MobiS: A Specification Language for Mobile Systems

Cecilia Mascolo

Department of Computer Science, University of Bologna,
Mura Anteo Zamboni ,7, I-40127 Bologna (Italy)
phone: +39 51 354871, fax: +39 51 354510
e-mail: {mascolo}@cs.unibo.it

Abstract. New formal languages have been proposed for the specification of mobility aspects of systems and for understanding the recent devised technologies for mobile computing.
In this paper we introduce MobiS, a specification language based on a tuple-spaces based model which specifies coordination by multiset rewriting. We show how MobiS can be flexibly used to specify architectures containing mobile components and give formalization of some common mobility paradigms.
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 programming languages for the Internet like Java require novel software design techniques. An important feature in network-aware applications is mobility. Since there are several kind of entities that can be mobile, like for instance data, code, agents, operating environments, an issue designers have to face is the description of the entities that can move and the justification of why and when they should move. New formal languages are being proposed in order to specify mobile entities; a short list includes Bauhaus [3], Ambit [2], Join Calculus [8], Klaim [6], and Mobile Unity [13]. 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 applications including Java applets, 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 suggests the formalization of new architectural styles based on mobility paradigms [9], using special languages for specifying mobile systems [7].

In this paper we study some architectural styles for mobility. We present such styles using MobiS, a specification language based on a coordination model with hierarchical tuple-spaces and multiset rewriting. We show how MobiS can be used for specifying and reasoning on different mobile entities, clarifying their properties and features.

We introduce an original classification aiming at precisely defining several different types of mobile entities. The paper [9] defines a general taxonomy for systems including mobility; the paper shows a classification of mobility in three dimensions: technologies, paradigms, and applications. We focus on paradigms and present a more fine grained taxonomy merging the handling of code and execution state mobility to the handling of the data (this is the terminology in [9]).

This paper has the following structure. In Section 2 we introduce the MobiS language. Section 3 contains the description of the mobile paradigms and their MobiS formal specification. Section 4 contains the specification of the software architecture of a Purchasing System as an instance of some mobility styles we define. In Section 5 we list some related work, draw conclusions and illustrate our current work.

2 Overview of MobiS

MobiS is an extended version of PoliS, a specification language based on multiple tuple spaces [5]. MobiS specifications are hierarchically structured: a MobiS specification denotes a tree of nested spaces that dynamically evolves in time. Spaces are first class entities, and can move, namely they can change their position in the tree.

The typical structure of a MobiS specification is graphically shown in Figure 1 (First Step). In such a figure the structure of the tree represents the hierarchical structure of the nested tuple-spaces. The environment contains two "Hosts" ($Host1$ and $Host2$). "Host1" contains an "Agent". Formally, a MobiS space can contain three types of tuples: *ordinary tuples*, which are ordered sequences of values, *program tuples*, which represent code, and *space tuples*, which represent spaces.

A program tuple can modify a space removing and adding tuples (and therefore spaces). However, a program tuple 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 tuple. The action of the program tuple is performed atomically. A program tuple ("r" : R) refers to a rule R that defines which reaction 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 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. 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. 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 "$\uparrow$" 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 " $\uparrow$" 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. Spaces are first class entities in MobiS. In fact, they are themselves part of spaces (as space tuples) as program tuples, are and they can be read, consumed or produced just like ordinary tuples. A program tuple has the form (*"rule_id"*: *rule*) where *rule_id* is a rule identifier and *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. 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).

Mobility. As they are first class entities, spaces can move. Actually, a whole subtree (referring to the space tree of the MobiS specification) can migrate from a node to another. Mobility in MobiS consists of consuming and producing space tuples. 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.

