

An Ada95 Implementation of a Network Coordination Language with Code Mobility

Emilio Tuosto

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56100 Pisa - Italy
e-mail: etuosto@di.unipi.it

Abstract. One of the principal aims of distributed programming research is the definition of paradigms which permits the description of *Global Computation*, as Cardelli calls them, i.e. computations on a net of heterogeneous sites. In this context, it seems that code mobility is a good paradigm for limiting network traffic.

Here we will describe the implementation, in *Ada95*, of *Klaim*, a kernel language that uses code mobility and gives the possibility of coordinating the activity of processes running on a net.

Keywords. Ada Language and Tools, Ada Experience Reports, Case Studies and Experiments, Ada and other Languages, Distributed Systems.

1 Introduction

Defining suitable languages for writing *global computations* (computations operating on resources distributed on *global computers* Car96) is one of the aims of distributed programming research Car95,Car96,CG97,DFP97a,DFP97b,DFP98,GV97,Whi94. In these languages three fundamental aspects must be considered:

1. Network Traffic Minimization;
2. Network Heterogeneity;
3. Network Security.

In this paper, the third aspect will not be considered.

Recently, the interest in paradigms which can specify some form of code mobility has grown thanks to their intrinsic ability in limiting, at least in some cases, network traffic. For example, a distributed application which elaborates a large quantity of data (distributed on different physical sites) can be made more efficient if a paradigm that allows *agent*¹ movement on the sites containing data is used.

As in Vig98, we divide the distribution models in two classes: *Traditional Distributed Systems* (Fig. 1) and *Mobile Code Systems* (Fig. 2). In a traditional distributed system, the *True Distributed System* layer (TDS) hides the physical

¹ Here we use the term *agent* to indicate the code that can be moved across a network.

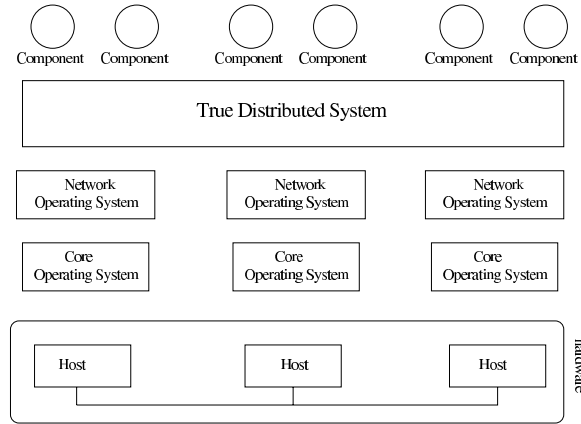


Fig. 1. Traditional Distributed System.

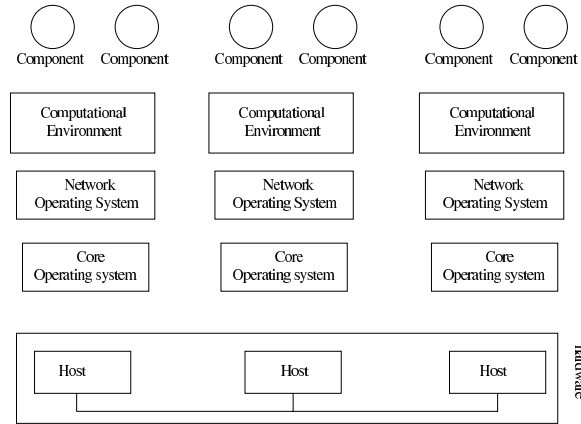


Fig. 2. Mobile Code System.

structure of the network (its sites and connections) from the application layer. A typical example of TDS is the mechanism adopted in CORBA OMG95 which renders an object visible on a net. In a mobile code system, the TDS layer is replaced by the *Computational Environment* (CE) layer, which is an abstraction of sites for the applications.

