A Catalog of Architectural Styles for Mobility

Paolo Ciancarini  Cecilia Mascolo

Technical Report UBLCS-98-02
April 1998

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
Recent Titles from the UBLCS Technical Report Series

96-5 The Shape of Shade: a Coordination System, S. Castellani, P. Ciancarini, D. Rossi, March 1996.
96-7 Using Bayesian Belief Networks for the Automated Assessment of Students’ Knowledge of Geometry Problem Solving Procedures, M. Roccetti, P. Salomoni, March 1996 (Revised March 1997).
96-9 Towards an Algebra of Actors, M. Gaspari, April 1996.
96-10 Mobile Petri Nets, A. Asperti, N. Busi, May 1996.
96-12 A Logic Coordination Language Based on the Chemical Metaphor, P. Ciancarini, D. Fogli, M. Gaspari, July 1996.
97-7 Validating a Software Architecture with respect to an Architectural Style, P. Ciancarini, W. Penzo, July 1997.
Abstract
Network-aware applications centered around the Internet and the WWW require special architectural patterns. An important feature in network applications is mobility; however, it is still unclear which entities can be mobile (e.g., data, references, code, agents, operating environments) and especially why and when they should move over the network.

In this paper we describe and compare a series of architectural styles for mobile, network aware applications. We classify the styles according to what entities are moving, and specify their features using MobiS, a specification language based on a “chemical” coordination model.

We explore the styles we introduce showing how they model the software architecture of a “Purchasing System”, a case study in electronic commerce.
1 Introduction

Modern network technologies based on mobile computers and devices, and the programming languages for the Internet like Java require novel software design techniques. An important feature in network applications is mobility; however, it is still unclear which entities can be mobile and especially why and when they should move over the network.

New formal languages are being proposed in order to specify mobility; a short list could include Bauhaus [4], Ambit [3], Join Calculus [11], Klaim [9], and Mobile Unity [15]. These languages can be used to describe and analyze software architectures including mobile entities. Mobility can range from mobility of data, as in client-server architectures, to mobility of code, as in Java based applications, to mobility of agents, as in some applications for virtual shopping, to mobility of whole operating environments, as in platforms including mobile hardware.

The increasing diffusion of systems including mobile components requires the formalization of new architectural patterns based on mobility paradigms [5], using special languages for specifying mobile systems [10].

In this paper we introduce some architectural patterns for mobility. We present such patterns using MobiS, a specification language derived from the PoliS coordination model [7, 6]. We show how MobiS can be used for the specification of mobile architectural styles. We introduce an original classification aiming at precisely defining several different types of mobile entities. The papers [5, 8] define a general taxonomy for systems including mobility; we focus on what is moving, and on how the mobile entity can interact with its environment.

This paper has the following structure. In section 2 we introduce the MobiS language. Section 3 contains the description of the mobile architectural styles and their MobiS formal specification. Section 4 contains the specification of the software architecture of a Purchasing System as instance of the mobility styles just defined. In section 5 we draw some conclusions and illustrate some future work.

2 Overview of MobiS

MobiS is a specification language based on multiple tuple spaces. MobiS specifications are hierarchically structured: a MobiS specification denotes a tree of nested spaces that dynamically evolves in time. It is an enhanced version of PoliS [7].

In MobiS spaces are first class entities, and can move. Formally, a MobiS space contains three types of tuples: ordinary tuples, which are ordered sequences of values, program tuples, which represent agents, and space tuples, which contain subspaces.

A program tuple denotes an agent, which can modify a space removing and adding tuples (and therefore spaces). However, an agent can only handle the tuples of the space it belongs to and the tuples of its parent space. This constraint defines both the “input” and the “output” environment of any agent.

The typical structure of a MobiS specification is graphically shown in Figure 1. In such a figure an ellipse represents a tuple space, a bracketed sequence of values (for example $(5, 6)$) is an ordinary tuple and a pair $(r^*: R)$ is a program tuple; nested ellipses represent nested spaces.

