUGLIESE, R. 2002c. KLAVA: a Java package for distributed and mobile applications. *Software – Practice and Experience*, **32**(14), 1365–1394.

BETTINI, L., LORETI, M., & VENNERI, B. 2003a. On Multiple Inheritance in Java. *Pages 1–15 of:* D'HONDT, T. (ed), *Technology of Object-Oriented Languages, Systems and Architectures, Proc. of TOOLS Eastern Europe 2002*. Kluwer Academic Publishers.

BETTINI, L., BONO, V., & VENNERI, B. 2003b. Subtyping Mobile Classes and Mixins. *In: Proc. of Int. Workshops on Foundations of Object-Oriented Languages, FOOL 10*.

BETTINI, L., DE NICOLA, R., & LORETI, M. 2004a. Formulae Meet Programs Over the Net: A Framework for Correct Network Aware Programming. *Automated Software Engineering*, **11**(3), 245–288. Special Issue on Distributed and Mobile Software Engineering.

BETTINI, L., BONO, V., & VENNERI, B. 2004b. O'KLAIM: a coordination language with mobile mixins. *Pages 20–37 of: Proc. of Coordination 2004*. LNCS, vol. 2949. Springer.

BETTINI, L., BONO, V., & VENNERI, B. 2004c. Subtyping-Inheritance Conflicts: The Mobile Mixin Case. *In: Proc. of Third IFIP International Conference on Theoretical Computer Science (TCS 2004)*. Kluwer Academic Publishers. To appear.

BRACHA, G., & COOK, W. 1990. Mixin-based Inheritance. *Pages 303–311 of: Proc. OOPSLA '90*.

BUTCHER, P., WOOD, A., & ATKINS, M. 1994. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, **6**(6), 505–516.

CARRIERO, N., & GELERNTER, D. 1989a. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, **21**(3), 323–357.

CARRIERO, N., & GELERNTER, D. 1989b. Linda in Context. *Communications of the ACM*, **32**(4), 444–458.

CUGOLA, G., GHEZZI, C., PICCO, G.P., & VIGNA, G. 1997. Analyzing Mobile Code Languages. *In:* VITEK, J., & TSCHUDIN, C. (eds), *Mobile Object Systems*. LNCS, no. 1222. LNCS.

DE NICOLA, R., FERRARI, G., & PUGLIESE, R. 1998. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, **24**(5), 315–330.

FLATT, M., KRISHNAMURTHI, S., & FELLEISEN, M. 1998. Classes and Mixins. *Pages 171–183 of: Proc. POPL '98*.

GAMMA, E., HELM, R., JOHNSON, R., & VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GELERNTER, D. 1985. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, **7**(1), 80–112.

GELERNTER, D. 1989. Multiple Tuple Spaces in Linda. *Pages 20–27 of:* ODIJK, E., REM, M., & SYRE, J. (eds), *Proc. Conf. on Parallel Architectures and Languages Europe (PARLE 89)*. LNCS, vol. 365. Springer.

HOHLFELD, M., & YEE, B.S. 1998. *How to Migrate Agents*. Available at `http://www.cs.ucsd.edu/~bsy`.

KORPELA, E., WERTHIMER, D., ANDERSON, D., COBB, J., & LEBOFSKY, M. 2001. SETI@home: Massively Distributed Computing for SETI. *IEEE Computing in Science and Engineering*, January.

LENISA, M., & MICULAN, M. (eds). 2001. *TOSCA 2001*. ENTCS, vol. 62. Elsevier.

MUSSER, D.R., & STEPANOV, A.A. 1989. Generic programming. *Pages 13–25 of:* GIANNI, P. (ed), *Proc. Symbolic and algebraic computation: International Symposium ISSAC '88*. LNCS, vol. 358. Springer.

Index