In this paper *Klada*, an implementation of *Klaim* DFP98, will be presented. *Klaim* is a kernel coordination language of agent interaction and mobility which derives from *Linda* CG89 and *Llinda* DFP97a, *Klaim* processes interact via a “blackboard” communication model. In fact, the data are exchanged by visiting a multiset of tuples (finite sequences of values and variables). This multiset is called *Tuple Space* (TS) and it is distributed on network sites. Processes can be used as fields of tuples. Language communication primitives select the TS to use by an explicit notion of locality. Localities are first class objects, as processes are,

so they can be exchanged in communications. `Klaim` processes can be composed by a non-deterministic choice operator and by a parallel operator.

We will provide a compiler which transforms `Klada` programs to `Ada95`. Given a `Klada` program, the compiler generates :

1. the partitions of the corresponding `Ada95` distributed program;
2. a configuration file which when passed to `Glade`, a tool developed by GNU organization that implements the directives described in `ARM95`, permits the execution of `Ada95` programs on (heterogeneous) networks.

The principal features of `Klada` that require some peculiarities of `Ada95` are those regarding the dynamic characteristics of `Klada` nets. In particular, implementing the `newloc` primitive requires the use of remote access type objects and the introduction of a *Name.Manager*.

The languages that provide code mobility may be divided in two classes: *strong mobility* and *weak mobility* languages. The first class includes the languages in which it is possible to move execution threads of control among sites. An example of a strong mobility language is `Telescript` Whi94. The languages in which the remote code execution is obtained by activating a new thread that is independent from the thread that moves the code, belong to the second class. `Klada` is a weak mobility language. In the implementation, mobility is obtained by introducing a `Klada` interpreter, `kvm`. Each `Ada95` partition implementing CE of `Klada` sites is linked to `kvm`.

The definition and the implementation of `Klada` are part of a thread of research in the field of concurrent and distributed languages. Other models of concurrency and distribution have been proposed and implemented. For example we can indicate `Pict` PT97, an implementation of (a variant of) the π -calculus described in MPW92, `Oblig` Car94, Car95, a lexically-scoped untyped interpreted language that supports distributed object-oriented computation, `FACILE` GMP89, `TLG92` an extension of the `ML` functional language with communication primitives. Moreover, `Klaim` has also a `Java` implementation defined in BDFP98.

2 Klada

An informal description of the actions of `Klaim` processes follows:

- `in(t)@l`: evaluates the tuple t and searches for a matching tuple t' in the TS denoted by locality l . Whenever t' is found, it is removed from the TS and the its values are assigned to the corresponding variable fields of t , then the operation terminates. If no matching tuple is found, the operation is suspended until one is available.
- `read(t)@l`: similar to `in`, but after the assignment to the variables of t , the tuple t' is not removed from the TS.
- `out(t)@l`: evaluates the tuple t and adds it to the TS located at l . The `out` primitive needs no synchronization: it is an asynchronous action.
- `eval(P)@l`: agent P is executed on the site named l . This corresponds to the remote code evaluation. Like `out`, `eval` is asynchronous.

- **newloc**(u): creates a “fresh” site that can be accessed via the locality variable u . Also **newloc** is a non-blocking instruction.

A distinguished logical locality, **self**, is used by **Klaim** processes to denote their execution site.

From this informal description, it can be noted that **Klaim** specifies two code mobility mechanisms: the first is the possibility of exchanging processes as data (**in, read/out**) and the second one is the remote evaluation of processes (**eval**). An important aspect to note is that the localities used by a **Klaim** process are logical names: a programmer, therefore, has an abstract view of the net.

Klaim uses both static and dynamic scoping strategies to evaluate the logical localities of the agents. The agents that occur in tuples fields are evaluated with a static scoping discipline, but, for the code moved using **eval**, a dynamic scoping is adopted. In particular, a table which maps the logical localities on physical sites is associated to the CE representing the **Klaim** sites. When an agent is moved through tuple spaces, the starting CE constructs a “closure”, i.e. a couple of code and a (partial) mapping from logical localities to sites. When the code is executed on a new site and a locality is evaluated, this mapping will be used before the one of the site CE. On the other hand, for **eval**, the closure built by the starting CE has a void mapping and so only the site environment of the executing site will be searched. Moreover, it must be observed that a **Klaim** network may dynamically grow by adding new sites with **newloc**.