A space is modified by reactions that transform multi-sets of tuples in multi-sets of tuples. Every program tuple $(r^*: R)$ refers to a rule $R$ that defines which reactions can take place. A rule can act either on the tuples of the space in which it resides or on the tuples of the parent space. We call these spaces the rule scope. A rule defines an agent that reads and consumes tuples in its scope, performs a sequential computation, and produces new tuples in its scope.

Formally, a rule consists of a preactivation, a local computation, and a postactivation. The preactivation is a multi-set of tuples to be found in its scope; the local computation is any sequential computation which does not modify the tuple space; the postactivation consists of a multi-set of tuples to be produced in its scope. Notice that this is a very general definition; actually rules need not to be made up of all the components listed above. In fact, a rule can have an empty preactivation, can require no local computation, or can produce no tuples.
The preactivation can include formal tuples, that are tuples whose fields can be identifiers; moreover, it includes the primitive ask, that permits to check the values that are assigned to the identifiers of a formal tuple matched against a tuple in the space.

The semantics of a program tuple PT is that a reaction takes place if a space includes both PT and a multi-set of tuples matching the preactivation of PT. A match predicate checks whether a multi-set of formal tuples $M_M$ can be instantiated by a multi-set $M_{gt}$ of ground tuples. Consequently, such a match relation is defined between pairs of multi-sets of tuples and not between pairs of tuples.

A program tuple is a multi-set rewriting rule: both the preactivation and the postactivation are multi-sets. The local computation is denoted by a label over the arrow between preactivation and postactivation.

When a rule is activated in a space, a reaction takes place: the tuples to be locally consumed are removed, the tuples to be consumed externally are removed from the parent space, the local computation is performed, and finally some tuples are created.

A tuple in the preactivation must be read if the symbol "/?" is put in front of it and must be consumed otherwise; a read or consume operation involves the parent space if the symbol "?" is put in front of a tuple and involves the local space if the symbol is missing; a tuple in the postactivation must be produced in the parent space if the symbol "↑" is put in front of it and must be produced locally otherwise.

Spaces are represented as space tuples ("name" * $SP$) where name is the name of the space. $SP$ is the specification of the contents of the space. A space is a multi-set of tuples.

Both agents and spaces are first class entities in MobiS. In fact, they are themselves part of spaces as program or space tuples, respectively, that can be read, consumed or produced just like ordinary tuples. A program tuple has the form ("rule_id": rule) where rule is a MobiS rule. While a space tuple has the form ("name" * $SP$) where name is the name that identifies the space, and $SP$ is the space configuration. Program and space tuples have an identifier which simplifies reading or consuming program and space tuples ("rule_id" and "name").

Whenever disjoint multi-sets of tuples satisfy the activation preconditions of a set of rules, such rules can be executed independently and simultaneously: every rule modifies only the portion of space containing the tuples that must be read or consumed and therefore other rules can modify other tuples in the space or other spaces.

As they are first class entities, spaces can move. Therefore, whole subtrees (referring to the space tree of the MobiS specification) can migrate from one part of the tree to another.

The mobility in MobiS is based on the consuming and the producing spaces tuples by the rules. As the scope of the rules is the local space and the parent space, the moving is performed "step by step", from a space to its parent and so on.

A simple example helps in explaining both the syntax and the semantics of MobiS. Let us consider a client-server system: the client sends an agent performing some actions to the server...
The server accepts the agent sent from the client. It can always kill the agent if it performs some “illegal actions”.

Such a system can be described by two distinct spaces both included in the main space representing the client and the server. The agent space, that initially is located inside the client space, is moved from the client space to the server space. Such a system is graphically shown in Figure 2.

![Figure 2. Client-Server: spaces topology](image)

Table 1 contains the specification of the system. The StartContext space is the main space, that contains the program tuple (“create” : CREATE). The program tuple indicates that the rule CREATE, specified below in Table 1, is contained in the main space.

The rule CREATE creates two spaces Cl and Sr that contain the tuples describing the client and the server respectively: their specification is described by the Client and Server spaces below in the table.