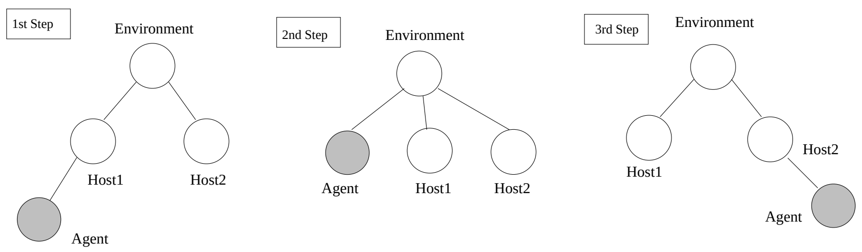

Fig. 1. An Agent Moves

Environment

$$Environment = \{\!\!|\, (\text{“}create\text{”} : CREATE) \,|\!\!\}$$

$$CREATE = \{\!\!|\, (\text{“}create\text{”} : CREATE) \,|\!\!\} \longrightarrow \{\!\!|\, (\text{“}H1\text{”} * Host1), (\text{“}H2\text{”} * Host2) \,|\!\!\}$$

Host1

$$Host1 = \{\!\!|\, (\text{“}name\text{”}, k), (\text{“}send\text{”} : SEND), (\text{“}agent\text{”} * AGENT) \,|\!\!\}$$

$$SEND = \{\!\!|\, (\text{“}agent\text{”} * AGENT) \,|\!\!\} \longrightarrow \{\!\!|\, \uparrow(\text{“}agent\text{”} * AGENT), (\text{“}wait\text{”}) \,|\!\!\}$$

AGENT

$$AGENT = \left\{\!\!\left| \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}code\text{”} : CODE), \\ (\text{“}authorize\text{”} : AUTHORIZE), (\textbf{invariant} : END) \end{array} \right|\!\!\right\}$$

$$AUTHORIZE = \{\!\!|\, ?(\text{“}name\text{”}, j) \,|\!\!\} \longrightarrow \{\!\!|\, \uparrow(\text{“}name\_agent\text{”}, j) \,|\!\!\}$$

$$CODE = \{\!\!|\, \uparrow(\text{“}go\text{”}), \uparrow(\text{“}data\text{”}, d) \,|\!\!\} \xrightarrow{(r) \leftarrow f(d)} \{\!\!|\, (\text{“}result\text{”}, r) \,|\!\!\}$$

where $f(dat) = (res)$

$$END = \{\!\!|\, \uparrow(\text{“}data\text{”}, \text{”}1000\text{”}) \,|\!\!\} \longrightarrow \{\!\!|\, \uparrow(\text{“}agent\_expired\text{”}) \,|\!\!\}$$

Host2

$$Host2 = \left\{\!\!\left| \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}getag\text{”} : GET), (\text{“}kill\text{”} : KILL), \\ (\text{“}recognize\text{”} : RECOGNIZE), (\text{“}data\text{”}, \text{”}d\text{”}) \end{array} \right|\!\!\right\}$$

$$GET = \{\!\!|\, \uparrow(a * AGENT) \,|\!\!\} \longrightarrow \{\!\!|\, (a * AGENT) \,|\!\!\}$$

$$RECOGNIZE = \{\!\!|\, (\text{“}name\_agent\text{”}, j), \textbf{ask}(j \leq 100) \,|\!\!\} \longrightarrow \{\!\!|\, (\text{“}go\text{”}) \,|\!\!\}$$

$$KILL = \{\!\!|\, (\text{“}name\_agent\text{”}, j), \textbf{ask}(j > 100), (a * AGENT) \,|\!\!\} \longrightarrow \{\}$$

Table 1. Specification of a System including a Mobile Agent

A simple example helps in explaining both MobiS syntax and semantics. Let us consider a system with two hosts: Host1 sends an agent performing some actions to Host2 site. Host2 accepts the agent sent from Host1. Host2 can kill the agent if it performs some “illegal actions”. The two hosts can be described by two distinct spaces both included in a main space representing an “environment”. The agent space, that initially is located inside the Host1 space, moves from the Host1 space to the Host2 space. Such a movement is graphically shown in Figure 1. Table 1 contains the specification of the system. The *Environment* 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 $H1$ and $H2$ containing the tuples describing Host1 and Host2 respectively: their specification is included in the table. $Host1$ contains the host name ("*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 a space AGENT that is specified in the table as well. The rule $SEND$ moves the agent space from the host space to the main space ($\uparrow$("*agent*" $* AGENT$)) and sets the host state to "wait" emitting the tuple ("*wait*"). The *agent* space is described below in the same table: it contains a 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 Host2 recognize it. The rule $CODE$ is enabled when the agent is authorized by the Host2 with the tuple ("*go*"). It exploits some data from the Host2 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. $Host2$ 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 (e.g.. we check that the name is a number smaller than 100), and emits the tuple ("*go*"), granting the agent the permission to move.