Klada processes may be derived from the grammar in Table 1. In Table 2 we give the syntax of **Klada** nets. Types, localities and tuples are specified in Table 3. The usual boolean, numeric and string expressions are not specified. We can see that **Klada** is an extension of **Klaim** with some constructs typically used in imperative programming (**if, while, :=**). This extension, in any case, does not give more expressive power because it is possible to prove that **Klaim** can simulate the new constructs Tuo98. In **Klada**, a new choice operator is defined, *Ext_Ch*, similar to the guarded command of CSP Hoa85. *Ext_Ch* is used for deriving the external choices, i.e. the choices that need an external synchronization to be resolved. *Int_Ch*, the **Klaim** choice operator, is used to choose non-deterministically among some branches without external conditioning. Moreover, the declaration constructs of variables and constants are expressed. [h]

We have used the following shorthands:

- productions such as $A ::= B \mid \epsilon$ will be abbreviated to $A ::= [B]$;
- $[A]^*$ is $\overbrace{A \dots A}^n$, where $n \geq 0$;
- $[A]^+$ is $A [A]^*$;
- **id** stands for a generic identifier.

Let’s make some considerations on the grammar given in Table 2.

A *Net* is a set of *Nodes* that may be preceded by some definitions of process constants (used to define recursive processes). A *Node* is composed by an **id** (the name of a network machine), an allocation environment, *Env*, and the processes

Table 1. Grammar of Klada processes

<i>Task</i>	::= <i>Ext_Ch Ext_Ch</i> <i>Int_Ch</i>
<i>Ext_Ch</i>	::= in (<i>Tuple</i>)@ <i>L.exp</i> . <i>Seq</i> [<i>Ext_Ch</i>] read (<i>Tuple</i>)@ <i>L.exp</i> . <i>Seq</i> [<i>Ext_Ch</i>]
<i>Int_Ch</i>	::= <i>Seq</i> <i>Seq</i> # <i>Int_Ch</i>
<i>Seq</i>	::= <i>Com</i> <i>Com</i> ; <i>Seq</i> call id [(<i>Tuple</i>)] id' nil
<i>Com</i>	::= eval (<i>Task</i>)@ <i>L.exp</i> out (<i>Tuple</i>)@ <i>L.exp</i> in (<i>Tuple</i>)@ <i>L.exp</i> read (<i>Tuple</i>)@ <i>L.exp</i> newloc (<i>L.exp</i>) if <i>Exp</i> then <i>Task</i> else <i>Task</i> fi while <i>Exp</i> do <i>Task</i> od id := <i>Exp</i> (<i>Task</i>)

Table 2. Syntax of the Klada nets

<i>Net</i>	::= define id [[id : <i>Type</i> ;]*] as Const on net <i>Node</i> quit
<i>Const</i>	::= <i>NamedProc</i> <i>NamedProc</i> ; <i>Const</i>
<i>NamedProc</i>	::= rec id [(id : <i>Type</i> ; id : <i>Type</i>)*] declareDec begin Task end
<i>Node</i>	::= id :: { [<i>Env</i>] } <i>ProcDef</i> <i>Node</i> <i>Node</i>
<i>Dec</i>	::= const id := <i>Exp</i> id , id * : <i>Type</i> [:= <i>Exp</i>]
<i>Env</i>	::= id ~ id <i>Env</i> , <i>Env</i>
<i>ProcDef</i>	::= declareDec begin Task end <i>ProcDef</i> <i>ProcDef</i>

allocated on the node. *Env*, the syntactic category for the allocation environment, is a list of pairs of **id**: the first is the (name of) the logical locality and the second represents the (name of) the physical machine (IP address).

In Table 3, the distinguished locality **screen** is used to model the data output on external devices (for the time being, only the screen) using the same philosophy of the tuple spaces. A screen may be thought of as a particular kind of TS, i.e. when we write **out**(*t*)@**screen**, we will be able to printout the data in the tuple *t* on the screen.