**Client** is the client space, that contains the name of the client (“name”, k), the program tuple (“send” : SEND) that refers to the rule SEND specified below in the table, and the space tuple (agent * AGENT), that refers to the space AGENT that is specified in the table too. The rule SEND moves the agent space from the client space to the main space († (“agent” * AGENT)) and sets the client state to “wait” emitting the tuple (“wait”).

The agent space is described below in the same table: it contains its name tuple, the code to be executed and three rules. The rule AUTHORIZE emits a tuple containing the name of the agent into the external space in order to let the server recognize it. The rule JOB is enabled when the agent is authorized from the server with the tuple (“go”). It exploits some data from the server space and executes the code computing a result. The function \( f \) on the top of the arrows computes the result: its parameters are specified in the **where** clause, at the end of the rule.

**Server** is the server space. It initially contains its name, some data, and three rules: the rule GET checks if an agent is present in the main space, then moves it in the local space. The rule RECOGNIZE verifies the identity of the agent checking its name (for simplicity we only check that the name is a number smaller than 100), and emits the tuple (“go”).

Spaces can be terminated by external rules (as rules can consume space tuples) as the rule KILL in the server space, that terminates the agent when its name is not an authorized one (a number greater than 100): the primitive ask checks the value; or spaces can terminate themselves by invariant rules. The preactivation of an invariant rule defines a constraint that must hold for all the tuple space lifetime. Whenever an invariant is violated, the tuple space terminates and disappears; its sub-spaces disappear as well. A MobiS invariant is a condition on the tuple space contents: it asserts that the space will never contain a given multi-set of tuples. Invariant rules can only read tuples locally (the tuples that must not belong to the tuple space) and produce tuples in the parent space. When the tuples to be read are in the space, the reaction specified by the invariant takes place in the usual way. Local computation and tuple production are used to communicate possible results to the parent space and then the space dies. Invariants are given by means of special program tuples whose names are replaced by the keyword invariant. MobiS rules cannot consume the space in which they are. For this purpose the specifier should use the Invariant rules. An example is the invariant rule END in the AGENT space that makes the agent die when the data tuple (“data”, “1000”) is in the extern space. It emits a tuple (“agent_expired”) to signal the dead of the space.
Tuples representing messages are put in a space shared by components which have to communicate. Hence, communication is decoupled because components do not know each other, since they access tuples by pattern matching. Since messages have no destination address, their contents determine the set of possible receivers, (communication is property driven).

In summary, a space represents at the same time both a component performing a (chemical) computation and a persistent, multicast channel supporting communication among the components it contains.

That the coordination model characteristics can easily encode different mobility paradigms: in the next section we introduce these paradigms.

3 Specification of Mobile Architectural Styles

An architectural style is an abstract skeleton which helps in designing, understanding, and analyzing actual software architectures, said instances of such a style.

There are at least three reasons why it is important and useful to systematically study architectural styles:

1. to help designers to choose a specific style in a given design situation; the definition and classification of common architectural styles with clearly defined properties supports both
design and code reuse;
2. to build a library of styles, so that software designers can choose the most appropriate one;
3. to support analysis methods and tools suitable to deal with style instances, namely concrete
   software architectures, understanding and reasoning on their properties.

We have defined a basic set of architectural styles for mobility. We catalog these mobile architectural
styles in terms of what is moving, namely which entities move with respect to an infrastructure
including at least two immobile entities: a requester entity and a supplier entity: the requester
asks the supplier for a service. Both these immobile entities can be thought of as two Internet
sites connected by some channel able to transport mobile entities from a site to another.

In figure 3 we have illustrated the different styles of mobility described below. The non-
dashed arrows represent sending of tuples (or spaces) between the two components (i.e. the
requester and the supplier). The dashed arrows show the reference links among entities. The
non-dashed circles represent the spaces, while the dashed ones represent the groups of tuples that
move. For instance, in the style 6 code and store move together and are graphically represented
in figure 3 in a dashed circle. The request to the supplier and the related answers of the code and
store tuples are represented by the non-dashed arrows.