Spaces can be terminated by external rules (as rules can consume space tuples) as the rule $KILL$ in the Host2 space, that terminates the agent when its name is not an authorized one (e.g.. 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 kills the agent when the data tuple ("*data*", "1000") is in the external space. It emits a tuple ("*agent_expired*") to signal the termination of the space.

In summary, a space represents at the same time both a component performing a (reactive) computation and a persistent, multicast channel supporting communication among the components it contains. The coordination model underlying MobiS can easily encode different mobile entities: in the next section we introduce a fine grained classification of mobile entities.

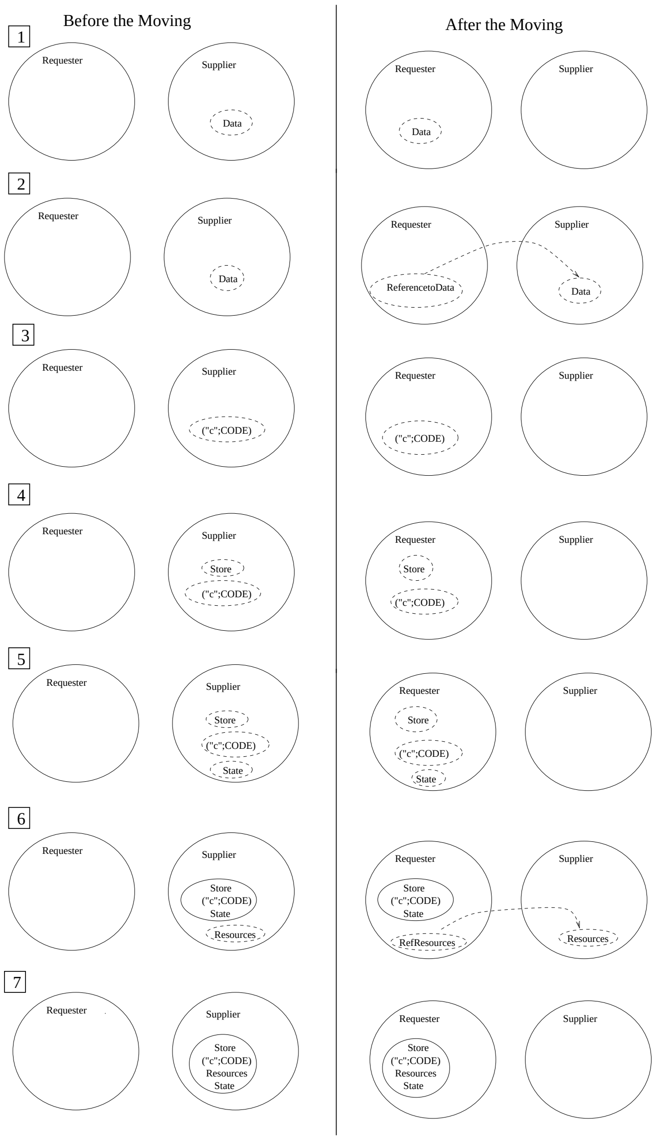

Fig. 2. A Classification of Mobility

3 Specification of Architectural Styles for Mobility

Environment

$$Environment = \{\!\!\{ (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier) \}\!\!\}$$

Requester

$$Requester = \left\{\!\!\left\{ \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}refreq\text{”}, req), (\text{“}request\text{”} : REFREQ), \\ (\text{“}get\text{”} : GET)(\text{“}code\text{”} : CODE), (\text{“}data\text{”}, d), (\text{“}state\text{”}, s) \end{array} \right\}\!\!\right\}$$

$$REFREQ = \{\!\!\{ ?(\text{“}refreq\text{”}, req) \}\!\!\} \longrightarrow \{\!\!\{ \uparrow(\text{“}refreq\text{”}, req) \}\!\!\}$$

$$GET = \{\!\!\{ \uparrow(\text{“}refdata\text{”}, rd), (\text{“}refreq\text{”}, req) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}refdata\text{”}, rd) \}\!\!\}$$

$$CODE = \{\!\!\{ ?(\text{“}refdata\text{”}, rd), ?(\text{“}state\text{”}, s), \}\!\!\} \xrightarrow{(s') \leftarrow f(s, rd)} \{\!\!\{ (\text{“}state\text{”}, s') \}\!\!\}$$