Table 3. Types, expressions, tuples and localities

<i>Type</i>	::= int bool str location process
<i>Exp</i>	::= ...
<i>Tuple</i>	::= <i>Exp</i> <i>L.exp</i> !id begin Task end <i>Tuple</i> , <i>Tuple</i>
<i>L.exp</i>	::= self id screen

3 Klada Semantics: A Programming Example

To clarify the Klada semantics, it's better to give an example that shows most of the features of the language. For a formal definition the reader is referred to DFP98.

We will provide an electronic commerce application. We want to program an agent that visits the camera shops which are “nearest” to a distance d , searching for a camera with the lowest price. Camera shops will be modelled by processes such as the following:

```
shop ≡ begin
    out(“camera1”, price1)@self; ... ; out(“camerak”, pricek)@self;
end
```

The *MarketPlaceClient* in Fig. 3, asks the local server *MarketPlace* for the list of camera shops nearest to d (line 5). Once *MarketPlaceClient* has received

```
1. rec MarketPlaceClient (Cam: str; RLoc: loc; d: int) declare
2.   var ShopList, FirstShop: loc;
3.   var NumShop: int
4. begin
5.   out(“Shop list”, d)@self;
6.   in(!ShopList)@self;
7.   in(!NumShop)@ShopList;
8.   if NumShop = 0
9.     then out(Cam, “NO SHOP”, -1)@RLoc
10.    else
11.      in(!FirstShop)@ShopList;
12.      out(NumShop - 1)@ShopList;
13.      eval(call ShopAgent (Cam, ShopList, RLoc, 0))@FirstShop
14.   fi
15. end
```

Fig. 3. *MarketPlaceClient*

the (locality of the) list (line 6), if there is a shop to visit, it remotely evaluates the *ShopAgent* by passing to it the camera type, the list of shops, the locality where the results are to be returned and the current obtained price (lines 7 - 13).

The server *MarketPlace* (Fig. 4) has the list of shops with their distances (line 5) and executes an endless loop waiting for a request (lines 6 - 7). To satisfy a request, *MarketPlace* creates a new site where the shop list will be allocated (line 8), constructs the list (lines 9 - 15), returns the (locality of the) list to the client (lines 16) and prints a message on the screen (line 17).

When *ShopAgent* (Fig. 5) starts its execution on the site of a new shop, it reads the price of the camera (line 5) and, if the new price is lower than the old one, it removes the old price from the TS at RLoc and replaces it with the new price (lines 9 - 15). Then *ShopAgent* reads the number of shops that must be

examined (line 17) and, if a shop exists, it updates the number of shops, takes the address of the next shop and executes a copy of itself on the new shop (lines 21 - 23). The new copy of *ShopAgent* will continue the search in the remaining shops.

```

1. declare
2.   varDist, ShopDist, Pos, PosList: int := 0;
3.   varShopLoc, ShopList: loc
4. begin
5.   out(1, "shop1", d1)@self; ...; out(h, "shoph", dh)@self;
6.   while true do
7.     in("Shop list", !Dist)@self;
8.     newloc(ShopList);
9.     whilePos ≤ h do
10.      read(Pos + 1, !ShopLoc, !ShopDist)@self;
11.      if ShopDist < Dist
12.        then out(ShopLoc)@ShopList; PosList:= PosList + 1; Pos:= Pos + 1
13.        elsePos:= Pos + 1
14.      fi
15.    od;
16.    out(PosList)@ShopList; out(ShopList)@self;
17.    out("List builded...")@screen
18.  od;
19. end

```

Fig. 4. *MarketPlace*

4 Ada95 Distribution Model and Glade

The Ada95 distribution model ARM95 (Annex E) defines the *partition*, the principal distribution unit, as a set of structures and/or algorithms which can be allocated on the nodes of a net. A partition can be considered the union of two logical components. The first component implements the algorithms for the application that one wants to obtain. The second component defines the communication interfaces to/from the other partitions. In the algorithm implementation phase, an Ada95 programmer must satisfy only some constraints required by the language distribution model. Problems concerning partition distribution and communications are application-independent, so they may be faced in a separate phase. During the application development, there is no need for hypotheses about the architecture on which the program will be executed; the association node-partitions are established in the configuration phase.