1. **Data**: This is the simpler kind of mobility to understand. The mobile entities are data from
   the supplier to the requester. A typical example is a client-server architecture based on
   a protocol like HTTP: HTTP servers send to HTTP clients data in form of HTML pages
   (HTML being a non Turing equivalent language).
2. **Reference**: in this case what moves from the supplier to the requester is only a reference to
   some entity, not some data or program. An example of reference mobility style is an URL
   address: the requester receives an URL and can “move” its browser to the referred page.
3. **Code**: in this case the executable code can move from a site to an other. Java applets are
   based on code mobility.
4. **Code&Store**: what moves is both the code and the values of the variables it refers to (store).
This kind of mobility is usually termed agent mobility and is the form of mobility offered by most Java mobile agent systems.

5. **Code, Store & State:** Code, store, and scheduling state of an agent/thread can move. Referential transparency is the main feature of this kind of mobility, as the execution continues on a different site from the exact point it had been suspended on the other site. This style of mobility is implemented in Telescript.

6. **Closure:** An agent moves and maintains its links to resources which were local to the agent source location and become remote in the destination. As an example we quote Obliq [2], that uses exactly this kind of mobility.

7. **Ambient:** This style describes the moving of the whole ambient involved in a computation. Ambients can contain other ambients that are moved too. In this way it is possible to model, for instance, the moving of a set of programs from a workstation to a laptop. At the moment no languages exist allowing this kind of mobility. However, Cardelli and Gordon have proposed a specification language based on this paradigm [3].

<table>
<thead>
<tr>
<th>Table 2. MobiS specification of the Reference paradigm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Startcontext</strong></td>
</tr>
<tr>
<td>$\text{Startcontext} = \langle \text{&quot;R&quot; * Requester, &quot;S&quot; * Supplier} \rangle$</td>
</tr>
<tr>
<td><strong>Requester</strong></td>
</tr>
<tr>
<td>$\text{Requester} = \langle \text{&quot;name&quot;, k, &quot;refreq&quot;, req, &quot;request&quot; : REFREQ, &quot;get&quot; : GET, &quot;exe&quot; : EXE, &quot;data&quot;, d, &quot;state&quot;, s} \rangle$</td>
</tr>
<tr>
<td><strong>REFREQ</strong></td>
</tr>
<tr>
<td>$\text{REFREQ} = \langle \text{&quot;refreq&quot;, req} \rangle$</td>
</tr>
<tr>
<td><strong>GET</strong></td>
</tr>
<tr>
<td>$\text{GET} = \langle \text{&quot;refdata&quot;, rd, &quot;refreq&quot;, req} \rangle$</td>
</tr>
<tr>
<td><strong>EXE</strong></td>
</tr>
<tr>
<td>$\text{EXE} = \langle \text{&quot;code&quot;, c, &quot;refdata&quot;, rd} \rangle_{\text{where } f(c, data, ref, state) = (nstate(c, data, state))}$</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
</tr>
<tr>
<td>$\text{Supplier} = \langle \text{&quot;name&quot;, k, &quot;data&quot;, d, &quot;put&quot; : PUT, &quot;getreq&quot; : GETREQ} \rangle$</td>
</tr>
<tr>
<td><strong>GETREQ</strong></td>
</tr>
<tr>
<td>$\text{GETREQ} = \langle \text{&quot;refreq&quot;, req, &quot;data&quot;, d} \rangle$</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
</tr>
<tr>
<td>$\text{PUT} = \langle \text{&quot;refreq&quot;, req, &quot;data&quot;, d} \rangle_{\text{where } f(x) = (reference to(x))}$</td>
</tr>
</tbody>
</table>

The MobiS language allows the specification of architectural styles for mobility. Architectural styles are abstractions including components, connectors [16]. Components are computation loci while connectors define the interactions among components. In MobiS components can be specified as spaces (that can also be nested): new components can be generated (i.e. spaces can be created), eliminated (i.e. spaces can be consumed), or can migrate (i.e. spaces can be consumed and recreated elsewhere).

