An Infrastructure Language for Open Nets [*]

Lorenzo Bettini   Michele Loreti   Rosario Pugliese
Dipartimento di Sistemi e Informatica, Università di Firenze
Via Lombroso 6/17, 50134 Firenze, Italy
{bettini,loreti,pugliese}@dsi.unifi.it

Abstract

The structure of open nets, like the Internet, is highly dynamic, as the topology of component networks continuously evolves. In this context, *node connectivity* is a key aspect and a language for distributed network-aware mobile applications should provide explicit mechanisms to handle it. In this paper, we address the problem of expressing dynamic changes of node connectivity at linguistic level and, in particular, we focus on a slight extension of the language KLAIM, that is targeted to this aim. The extension consists of the introduction of a new category of processes that, in addition to the standard process operations, can execute a few new coordination operations for establishing new connections, accepting connection requests and removing connections. Our extension puts forward a clean separation between the coordinator level and the user level and, hence, it is modular enough to be easily applicable also to other network-aware languages. We will also show that our approach can be used as a guide for actual distributed (i.e. without a single centralized server) implementations of mobile systems.

Keywords

Open Nets, Distributed Applications, Mobility, Coordination Languages.

1. Introduction

Open nets are, by their own nature, dynamically evolving structures, since new nodes can get connected or existing nodes can disconnect. Connections and disconnections can be temporary and unexpected. For instance, temporary connections can be established "on the fly" among terminals equipped with wireless devices and ad-hoc paths to services and remote resources can be built dynamically among components. In these scenarios, mobile devices such as laptops, PDAs and cellular phones highly rely on a dynamically evolving communication infrastructure, which is able to reconfigure itself. Thus, the assumption that the underlying communication network will always be available is too strong. Moreover, the knowledge of node addresses may not suffice to establish connections or to perform migrations, since network routes may be affected by restrictions (such as temporary failures or firewall policies).

From the point of view of the programming language design, the need arises to extend languages for distributed network-aware mobile applications (see, e.g., [8, 16, 13, 17]) with constructs for explicitly handling changes in the network topology. Thereby, for instance, mobile clients can explicitly express the intention of entering a specific server domain or that of exiting from a particular subnet and, conversely, servers can easily manage clients in their own subnets. With respect to solutions based on, e.g., middleware extensions, using an enriched language model will permit expressing real scenarios more naturally, thus facilitating debugging and property checking of distributed network-aware mobile applications.

In this paper, we present a family of constructs specifically designed for expressing the dynamic evolution of open nets. These constructs are largely independent of any application programming language. However, to put it in a concrete form, we will focus on the integration of such constructs with the language KLAIM [9], an experimental kernel programming language specifically designed to model and to program distributed concurrent applications with code mobility. The obtained language can be used to model, to analyze and to drive the implementation of distributed applications.

The extension exploits the notion of *node connectivity* and, syntactically, consists of the introduction of a new category of processes, called *NodeCoordinators*, that, in addition to the standard process operations, can execute a few new privileged operations. Such operations will permit establishing new connections, accepting connection requests and removing connections. However, they can also be interpreted as movement operations: entering a new administrative domain, accepting incoming nodes and exiting from an administrative domain, respectively. As such, the new operations can be used for expressing the physical mobility of a device, e.g. a PDA crossing some wireless network cells. Of course, the new operations represent a core extension: other powerful constructs for typical client-server applications in open nets can be derived from them (see the example of Section 4).

Our approach puts forward a clean separation between the coordinator level (made up by NodeCoordinator processes) and the user level (made up, in this paper, by standard KLAIM processes). This separation makes a considerable impact. From an abstract point of view, the coordinator level may represent the network operating system running on a specific computer and the user level may represent the processes running on that computer. The new privileged operations are then system calls supplied by the network operating system. From a more implementative point of view, the

[*]This work has been partly supported by MURST Projects *SALADIN* and *TOSCA* and by *Microsoft Research Ltd.*

coordinator level may represent the part of a distributed application that takes care of the connections to a remote server (if the application is a client) or that manages the connected clients (if the application is a server). The user level then represents the remaining parts of the application that can interact with the coordinator by means of some specific protocols. Finally, the separation makes our approach easily applicable also to other languages/calculi for distributed network-aware mobile applications such as, e.g., *Dπ* [12] and *DJoin* [10], in order to get an infrastructure language for open systems.