Partition cooperation is based on a *Remote Subprogram Call* mechanism enriched by the possibility of *dynamic dispatching* which, at run-time, permits the establishment of procedures which are to be executed, according to the values of some actual parameters.

```

1. rec ShopAgent (Cam: str; ShopList: loc; RLoc: loc; Price: int) declare
2.   var NewPrice, NumShop: int;
3.   var OldShop, NextShop: loc
4. begin
5.   read(Cam, !NewPrice)@self;
6.   if Price = 0
7.     then out(Cam, self, NewPrice)@RLoc
8.     else
9.       if NewPrice < Price
10.        then
11.          in(Cam, !OldShop, !Price)@RLoc;
12.          out(Cam, self, NewPrice)@RLoc;
13.          Price := NewPrice
14.        elsenil
15.      fi
16.   fi;
17.   in(!NumShop)@ShopList;
18.   if NumShop = 1
19.     then out("go")@RLoc
20.     else
21.       out(NumShop - 1)@ShopList;
22.       in(!NextShop)@ShopList;
23.       eval(call ShopAgent (Cam, ShopList, RLoc, Price)@NextShop
24.     fi
25. end

```

Fig. 5. *ShopAgent*

The distributed systems annex does not describe how a distributed application should be configured. The tool **Glade** and its configuration language have been purposely designed to allow to specify the partition definition and the machines where each partition must be executed. **Glade** combines the distributed and object-oriented features of **Ada95**, it reads a configuration file and builds several executables, one for each partition. Using **Glade** it is possible to create applications where objects are physically distributed over a network of heterogeneous machines (without having to interface to any low-level communication layer), to support different network protocols, and to provide replication and fault-tolerance.

Therefore, it can be stated that the **Ada95** distribution model is a traditional distributed system, whereas **Klaim** may be considered an hybrid between traditional systems and mobile code systems because the application programmers have an abstract view of the network and can address network nodes using logical localities.

The choice of **Ada95** to implement **Klaim** is an attempt to study the “naturalness” of programming a support of the mechanisms offered by a mobile code system by using a traditional distributed system.

5 Implementation

In this section we will present some of the implementation choices adopted in order to realize **Klada**. In particular we will discuss the implementation of code mobility and **newloc** primitive. Other aspects, like external choice and the representation of the tuple spaces, will not be faced here.

5.1 The Compiler

A **Klada** program N describes a net on which some processes are (statically) allocated. For example, the commerce system described in section 3 may be organized as shown in Fig. 6 (the only agent is *ShopAgent*). To make the execution

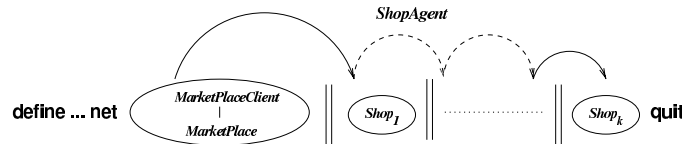


Fig. 6. The electronic commerce net.

of N more efficient we have developed a compiler that generates, for every node expressed in N , an **Ada95** partition in which the code part corresponds to the processes allocated on the node. This also permits some static controls such as **ids** must be declared before use and can't have more than one declaration, all variables must be initialized before use, no assignment to constants is allowed, formal and actual parameters in recursive process calls match, etc. Furthermore, the compiler also performs type checking.

Every partition implementing a node of N is referred by a locally declared remote access object belonging to the class declared by

type *Locality* is abstract tagged limited private

these remote access objects may be considered the **self** counterparts of **Klada**. The methods of *Locality* implement the **Kla_{im}** primitives that may be referred by a remote process (**in/read, out, eval**). For example the code generated by the compiler to translate **in(t)@ ℓ** is

```

    < evaluate  $t$  >;
    < get  $s$ , the remote access object associated to (the name)  $\ell$  >;
    in( $s$ , ...);

```

where **in** is the method that implements **in**. Depending on s , the **Ada95** run-time support will execute the **in** procedure of the partition where s has been created (possibly a remote machine). This is the *dynamic dispatching* mechanism of **Ada95** and it provides remote subprogram calls in which the subprogram is determined at run-time.