where $f(dataref, state) = (nstate(data, state))$

Supplier

$$Supplier = \{\!\!\{ (\text{“}name\text{”}, k), (\text{“}data\text{”}, d), (\text{“}put\text{”} : PUT), (\text{“}getreq\text{”} : GETREQ) \}\!\!\}$$

$$GETREQ = \{\!\!\{ \uparrow(\text{“}refreq\text{”}, req), ?(\text{“}data\text{”}, d) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}refreq\text{”}, req) \}\!\!\}$$

$$PUT = \{\!\!\{ (\text{“}refreq\text{”}, req), ?(\text{“}data\text{”}, d) \}\!\!\} \xrightarrow{(rd) \leftarrow f(d)} \{\!\!\{ \uparrow(\text{“}refdata\text{”}, rd) \}\!\!\}$$

where $f(x) = (referenceto(x))$

Table 2. MobiS specification of the Reference paradigm

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:

- 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;
- to build a library of styles, so that software designers can choose the most appropriate one;
- 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, and the supplier provides the service. These two entities are actually part of the styles as they characterize the structure of the environment. 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 2 we have illustrated the different paradigms for mobility described below. The dashed arrows show the reference links among entities. The non-dashed circles represent the spaces, while the dashed ones represent the tuples. The figure shows, for each style, the location of the tuples/spaces before and after the moving.

We now describe a number of architectural styles including different mobile components and give some concrete examples of their use.

1. **Data Style**: This is the simplest kind of mobility to understand. The mobile entities are data from a supplier to a 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 not Turing complete language).
2. **Reference Style**: in this case what moves from the supplier to the requester is only a reference to some entity, not data or programs. An example of a mobile reference is an URL address: a requester receives an URL and can "move" its browser to the referred page.
3. **Code Style**: in this case some executable code can move from a site to an other. Java applets are based on code mobility.
4. **Code&Store Style**: in this case both some code and the values of the variables it refers to (a *store*) move. This kind of mobility is usually termed *agent mobility* and it is the form of mobility offered by current Java mobile agent systems.
5. **Code, Store&State Style**: in this case code, store, and scheduling state (threads) of an agent can move. Referential transparency is the main goal 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 has been implemented in Telescript.
6. **Closure Style**: In this case an agent (code+store) moves and maintains its links to resources which were local to its source location and become remote in its destination. As an example we quote Obliq [1], that uses exactly this kind of mobility.
7. **Ambient Style**: 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 programming language based on this paradigm [2].

MobiS allows the specification of architectural styles for mobility. Architectural styles are abstractions including components, connectors [14]. Components

Environment

$$Environment = \{\!\!|\ (\text{``}R\text{''} * Requester), (\text{``}S\text{''} * Supplier)\ |\!\!\}$$

Requester

$$Requester = \left\{\!\!\left| \begin{array}{l} (\text{``}name\text{''}, k), (\text{``}codereq\text{''}, req), (\text{``}request\text{''} : CODEREQ), \\ (\text{``}get\text{''} : GET), (\text{``}store\text{''}, st), (\text{``}state\text{''}, s) \end{array} \right|\!\!\right\}$$

$$CODEREQ = \{\!\!|\ ?(\text{``}codereq\text{''}, req)\ |\!\!\} \longrightarrow \{\!\!|\ \uparrow(\text{``}codereq\text{''}, req)\ |\!\!\}$$

$$GET = \{\!\!|\ \uparrow(\text{``}serializedcode\text{''}, sc), (\text{``}codereq\text{''}, req)\ |\!\!\} \xrightarrow{(c) \leftarrow f(sc)} \{\!\!|\ (\text{``}code\text{''}, c)\ |\!\!\}$$

where $f(x) = (f_{code}(x))$

Supplier

$$Supplier = \{\!\!|\ (\text{``}name\text{''}, k), (\text{``}c\text{''} : CODE), (\text{``}put\text{''} : PUT), (\text{``}getreq\text{''} : GETREQ)\ |\!\!\}$$

$$GETREQ = \{\!\!|\ \uparrow(\text{``}codereq\text{''}, req), ?(\text{``}code\text{''}, c)\ |\!\!\} \longrightarrow \{\!\!|\ (\text{``}codereq\text{''}, req)\ |\!\!\}$$