A further benefit of our approach, as we shall see, is that it is suitable for guiding actual distributed implementations of mobile systems, while the most mobile system implementations heavily rely on a single centralized server that manages the distributed system components. Typically, there is a server running on a known machine, that acts as a sort of name server for the other nodes.

The rest of the paper is organized as follows. Section 2 informally summarizes the main features of the language KLAIM, while Section 3 presents our KLAIM-based infrastructure language for open nets. Section 4, by means of a simplified chat system, illustrates how the new linguistic features can be advantageously used. Finally, Section 5 concludes the paper with a few comments.

2. The language Klaim

KLAIM[1] (*Kernel Language for Agent Interaction and Mobility*), is inspired by the Linda coordination model [11, 7], hence it relies on the concept of *tuple space*. A tuple space is a multiset of *tuples*; these are containers of information items (called *fields*). There can be *actual fields* (i.e. expressions, processes, localities, constants, identifiers) and *formal fields* (i.e. variables). Syntactically, a formal field is denoted with *!ide*, where *ide* is an identifier. For instance, the sequence (*"foo"*, *"bar"*, *!Price*) is a tuple with three fields: the first two fields are string values while the third one is a formal field.

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 (*"foo"*, *"bar"*, $100 + 200$) matches with (*"foo"*, *"bar"*, *!Val*). After matching, the variable of a formal field gets the value of the matched field: in the previous example, after matching, *Val* (an integer variable) will contain the integer value 300.

In Linda there is only one global shared tuple space; KLAIM extends Linda by handling multiple distributed tuple spaces. Tuple spaces are placed on *nodes* that are part of a *net*. Each node contains a single tuple space and processes in execution; a node can be accessed through its *address*. There are two kinds of addresses: *Sites* are the identifiers through which nodes can be uniquely identified within a net; *Localities* are symbolic names for nodes. A reserved locality, `self`, can be used by processes to refer to their execution node. Sites have an absolute meaning and can be thought of as IP addresses, while localities have a relative meaning depending on the node where they are interpreted and can be thought of as aliases for network resources. Localities are associated to sites through *allocation environments*, represented as partial functions. Each node has its own environment that, in particular, associates `self` to the site of the node.

KLAIM processes may run concurrently, both at the same node or at different nodes, and can perform five basic operations over nodes. $\mathbf{in}(t)@\ell$ evaluates the tuple t and looks for a matching tuple t' in the tuple space located at ℓ. Whenever the matching tuple t' is found, it is removed from the tuple space. The corresponding values of t' are then assigned to the formal fields of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. $\mathbf{read}(t)@\ell$ differs from $\mathbf{in}(t)@\ell$ only because the tuple t', selected by pattern-matching, is not removed from the tuple space located at ℓ. $\mathbf{out}(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space located at ℓ. $\mathbf{eval}(P)@\ell$ spawns process P for execution at node ℓ. $\mathbf{newloc}(s)$ creates a new node in the net and binds its site to s. The node can be considered a "private" node that can be accessed by the other nodes only if the creator communicates the value of variable s, which is the only way to access the fresh node. Finally, KLAIM processes can be built from the basic operations by using standard operators borrowed from process algebras [14], such as, e.g., *action prefixing* and *parallel composition*.

We now show how to program in KLAIM a *news gatherer*, namely a mobile agent that retrieves information on remote sites. We assume that each node of a database distributed over a KLAIM net contains a tuple of the form (*"item"*,*data*), where the string *"item"* is the search key and *data* is the associated data, or a tuple of the form (*"item"*,ℓ), where ℓ is a locality where more data associated to *"item"* can be searched.

$$
\begin{array}{l}
\mathit{NewsGatherer}(\mathit{searchKey}, \mathit{retLoc}) = \\
\quad \mathbf{in}(\mathit{searchKey}, !\mathit{dataVal})@\mathtt{self}.\mathbf{out}(\mathit{dataVal})@\mathit{retLoc}.\mathbf{nil} \\
\quad | \\
\quad \mathbf{read}(\mathit{searchKey}, !\mathit{nextLoc})@\mathtt{self}. \\
\qquad \mathbf{eval}(\mathit{NewsGatherer}(\mathit{searchKey}, \mathit{retLoc}))@\mathit{nextLoc}.\mathbf{nil}
\end{array}
$$

The agent *NewsGatherer* works as follows: it tries to remove data locally associated to *searchKey* and to forward them to the return location *retLoc*, and, concurrently, tries to send a copy of itself to the locality denoted by *nextLoc* that may contain further data associated to *searchKey*.