As we shall see in the **newloc** implementation, these remote access objects are also used to provide new physical sites to the net.

5.2 Code Mobility

The most important task of the CEs of a MCS language is the preparation of the execution environment that an agent needs during its migrations. When the runtime support of a MCS language has to move an agent from site s_1 to site s_2 , the CE of s_1 will provide the allocation on (CE of) s_2 of the complete environment the code needed for his execution. For example, if on s_1 the following code is running:

```
A : ...; B : ...; begin... eval(C)@s2; ... end;
```

and C uses A but not B , then the CE of s_1 must construct, on (the CE of the) site s_2 , an environment in which the value of A is specified and B may not be defined or have a different definition.

Ada95 has no support for the code mobility. For this reason we are compelled to define an encoding of the mobile agents and a corresponding interpreter for such an encoding (i.e. the Klada CEs need an interpreter to execute agents).

In our implementation, an agent has the following representation:

```
type Agent_Type is record  
  Code : Code_type;  
  Env  : Environment_Type;  
  Mem  : Memory_Type;  
end record
```

When the compiler detects an agent A (code appearing in a field of a tuple or in an **eval** instruction), it generates a *closure*, a representation of A (its parse tree) and the environment for the evaluation of the logical localities referred by A (if it is moved with **eval**, the environment part will be void). Furthermore, the compiler also attaches a “memory” (a private address space) to the closure in which the agent can store and retrieve the values of its variables.

The implementation of code migration is obtained by making a remote call to `kvm`, the Klada interpreter, and by passing it an object, A , of type *Agent_Type*. By visiting the tree A .Code and using A .Env and A .Mem, `kvm` is able to execute the agent represented by A . Therefore, `kvm` also may be thought of as a method of the class *Locality*.

5.3 Dinamic Site Creation

A new (physical) site may be added to a preexisting network executing a *void_site* command from an operating system shell. The *void_site* command executes an Ada95 partition that simply generates its site-address (by allocating a new remote access object), sends it to the *Name_Manager* and waits for a partition that uses the new site.

The *Name_Manager* handles L , the list of new added sites that can be given to the **newloc** invocation it receives (Fig. 7). It must be said, however, that if L is void, *Name_Manager* will create a new “virtual” site on the caller machine. This behaviour is due to the non-blocking semantics of the **newloc** operation

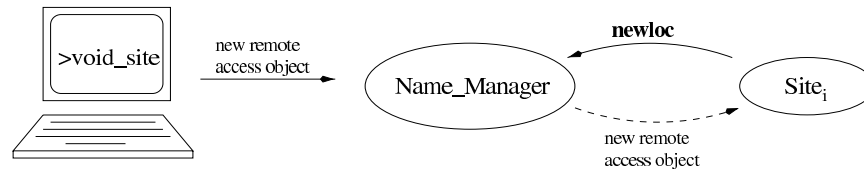


Fig. 7. Creating new sites.

Referring to Fig. 7, once $Site_i$ has received the new site-address, it may, for example, use the new tuple space and its underlying machine for tuple exchanging of remote code evaluations respectively.

The tool we use to distribute the Ada95 programs on a network is **Glade**. The compiler creates an Ada95 partition for every Klada site. Furthermore, a configuration file for **Glade** is generated. According to this configuration file, **Glade** will compile and distribute the partitions on the physical sites specified.

6 Conclusions

We have introduced some aspects of an Ada95 implementation of Klada, a coordination language with code mobility. The purpose of this implementation is to study the “naturalness” that TDS models, like the Ada95 distribution model, have when they are used to realize MCS. What we can say is that Ada95 was quite suitable for the realization of all the features of Klada, but the lack of any support for code mobility compels us to provide an interpreter of Klada agents. Furthermore, the Ada95 distribution model and Glade allow us to concentrate on the implementation of Klaim primitives, without worrying about the communications among different partitions, therefore avoiding low level programming. Some improvements, that in this first release have been neglected, may be introduced. In this connection, it is possible to add a new functionality to the *Name_Manager*: when a site S asks for a new physical site (**newloc**), if there are no available machines, *Name_Manager* allocates on M_S , the machine of S , a new site. In this manner, if the processes on S execute **newloc** many times, M_S would be overloaded. We can modify *Name_Manager* to take account of the network load and select the less used machine. In such a way *Name_Manager* can distribute the load on the whole network.