$$PUT = \{\!\!|\ (\text{``}codereq\text{''}, req), ?(\text{``}code\text{''}, c)\ |\!\!\} \xrightarrow{(sc) \leftarrow f(c)} \{\!\!|\ \uparrow(\text{``}serializedcode\text{''}, sc)\ |\!\!\}$$

where $f(x) = (serialize(x))$

$$CODE = \left\{\!\!\left| \begin{array}{l} ?(\text{``}store\text{''}, st), \\ ?(\text{``}state\text{''}, s), \end{array} \right|\!\!\right\} \xrightarrow{(s', st) \leftarrow f(s, st)} \left\{\!\!\left| \begin{array}{l} (\text{``}state\text{''}, s'), \\ (\text{``}store\text{''}, st') \end{array} \right|\!\!\right\}$$

where $f(store, state) = (nstate(store, state))$

Table 3. MobiS specification of Code Paradigm

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 way in which MobiS models software architectures [12] is similar to the one described in [10] 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. The concept of connector in MobiS is in some sense implicit (as in [11, 10]): components interactions are defined by the coordination model, whereas communication is specified using the asynchronous mechanism of multiset rewriting.

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 *REFREQ* sends a reference request to the main space. The rule *GET* gets from the main space the data reference. The rule *CODE* formalizes the execution of the requester 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 4 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 [8].

Environment

$$Environment = \{\!|\, (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier) \,|\!\}$$

Requester

$$Requester = \left\{\!\!\left|\begin{array}{l} (\text{“}name\text{”}, k), (\text{“}req\text{”}, req), (\text{“}request\text{”} : REQ), (\text{“}get\text{”} : GET), \\ (\text{“}name\text{”}, \text{“}Requester\text{”}) \end{array}\right|\!\!\right\}$$

$$REQ = \{\!|\, ?(\text{“}req\text{”}, req) \,|\!\} \longrightarrow \{\!|\, \uparrow(\text{“}req\text{”}, req) \,|\!\}$$

$$GET = \left\{\!\!\left|\begin{array}{l} \uparrow(j * Closure), (\text{“}name\text{”}, u) \\ (\text{“}req\text{”}, req) \end{array}\right|\!\!\right\} \xrightarrow{(k) \leftarrow f(j,u)} \{\!|\, (k * Closure) \,|\!\}$$

where $f(x, y) = (concat(x, y))$

Supplier

$$Supplier = \{\!|\, (\text{“}name\text{”}, k), (k.Cl * Closure), (\text{“}resource\text{”}, r), (\text{“}put\text{”} : PUT) \,|\!\}$$

$$PUT = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{“}req\text{”}, req), ?(\text{“}name\text{”}, i), \\ (k * Closure), \\ (\text{“}ready\text{”}), ?(\text{“}resource\text{”}, r) \end{array}\right|\!\!\right\} \xrightarrow{(j) \leftarrow f(i,k)} \{\!|\, \uparrow(j * Closure) \,|\!\}$$

where $f(x, y) = (diff(x, y))$

Closure

$$Closure = \left\{\!\!\left|\begin{array}{l} (\text{“}name\text{”}, k), (\text{“}c\text{”} : CODE), (\text{“}store\text{”}, st), \\ (\text{“}state\text{”}, s), (\text{“}tr\text{”} : TRANSF) \end{array}\right|\!\!\right\}$$

$$TRANSF = \left\{\!\!\left|\begin{array}{l} ?\uparrow(\text{“}name\text{”}, k), ?(\text{“}c\text{”} : CODE), \\ ?(\text{“}store\text{”}, st), ?(\text{“}state\text{”}, s), \\ ?\ \uparrow(\text{“}resources\text{”}, r), \\ (\text{“}tr\text{”} : TRANSF) \end{array}\right|\!\!\right\} \longrightarrow \left\{\!\!\left|\begin{array}{l} (\text{“}remote\_res\text{”}, k), \\ \uparrow(\text{“}ready\text{”}) \end{array}\right|\!\!\right\}$$

$$CODE = \left\{\!\!\left|\begin{array}{l} ?(\text{“}store\text{”}, st), ?(\text{“}state\text{”}, s), \\ ?(\text{“}remote\_res\text{”}, r) \end{array}\right|\!\!\right\} \xrightarrow{(s', st') \leftarrow f(r,s,st)} \left\{\!\!\left|\begin{array}{l} (\text{“}store\text{”}, st'), \\ (\text{“}state\text{”}, s') \end{array}\right|\!\!\right\}$$

where $f(res, store, state) = (nstate(res, store, state), nstore(res, store, state))$