The concept of connector in MobiS is much more implicit (as in [13, 12]): the components interact thanks to the coordination model of the language and communication is specified using the asynchronous multiset rewriting mechanism.

The way in which MobiS models software architectures is similar to the one described in [12] where the CHAM coordination model is used. The coordination allows flexible moving of components and extensibility of the model. Our model, where rules and spaces as first class
entities, provides a framework in which encoding all the different styles listed above.

We now specify some mobility styles using MobiS.

Table 2 contains the specification of the Reference style. The main space contains two spaces, a “Requester” and a “Supplier”. The Requester space contains a data reference request tuple, the code, and the state. The rule \texttt{REF REQ} sends a reference request to the main space. The rule \texttt{GET} gets from the main space the data reference. The rule \texttt{EXE} formalizes the execution of the request code using the reference to the data. Notice that as the spaces are organized in a tree it is quite easy to specify the access to the referenced data: spaces have names in form of paths.

In table 5 where the closure style is specified, the spaces maintain some knowledge of the remote location of the resources exploiting the tree structure of the spaces as in the Join Calculus [11].

A Supplier space contains the data and two rules. The rule \texttt{GET REQ} accepts requests of data references while the rule \texttt{PUT} emits a reference tuple the main space in order to let the requester to catch the required reference to data.

We omit the formalization of the Data style, in fact it is quite similar to the Reference style: the only difference being that the supplier has to send data to the requester and not simply references to data.

Table 3 contains the formalization of the Code style: The Requester space contains a code-reference tuple, the state, and the store. The rule \texttt{CODE REQ} sends a code request in the main space. The rule \texttt{GET} gets from the main space the serialized code sent by the supplier. The rule \texttt{EXE} formalizes the execution of code updating the values of the state and the store. The Supplier space contains the code and two rules. The rule \texttt{GET REQ} accepts a request of code from the main space, and the rule \texttt{PUT} emits the serialized code in the main space.

\begin{verbatim}
Startcontext = (("R" * Requester), ("S" * Supplier))

Requester = (("name", k), ("codereq", req), ("request": CODE REQ), ("get": GET))

CODE REQ = \{ \{ \{"codereq", req\} \} \} \rightarrow \{\{"codereq", req\}\}

GET = \{ \{"serializedcode", sc\}, ("codereq", req) \} \xrightarrow{\{sc\} \cdot f(st)} \{\{"code", c\}\}

where f(x) = (f, code(x))

EXE = \{ \{"code", c\}, \{"store", st\}, \{"state", s\} \} \xrightarrow{(f', st'} \{\{"store", st'\}\, \{"state", s'\}\}

where f(st, code, store, state) = (n state(st, code, store, state), n store(st, code, store, state))

Supplier = (("name", k), ("code", c), ("put": PUT), ("getreq": GET REQ))

GET REQ = \{ \{"codereq", req\}, \{"code", c\} \} \rightarrow \{\{"codereq", req\}\}

PUT = \{ \{"codereq", req\}, \{"code", c\} \} \xrightarrow{\{"serializedcode", sc\} \cdot f(st)} \{\{"serializedcode", sc\}\}

where f(x) = (serialize(x))
\end{verbatim}

Table 3. MobiS specification of Code Paradigm

Table 4 shows the formalization of the Code & Store”style: the Requester space contains a code&store request tuple, and the state. The rule \texttt{REQ} is like in Table 3. The rule \texttt{GET} gets the
serialized code and the store from the main space. The EXE rule executes the code and updates the state and the store.