KLAIM provides many useful features for programming distributed network-aware systems with mobile code (we refer the interested reader to [9] for an overview of possible applications). However, its underlying model is not quite satisfactory when dealing with open nets. For instance, a process P, running at node s_1, that knows the site s_2, is able to perform actions at s_2. Abstractly, we can think of as nodes s_1 and s_2 are connected. Now, if P migrates to node s_3 then it is still able to perform actions at s_2: it is as if s_3 and s_2 get connected while s_1 and s_2 can possibly get disconnected[2] (if P was the only process in s_1 that knew s_2). In the Internet, however, the knowledge of the address of a remote host may not be sufficient to communicate with it, because there might be no route to the host. Moreover, the model is *flat* in that nodes cannot embody other nodes, thus subnets and hierarchical nets cannot be directly modeled. Finally, while the separation between concrete and symbolic addresses of nodes implements a sort of dynamism, still this dynamism does not suit open networks well. Indeed, the model has a static flavor that leads to a sort of "closed world": apart from the creation of new nodes, the topology of KLAIM networks does not change and all system localities must be known and mapped to sites in advance. The extension we present in the next section will overcome such limitations.

3. A Klaim-based infrastructure language

In this section we will present an infrastructure language for modeling and driving the implementation of large scale mobile

[1]The requirements and the design philosophy of the language are presented in [9]; KLAIM prototype implementation is described in [2].

[2]This mechanism is reminiscent of "link mobility" in the π-calculus [15].

systems whose structure can dynamically evolve in an unpredictable way. Our starting point will be the language presented in Section 3.1, obtained by providing KLAIM with a few mechanisms for dynamically updating nodes' allocation environments and with suitable notations for explicitly expressing node connectivity. Then, our infrastructure language, defined in Section 3.2, will be obtained by integrating the KLAIM dialect of Section 3.1 with a new category of processes, called *NodeCoordinators* that, in addition to the KLAIM operations, can execute coordination operations for establishing new connections, for accepting connection requests and for removing connections.

Any node will now play a double role: it is a computational environment for processes and a *gateway* that nodes can use for connecting to the net by means of explicit login operations. Moreover, nodes can act both as clients (belonging to a specific subnet) and as servers (taking in charge of, possibly private, subnets). Localities represent the names with which client nodes log in server nodes, and allocation environments, that can be dynamically updated with such information, actually represent dynamic tables mapping logical names (possibly not known in advance) into physical addresses (that likely change during the evolution). The client-server relation among nodes smoothly leads to a hierarchical model, also due to the way the resolution of logical names takes place: in order to find the mapping for a locality, allocation environments of nodes in this hierarchy are now inspected from the bottom upwards. This resembles name resolution within DNS servers.

Due to lack of space, in this paper we do not illustrate the operational semantics of our language. The semantics can be found in the full paper [4].

3.1 A Klaim dialect

The formal syntax of KLAIM processes is presented in Table 1. It slightly differs from previous KLAIM presentations in three respects.

- When tuples are evaluated, locality names resolution does not take place automatically anymore. Instead, it has to be explicitly required by putting the operator $*$ in front of the locality that has to be evaluated. For instance, $(3,l)$ and $(s,\mathbf{out}(s_1)@s_2.\mathbf{nil})$ are fully-evaluated while $(*l,\mathbf{out}(l)@\mathtt{self}.\mathbf{nil})$ is not.

- Operation **newloc** cannot be performed by user processes anymore. It is now part of the syntax of NodeCoordinator processes (see Table 3) because, when a new node is created, it is necessary to install one such process at it and, for security reasons, user processes cannot be allowed to do this.

- Operation **bind** has been added in order to enable user processes to enhance local allocation environments with new aliases for sites. For instance, $\mathbf{bind}(l,s)$ enhances the local allocation environment with the new alias l for s.