Table 4. MobiS specification of Closure paradigm

A *Supplier* space contains the data and two rules. The rule $GETREQ$ accepts requests of data references while the rule 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-request tuple, the state, and the store. The rule $CODEREQ$ sends a code request in the main space. The rule GET gets from the main space the serialized code sent by the supplier. The rule $CODE$ 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 $GETREQ$ accepts a request of code from the main space, and the rule PUT emits the serialized code in the main space.

We omit the formalization of **Code & Store Style** and **Code, Store, & State style** as their formalizations can be easily derived from the **Code Style** specification (Table 3).

The following two styles fully exploit the power of MobiS as they are based on the mobility of spaces. Table 4 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 (“$k.Cl$” $*$ $Closure$), and two rules ($GETREQ$ and PUT). The *Closure* space is specified below in the table and $k.Cl$ 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 ($diff$ 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 (“*ready*”) in the supplier space, indicating that the closure is ready to be moved elsewhere. The $CODE$ 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 requester name in the name path of the closure space (function *concat* in the **where** clause does it).

Table 5 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, and some resources.The rule $CODE$ executes the code using the local resources to the ambient, no matter where the ambient is located. The *Requester* rule GET gets the *Ambient* space from the parent space and updates its location name.

Environment

$$Environment = \{\!\!\{ (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier) \}\!\!\}$$

Supplier

$$Supplier = \{\!\!\{ (a * Ambient), (\text{“}name\text{”}, k)(\text{“}put\text{”} : PUT) \}\!\!\}$$

$$PUT = \left\{\!\!\left| \begin{array}{l} \uparrow(\text{“}req\text{”}, req), ?(\text{“}name\text{”}, i), \\ (a * Ambient) \end{array} \right|\!\!\right\} \xrightarrow{(z) \leftarrow f(a,i)} \{\!\!\{ \uparrow(z * Ambient) \}\!\!\}$$

where $f(x, y) = (diff(x, y))$

Ambient

$$Ambient = \left\{\!\!\left| \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}data\text{”}, d), (\text{“}state\text{”}, s), \\ (\text{“}c\text{”} : CODE), (\text{“}resources\text{”}, r) \end{array} \right|\!\!\right\}$$

$$CODE = \left\{\!\!\left| \begin{array}{l} ?(\text{“}store\text{”}, st), \\ ?(\text{“}state\text{”}, s), (\text{“}resource\text{”}, r) \end{array} \right|\!\!\right\} \xrightarrow{(s', st') \leftarrow f(r,s,st)} \left\{\!\!\left| \begin{array}{l} (\text{“}store\text{”}, st'), \\ (\text{“}state\text{”}, s') \end{array} \right|\!\!\right\}$$

where $f(res, store, state) = (nstate(res, store, state), nstore(res, store, state))$

Requester

$$Requester = \{\!\!\{ (\text{“}req\text{”}, req), (\text{“}request\text{”} : REQ), (\text{“}get\text{”} : GET), (\text{“}name\text{”}, k) \}\!\!\}$$

$$REQ = \{\!\!\{ ?(\text{“}req\text{”}, req) \}\!\!\} \longrightarrow \{\!\!\{ \uparrow(\text{“}req\text{”}, req) \}\!\!\}$$

$$GET = \left\{\!\!\left| \begin{array}{l} \uparrow(a * Ambient), (\text{“}req\text{”}, req), \\ ?(\text{“}name\text{”}, z) \end{array} \right|\!\!\right\} \xrightarrow{(j) \leftarrow f(z,a)} \{\!\!\{ (j * Ambient) \}\!\!\}$$

where $f(x, y) = (concat(x, y))$

Table 5. MobiS specification of the Ambient style

4 Application of the Styles to the Architecture of a Mobile System

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

Environment

$$Environment = \{\!\!\{ (\text{“}s1\text{”} * Shop), (\text{“}s2\text{”} * Shop), (\text{“}agent\text{”} * Ambient) \}\!\!\}$$