Klaim and its implementations (Klada and X-Klaim BDFP98) have been proposed for the study of the usefulness of code mobility at the application level. However, some work must be done to provide a true programming language; for example, new types or tools like a debugger or syntax driven editors may be implemented. From this point of view, an interesting direction is the definition of an Ada95 library that simply implements the Klaim primitives and can be used in Ada95 programs. In this way it would be possible to write large distributed applications in Ada95 that use code mobility as programming paradigms.

References

- [ARM95]International Standard ISO/IEC 8652:1995(E). Annotated Ada Reference Manual: Language and Standard Libraries. Intermetrics, Inc., 1994.
- [BDFP98]L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese. Interactive Mobile Agents in XKlaim *Proceedings of WETICE'98*, IEEE, 1998. (To appear)
- [Car94]L. Cardelli. Obliq: A language with distributed scope. SRC Research Report 122, Digital Equipment Corporation Systems Research Center, June 3, 1994.
- [Car95]L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27-59, MIT Press, 1995.
- [Car96]L. Cardelli. Global Computation. *ACM Computing Surveys*, Vol. 28, Number 4es, pp. 163-163, December 1996.
- [CG89]N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, April 1989.
- [CG97]L. Cardelli, A. D. Gordon. Mobile Ambients. FoSSaCS, *LNCS* 1378, pag. 140-155, 1998
- [DFP97a]R. De Nicola, G. Ferrari, R. Pugliese. Locality based Linda: programming with explicit localities. *FASE-TAPSOFT'97, Proceedings* (M. Bidoit, M. Dauchet Eds.), *LNCS* 1214, pp. 712-726, Springer, 1997.
- [DFP97b]R. De Nicola, G. Ferrari, R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. *COORDINATION'97, Proceedings* (D. Garlan, D. Le Metayer, Eds.), *LNCS* 1282, pp. 220-237, Springer, 1997.
- [DFP98]R. De Nicola, G. Ferrari, R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5), pp. 315-330, 1998.
(Available at the URL <http://www.di.unipi.it/~giangi/papers>).
- [GMP89]A. Giacalone, P. Mishra, S. Prasad. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming*, 18(2), 1989.
- [GV97]C. Ghezzi, G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. *Proceedings of the First International Workshop on Mobile Agents (MA97)*, Berlin, Germany, April 1997.
- [Hoa85]C.A.R. Hoare. Communicating Sequential Process. *Prentice Hall Int.*, 1985.
- [MPW92]R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, (part I and II), *Information and Computation*, 100:1-77, 1992.
- [OMG95]Object Management Group. *CORBA: Architecture and Specification*, August 1995.
- [PT97]B. C. Pierce, David N. Turner. Pict: A Programming Language based on the Pi-Calculus *Technical Report*, Computer Science Department, Indiana University, Number CSCI 476, March 1997.
- [TLG92]B. Thomsen, L. Leth, A. Giacalone. Some Issues in the Semantic of Facile Distributer Programming. *REX Workshop "Semantics: Foundations and Applications"* (J. W. de Bakker, W-P. de Roever, G. Rezenberg), *LNCS* 666, pp. 563-593, Springer, 1992.
- [Tuo98]E. Tuosto. Semantica e Pragmatica di un Linguaggio di Coordinamento di Attività su reti. *Master Thesis in Computer Science, University of Pisa, Italy*, 1998.
- [Vig98]G. Vigna. Mobile Code Technologies, Paradigms, and Applications. *Ph.D. Thesis, Politecnico di Milano*, 1998.
- [Whi94]J. E. White. Telescript Technology: Foundation for the Electronic Market Place. *General Magic White Paper*, 1994.