The Supplier space contains the code and the store, it gets the request from the main space by the rule GET REQ, and puts there the serialized code and the store.

\[
\begin{align*}
\text{Startcontext} &= \{ \langle \text{R} \ast \text{Requester} \rangle, \langle \text{S} \ast \text{Supplier} \rangle \} \\
\text{Requester} &= \{ \langle \text{name}, k \rangle, \langle \text{code} \land \text{store}, \text{req} \rangle, \langle \text{request} : \text{REQ} \rangle, \langle \text{get} : \text{GET} \rangle \} \\
\text{Supplier} &= \{ \langle \text{name}, k \rangle, \langle \text{code}, c \rangle, \langle \text{store}, st \rangle, \langle \text{put} : \text{PUT} \rangle, \langle \text{getreq} : \text{GETREQ} \rangle \}
\end{align*}
\]

\[
\begin{align*}
\text{REQ} &= \{ \langle \text{code} \land \text{store}, \text{req} \rangle \} \\
\text{GET} &= \{ \langle \text{serialized code} \land \text{store}, \text{sc} \rangle, \langle \text{code} \land \text{store req}, \text{req} \rangle \} \\
\text{EXE} &= \{ \langle \text{code}, c \rangle, \langle \text{store}, st \rangle \} \\
\text{GETREQ} &= \{ \langle \text{code} \land \text{store req}, \text{req} \rangle, \langle \text{code}, c \rangle, \langle \text{store}, st \rangle \} \\
\text{PUT} &= \{ \langle \text{code} \land \text{store}, \text{req} \rangle, \langle \text{code}, c \rangle, \langle \text{store}, st \rangle \} \\
\end{align*}
\]

where \( f(x) = (\text{code}(x), \text{state}(x)) \)

**Table 4. MobiS specification of Code&Store paradigm**

We omit the formalization of **Code, Store, & State style** that is very similar to the previous one: the Requester space does not contain anymore the state. It has to require code, state and store to the supplier by the rule REQ. The rule GET helps in acquiring the tuples. The EXE rule executes the code. The Supplier space contains the state, the store, and the code: it accepts requests from the main space and sends the store.

The following two styles fully exploit the power of MobiS as they are based on the mobility of spaces.

Table 5 contains the formalization of the **Closure style**: a closure is a group of entities, that we model as a space. The Supplier space contains the tuple with its name (location), a tuple indicating the subspace (\( \langle k.CI \ast \text{Closure} \rangle \)), and two rules (GETREQ and PUT). The Closure space is specified below in the table and \( k.CI \) is the name of the subspace, indicating the location of the subspace in the space tree. The rule PUT puts the closure space in the main space, when ready. It also updates the location of the subspace, eliminating the name of the supplier space from the location path of the name of the closure space (\( \text{diff} \) function in the where clause has this role).

The Closure space contains its name, the code, the state, and the store. The TRANSF rule is activated only once (it consumes itself). It creates a reference to the resources of the supplier space and emits a tuple (\( \text{"ready"} \)) in the supplier space, indicating that the closure is ready to be moved elsewhere. The EXE rule executes the code exploiting the resources of the supplier.
space, no matter if the closure space is not located in the supplier site anymore.

The Requester space contains the rule GET that gets the closure space from the parent space and updates its location adding the requestor name in the name path of the closure space (function concat in the where clause does it).

Table 6 contains the specification of the Ambient style: the Supplier space contains two rules and an Ambient subspace. The rule PUT transfers the ambient space outside. It changes the location of the ambient as in the Closure style. The Ambient space contains the code, the state, the store. The rule EXE executes the code using the local resources of the site, no matter where the ambient is located.

The Requester rule GET gets the Ambient space from the parent space and updates its location name.

4 Application of the styles to an architecture of a mobile system

Now we consider the software architecture of a system and apply these styles to see how these paradigms can be used.
As a case study we consider an electronic commerce application. With the advancements in the network technology new kinds of applications are now possible. A purchaser is trying to buy items at the best available prices on the network. The purchaser travels on the network looking for the best selling-price. We have simplified the problem supposing that the purchaser is looking for the best price of a single object.

We exploit the mobility styles defined in section 3 to specify the software architecture of the Purchasing System.

Using the Data style the purchaser can be seen as a requester that asks for the items prices from different stores. The stores send prices of the items to the purchaser that can remotely check the prices and choose the lower one. The use of the Reference style is very similar: the stores do not send directly the prices of the items, but they send references to their catalogs. The purchaser still does not move, and it can remotely check the prices on the catalogs.