Shop

$$Shop = \left\{\!\!\left\{ \begin{array}{l} (\text{“}name\text{”}, k)(\text{“}put\text{”} : PUT), (\text{“}req\text{”}, req), \\ (\text{“}request\text{”} : REQ), (\text{“}get\text{”} : GET)(\text{“}catalog\text{”}, l) \end{array} \right\}\!\!\right\}$$

$$PUT = \{\!\!\{ \uparrow(\text{“}req\text{”}, req), ?(\text{“}name\text{”}, i), (a * Ambient) \}\!\!\} \xrightarrow{(z)\leftarrow f(a,i)} \{\!\!\{ \uparrow(z * Ambient) \}\!\!\}$$

where $f(x,y) = (diff(x,y))$

$$REQ = \{\!\!\{ ?(\text{“}req\text{”}, req) \}\!\!\} \longrightarrow \{\!\!\{ \uparrow(\text{“}req\text{”}, req), (\text{“}wait\text{”}) \}\!\!\}$$

$$GET = \left\{\!\!\left\{ \begin{array}{l} \uparrow(a * Ambient), (\text{“}wait\text{”}), \\ ?(\text{“}name\text{”}, z), ?(\text{”}catalog\text{”}, l) \end{array} \right\}\!\!\right\} \xrightarrow{(j)\leftarrow f(z,a)} \{\!\!\{ (j * Ambient) \}\!\!\}$$

where $f(x,y) = (concat(x,y))$

Ambient

$$Ambient = \left\{\!\!\left\{ \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}state\text{”}, s), (\text{‘}bestpricefound\text{‘”}, b), \\ (\text{“}printer''}, p), (\text{“}update\text{”} : UPDATE), (\text{“}print\text{”} : PRINT) \end{array} \right\}\!\!\right\}$$

$$UPDATE = \left\{\!\!\left\{ \begin{array}{l} ?(\text{“}bestpricefound\text{”}, b), \\ ?(\text{“}state\text{”}, s), \\ \uparrow(\text{“}catalog\text{”}, l), \\ \mathbf{ask}(l < b) \end{array} \right\}\!\!\right\} \xrightarrow{(s',b')\leftarrow f(s,l)} \left\{\!\!\left\{ \begin{array}{l} (\text{“}bestpricefound\text{”}, b'), \\ (\text{“}state\text{”}, s') \end{array} \right\}\!\!\right\}$$

where $f(s,l) = (compute(s',l),l))$

$$PRINT = \{\!\!\{ ?(\text{“}bestpricefound\text{”}, b), \uparrow(\text{“}printer\text{”}, p) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}outonprinter\text{”}, b) \}\!\!\}$$

Table 6. Specification of the Purchasing Architecture in the Ambient Style

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 an other moving its code. Every store puts an advertisement request tuple, $(\text{“}newsellingprice\text{”}, reqselling)$, 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, $(\text{“}bestprice\text{”}, price)$, are sent from a store to another whenever a new advertisement tuple is received. This solution fits better the 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. The new $CODE$ rule (that has to instance the rule $CODE$ of the code&store style) 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 **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 a mobile "agent" traveling with all its data and exploiting its resources (printer, modem, cellular phone, ...) on different selling-stores looking for the best price for an item. The purchaser could, for example, use its printer to print the temporary best price found till that moment.

We give the specification of the architecture of the Purchasing System in the Ambient style in table 6. The Shops are Requester and Supplier at the same time. The resources in the Ambient Style shown in Table 5 are now portable resources (i.e. a printer, a scanner, ...). The rules PUT, REQ, and GET do the same operations. The Ambient stores the best price found and contains two rules refining the $CODE$ rule in Table 5: the rule $PRINT$ prints the best price temporary found on the portable printer.

These paradigms could also be composed. For instance in Table 5 we allows the agent to read from the shop catalog (i.e to access to some external resources). Therefore, the resulting architecture is a mixture among Code,State&Store style and Ambient style. The $UPDATE$ rule updates the best price found if the catalog of the shop offers a better price. 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 she wants to implement and can choose on the basis of these requirements the most suitable style of mobility. The resulting architecture could also be an integration among different paradigms.