P	$::=$	$\mathbf{nil}$	(null process)
	$\mid$	$act.P$	(action prefixing)
	$\mid$	$\mathbf{out}(et)$	(evaluated tuple)
	$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	X	(process variable)
	$\mid$	$A(\widetilde{P},\widetilde{\ell},\widetilde{e})$	(process invocation)
act	$::=$	$\mathbf{out}(t)@\ell \mid \mathbf{in}(t)@\ell \mid \mathbf{read}(t)@\ell \mid \mathbf{eval}(P)@\ell$	
	$\mid$	$\mathbf{bind}(l,s)$	
t	$::=$	$f \mid f,t$	
ℓ	$::=$	$l \mid s$	
f	$::=$	$e \mid P \mid \ell \mid *l \mid !x \mid !X \mid !\ell$	

Table 1: Process Syntax

N	$::=$	$\mathbf{0}$	(empty net)
	$\mid$	$s ::_{\rho}^{S} P$	(single node)
	$\mid$	$N_1 \parallel N_2$	(net composition)

Table 2: Net syntax

The formal syntax of KLAIM nets is presented in Table 2. A node is a 4-tuple of the form $(s ::_{\rho}^{S} P)$, where s is the site of the node (i.e. its physical address in the net), ρ is the local allocation environment, P is a set of concurrent processes running at s and S is the set of sites connected to s. In general, P is the parallel composition of many processes, among which processes of the form $\mathbf{out}(et)$, each representing an evaluated tuple of the local tuple space (namely, a tuple space is implemented as the parallel composition of evaluated tuples). A net can be an empty net $\mathbf{0}$, a single node or the parallel composition of two nets N_1 and N_2 with disjoint sets of node sites.

If $s ::_{\rho}^{S} P$ is a node in the net, then we will say that the nodes in S are *logged in* s and that s is a gateway for those nodes. A node can be logged in more than one node, that is it can have more than one gateway. Moreover, if s_1 is logged in s_2 and s_2 is logged in s_3 then s_3 is a gateway for s_1 too. Gateways are essential for communication: two nodes can interact only if there exists a node that acts as gateway for both. Moreover, to evaluate locality names, whenever s_1 is logged in s_2, if a locality cannot be resolved by just using the allocation environment of s_1, then the allocation environment of s_2 (and possibly that of nodes to which s_2 is logged in) is also inspected.

3.2 The infrastructure language

The syntax of NodeCoordinator processes, that are ranged over by $\mathbb{P}$, is given in Table 3. In addition to the standard KLAIM operations, a NodeCoordinator process can also perform four operations: $\mathbf{newloc}(s,\mathbb{P})$, $\mathbf{login}(\ell)$, $\mathbf{logout}(\ell)$ and $\mathbf{accept}(s)$. Notice that all these operations are not indexed with a locality, since they always act locally at the node where they are executed.

$\mathbb{P}$	$::=$	P	(process)
	$\mid$	$sact.\mathbb{P}$	(action prefixing)
	$\mid$	$\mathbb{P}_1 \mid \mathbb{P}_2$	(parallel composition)
	$\mid$	$A(\widetilde{\mathbb{P}},\widetilde{\ell},\widetilde{e})$	(NodeCoordinator invocation)
$sact$	$::=$	act	
	$\mid$	$\mathbf{newloc}(s,\mathbb{P}) \mid \mathbf{login}(\ell) \mid \mathbf{logout}(\ell) \mid \mathbf{accept}(s)$	

Table 3: NodeCoordinator Syntax

NodeCoordinators are special processes that cannot migrate and cannot be used as tuple fields. They are installed at a node either when the node is initially configured or when the node is dynamically created by performing $\mathbf{newloc}(s,\mathbb{P})$. The NodeCoordinator process of a node can be thought of as the coordinator of that node or, more abstractly, as a network operating system that lies at the node; conversely, standard KLAIM processes can be thought of as the user programs that can invoke system calls in the node. In order to integrate NodeCoordinators with the KLAIM dialect of the previous section, we have to extend the syntax of nets so that a NodeCoordinator process can be installed at a node. Thus, the clause for a single node, in the syntax of nets presented in Table 2, is replaced by the following one:

$$s ::_{\rho}^{S} \mathbb{P}$$

Informally, the meaning of the coordination primitives is the following. Operation **newloc**$(s, \mathbb{P})$ creates a new node in the net, binds the site of the new node to s and installs the NodeCoordinator $\mathbb{P}$ at the new node. Notice that a **newloc** does not automatically log the new node in the generating one. This can be done by installing a NodeCoordinator in the new node that performs a **login**. Differently from the standard KLAIM **newloc** operation, the environment is not explicitly inherited by the created node, instead it is subsumed by using the "logged in" relationships among nodes. Operation **login**(ℓ) logs the executing node, say s, in ℓ but only if at ℓ there is a NodeCoordinator process willing to accept a connection, namely a NodeCoordinator process of the form **accept**$(s').\mathbb{P}$. As a consequence of this synchronization, s is added to the set S of nodes logged in ℓ and s' is replaced with s within $\mathbb{P}$. Operation **logout**(ℓ) disconnects the executing node, say s, from ℓ. As a consequence, s is removed from the set S of nodes logged in ℓ and any alias for s is removed from the allocation environment of ℓ.

4. A Chat System

In this Section we show how our KLAIM-based infrastructure language can be fruitfully used for modeling a simplified chat system. The chat system is made of a server that dispatches the messages to all the clients connected to it. The system is dynamic because new clients can enter the chat and existing clients may disconnect. 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 locality. In the following we will use some syntactical shortcuts, such as **if** and **while** instructions and some meta language constructs on lists of data (our language is Turing powerful).

First of all, we introduce two derived operations, **subscribe** and **register**, that permit clients to specify the nicknames with which they want to log in the chat server. Both these operations are endowed with two different continuations, in order to be able to handle also failures. Notice that **register**$(s, l)[\mathbb{P}_1, \mathbb{P}_2]$ acts as a binder for l and s in the continuations $\mathbb{P}_1$ and $\mathbb{P}_2$.

```
subscribe(s,l)[P1,P2] ≜
    login(s).
    out("register", l)@s.
    in(l, !ok)@s.
    if ok then
        P1
    else
        logout(s).P2
    endif
```

```
register(s,l)[P1,P2] ≜
    accept(s).
    in("register", !l)@self.
    if l not already registered then
        out(l, true)@self.
        bind(l, s).P1
    else
        out(l, false)@self.P2
    endif
```

Moreover, we introduce two derived operations, **unsubscribe** and **unregister**, that are useful to keep track of disconnections (indeed, **login** has a complementary operation, while **logout** has not). Both operations will be used as action prefixes and **unregister** binds l in the continuation process.

```
unsubscribe(s,l) ≜
    out("unsubscribe", l)@s.
    logout(s).
    in("unsubscribed", l)@s
```

```
unregister(l) ≜
    in("unsubscribe", !l)@self.
    out("unsubscribed", l)@self
```

Basically, all these derived operations are implemented by means of simple client-server protocols; of course, the protocols could be made more complex in order to handle other situations such as, e.g., unsubscription of unexisting localities.

Now, in order to enter and exit the chat system, a client node will execute the following NodeCoordinator processes:

```
EnterChat() ≝
    in("enter", !server, !nickname)@self.
    subscribe(server, nickname)
    [out("connected", true)@self.
    ExitChat(server, nickname),
    EnterChat()]
ExitChat(server, nickname) ≝
    in("exit", server, nickname)@self.
    in("connected", true)@self.
    unsubscribe(server, nickname).
    EnterChat()
```

In these processes, the first operations are synchronization points with user processes and the tuples involved can be considered as system call exported interfaces.

Additionally, to receive messages from and deliver messages to the chat server, a client node will execute the following standard processes:

```
ReceiveMessages() ≝
    while true do
        in(!msg, !from)@self.
        print the message msg on the screen
    enddo
SendMessages(server, nickname) ≝
    while true do
        read("connected", true)@self.
        input the message msg
        out("message", msg, nickname)@server
    enddo
```

Conversely, to handle clients that enter and exit from the chat system, the server will execute the NodeCoordinator processes *HandleEnter* and *HandleExit*, and to dispatch messages to all the clients logged in, it will execute the standard process *BroadcastMessages*.

```
HandleEnter() ≝
    while true do
        register(s, l)
        [add l to the list of clients, nil]
    enddo
HandleExit() ≝
    while true do
        unregister(l).
        remove l from the list of clients
    enddo
BroadcastMessages() ≝
    while true do
        in("message", !message, !from)@self.
        for every l in the list of clients
            out(message, from)@l
    enddo
```

Like in real chat systems, the chat server could also inform clients of the nicknames of every client logged in, thus clients can also implement "private chat rooms". Moreover, the server could store the nicknames of clients that have disconnected, for enabling them to recover their old settings whenever they reconnect. Of course, these and many other features can be programmed in KLAIM with the presented extensions, but due to lack of space, we omit them.