In the Code style solution we imagine the purchaser migrating from a store site to another moving its code. Every store puts an advertisement request tuple, ("newsellingprice", reselling), in the main space. The store containing the code of the purchasing agent emits the code tuple in the main space and the store that puts the advertisement can obtain the purchasing code. However this solution is not suitable for the purchasing system, because the purchaser has to remember the best price found every time it moves. With the Code&Store style every node has the function of both requester and supplier. The purchaser code and a tuple for storing the best price found till that moment, ("bestprice", price), are sent from a store to another whenever a new advertisement tuple is received. This solution fits better the

Table 6. MobiS specification of the Ambient style
purchasing system than the one with the Code style, in fact it allows the store (the best price found) to be moved with the code, letting the purchaser do its job. Figure 4 shows the architecture.

The new EXE rule (that has to instance the one shown in Table 4) updates the store of the purchaser, with a new best price, if the price offered by the local space is better than the one in the previous store.

Code, Store, & State style is not suitable, in fact the same procedure has to be applied on every node in order to find the best price. We are not interested in transparently continuing the execution of the purchaser on different sites: the state can be moved, of course, but the case is not interesting.

The Closure style allows the references to remote resources. It would be useful, for instance, if the purchaser, while moving, has to communicate its temporary best price found to its company: it could for example print the offered price of every store on the company (remote) printer.

Using the Ambient style we imagine an “agent” traveling with all its data and exploiting local resources (printer, modem, …) on different selling-stores looking for the best price for an item. The purchaser could, for example, use the local printer to print the temporary best price found till that moment.

These architectures offer different advantages and some of them are better than the other for particular requirements. The designer knows the requirements of the systems that he/she wants to implement and can choose on the basis of these requirements the style of mobility that better suits its model.

5 Conclusions

In this paper we have formalized some architectural styles for mobility. We have used MobiS for the formal specification. The MobiS model fits very well the specification of mobile aspects of the systems: in [10] it is argued that mobile agents can easily be specified using this coordination model. We have formalized a system with mobile agent using PoliS (a previous version of MobiS without mobility of spaces) in [6]. In this paper an architecture of a simplified purchasing system has been described exploiting the different architectural styles formalized.

In [5] a different classification of code mobility is given, focusing on the way the moving is performed. In [15] a formalization of these paradigms using Mobile Unity is given: however, little attention is given to the architectural point of view and to the idea of instancing styles to specify software architectures.

Obliq [2] and the Ambient calculus [3] are specification languages for mobility: they both offer a particular approach to mobility based on a particular style (respectively Closure and Ambient).

Several research efforts have been devoted to address mobility using the π-calculus and its extensions, that are process algebras where processes communicate using mobile channel references. The distributed Join-calculus [11] is an extension of the π-calculus which introduces the explicit notions of named localities and distribution failure.
Klaim [9] and Bauhaus [4] are specification languages based on multiple tuple spaces, like MobiS. Bauhaus spaces can be nested as in MobiS while in Klaim spaces are not nested. Klaim includes a process algebra and uses a type system to perform some security checks. Differently from MobiS, Klaim and Bauhaus do not provide mobility of spaces.

In [14] the author makes an effort of combining software architecture and coordination languages. We have presented an approach in this direction exploiting a coordination language (MobiS) for the specification of architectural styles for mobility.

The coordination model of MobiS is quite similar to the semantics framework of the CHAM [1]. CHAM has been exploited to specify software architectures [12]. However the CHAM does not provide mobility of solutions (that are the CHAM spaces).

The MobiS model of mobility is quite realistic: “it is not realistic to imagine that an agent can migrate from any point A to any point B on the Internet … Access information is controlled at many levels” [3]: MobiS movement happens only between spaces which are parent and son. A complete travel is composed of many of these single steps movements.

Security is a key issue in research on mobile computation; we are trying to develop some security features on the MobiS model, exploiting this one-step moving mechanism.

We have developed a model checker for PoliS [6]. We are enhancing the model checker in order to make it work also with space mobility.

References