5 Related work and Conclusions

We have studied different kinds of mobile entities, and shown how we specify software architectures including these mobile entities. MobiS language has been used for the formal specification. An architecture of a simplified purchasing system has been described exploiting the different architectural styles formalized.

Other formal languages have been proposed for the specification of mobility aspects of systems. In [4, 9] a general classification is given on mobility paradigms, technologies, and applications. We present a catalog where the handling of data and code is merged, We also have formalized with MobiS the paradigms defined in the classification.

In [13] a formalization of some mobility paradigms is given using Mobile Unity. Mobile Unity is expressive enough to specify the different mechanisms of mobility and the Unity logic can be used in order to reason on the specifications. However, the unit of mobility is a complete UNITY program and there is no distinction between active and inactive programs: instead, we differentiate the sending of code or data from the sending of processes with execution state.

Obliq [1] and the Ambient calculus [2] are specification languages for mobility: they both offer a particular approach to mobility based on a particular style (respectively *Closure* and *Ambient*). The Ambient Calculus is based on the movement of spaces (ambients). Whenever there is a need for movement, an ambient is created, and what has to be moved is placed inside. The whole ambient is moved. The movement is performed step by step like in MobiS. "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" [2]. Also in MobiS mobility is quite realistic: an entity can move only between spaces which are parent and son. A complete travel is composed of many of these single steps movements.

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 [8] is an extension of the π-calculus which introduces the explicit notions of named localities and distribution failure.

Klaim [6] and Bauhaus [3] 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.

The coordination model of MobiS is quite similar to the semantics framework of the CHAM, a notation that has been exploited to specify software architectures [10]. However, the CHAM does not provide mobility of solutions (that are the CHAM equivalent of MobiS spaces).

We are developing a logic for reasoning on MobiS specification, in particular we are adapting the CTL logic we used for the ancestor of MobiS: PoliS [5]. The automatic verification of properties in another goal: we plan to enhance the PoliS model checker [5] in order to deal with MobiS mobility of spaces.

We are working on the integration of MobiS inside UML. UML offers support for the specification of many different approaches to design of systems, while MobiS is a formal language for the specification of the mobility of components. The integration of the two models offers the possibility to inspect different aspects of architectures that contains mobile components.

Acknowledgments: I would like to thank Paolo Ciancarini for the helpful suggestions and for the stimulating discussion.

References

1. L. Cardelli. A language with distributed scope. In *Proc. 22nd ACM Symposium on Principles of Programming Languages (POPL)*, pages 286–298, 1995.
2. L. Cardelli and A. Gordon. Mobile Ambients. In M. Nivat, editor, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155, Lisbon, Portugal, 1998. Springer-Verlag, Berlin.
3. N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Verlag, Berlin, 1995.
4. A. Carzaniga, G. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proc. 19th Int. Conf. on Software Engineering (ICSE)*, pages 22–32, Boston, Ma, May 1997.
5. P. Ciancarini, F. Franzé, and C. Mascolo. A Coordination Model to Specify Systems including Mobile Agents. In *Proc. 9th IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 96–105, Japan, 1998.
6. R. DeNicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
7. G. DiMarzoSerugendo, M. Muhugusa, and C. Tschudin. An Survey of Theories for Mobile Agents. *World Wide Web*, 1(3):139–153, 1998.
8. C. Fournet, G. Gonthier, J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421, Pisa, Italy, August 1996. Springer-Verlag, Berlin.
9. A. Fuggetta, G. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.
10. P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.
11. J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282 of *Lecture Notes in Computer Science*, pages 18–31, Berlin, Germany, September 1997. Springer-Verlag, Berlin.
12. P.Ciancarini and C.Mascolo. Software architecture and mobility. In D. Perry and J. Magee, editors, *Proc. 3rd Int. Software Architecture Workshop (ISAW-3)*, ACM SIGSOFT Software Engineering Notes, pages 21–24, Orlando, FL, November 1998.
13. G. Picco, G. Roman, and P. McCann. Expressing Code Mobility in Mobile Unity. In M. Jazayeri and H. Schauer, editors, *Proc. 6th European Software Eng. Conf. (ESEC 97)*, volume 1301 of *Lecture Notes in Computer Science*, pages 500–518. Springer-Verlag, Berlin, 1997.
14. M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.