5. Conclusions and Related Work

We have presented an infrastructure language for open nets. The language has been obtained by integrating the language KLAIM for agent interaction and mobility with coordination primitives for node connectivity. Such new coordination primitives can also be interpreted as mobility operations for nodes, because they allow nodes to enter and exit from (the administrative domains of) other nodes. This kind of mobility is reminiscent of *Ambient* [6] mobility, because a single node migrates as a "whole", i.e. with the entire computational environment. However, differently from Ambient, in our language there is a clean separation between user processes, that can migrate from node to node but cannot change node connectivity, and coordination processes, that can also change net topology but cannot migrate.

The framework presented in this paper generalizes that presented in [5], because now the notion of node and of *cluster* do coincide, therefore nodes can be naturally nested. Moreover, it simplifies that presented in [3], because the allocation environment hierarchy of nested nodes is directly implemented in the semantics without resorting to the use of *routing functions* and of ordering relations over sites as in [3].

Although in this paper we have concentrated on KLAIM, the same coordination primitives can be integrated with other languages/calculi for distributed network-aware mobile applications. However, while such integration should be smooth for programming notations like, e.g., the *Dπ* [12] and *DJoin* [10], that are not endowed with explicit primitives for node mobility, we expect it to be more difficult for notations like, e.g., *Ambient* [6], because of the possible interplay between the new mobility operations and the language primitives' ones. We plan to investigate such integration and other coordination primitive in the near future.

We also want to extend the current implementation of KLAIM with the coordination primitives introduced in this paper. The existing implementation consists of KLAVA, a Java [1] package that supplies the run-time system for KLAIM operations, and of X-KLAIM [2], a programming language that extends KLAIM with a high level syntax for processes: it provides programmers with variable declarations, KLAIM operations, assignments, conditionals, sequential and iterative process composition. A compiler, that translates X-KLAIM programs into Java programs that use KLAVA, is also supplied. X-KLAIM syntax and software can be found on-line at the KLAIM site `http://music.dsi.unifi.it`. In the existing implementation, a KLAIM net is implemented through a server where nodes must register by using their site. The server allows registered nodes to communicate both directly and indirectly (i.e. messages pass through the server). We believe that the primitives introduced in this paper can be smoothly accommodated in the existing implementation. For this, it should suffice to use more than one server in the same net and to allow a node to register in more than one server. Indeed, as we have pointed out in the Introduction, while the most mobile system implementations heavily rely on a single centralized server that acts as a sort of name server for the other nodes, our computation model suggests, and requires, the use of multiple distributed servers.

Acknowledgements. We thank the anonymous referees for their helpful comments.

6. REFERENCES

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 3rd edition, 2000.

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

[3] L. Bettini, M. Loreti, and R. Pugliese. Structured Nets in KLAIM. In J. Carroll, E. Damiani, H. Haddad, and D. Oppenheim, editors, *Proc. of ACM SAC 2000, Special Track on Coordination Models, Languages and Applications*, volume I, pages 174–180. ACM Press, 2000.

[4] L. Bettini, M. Loreti, and R. Pugliese. An Infrastructure Language for Open Nets, 2001. Draft, available at `http://music.dsi.unifi.it/papers.html`.

[5] L. Bettini, M. Loreti, and R. Pugliese. Modelling Node Connectivity in Dynamically Evolving Networks. In *Proc. of CONCOORD, Int. Workshop on Concurrency and Coordination*, volume 54 of *ENTCS*, 2001.

[6] L. Cardelli and A. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, number 1378 in LNCS, pages 140–155. Springer, 1998.

[7] N. Carriero and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4):444–458, 1989.

[8] G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems*, number 1222 in LNCS. LNCS, 1997.

[9] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[10] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. In U. Nestmann and B. C. Pierce, editors, *Proc. of HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *ENTCS*, pages 3–17. Elsevier, 1998. Full version available as CogSci Report 2/98, University of Sussex, Brighton.

[13] F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verification of Multiple - Agent Languages*, number 1192 in LNCS. Springer-Verlag, 1996.

[14] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[15] R. Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Dep. of Comp. Sci., Edinburgh Univ., 1991.

[16] T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997. Also Technical Report 1083, University of Rennes IRISA.

[17] J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.