Software Architecture and Mobility

P. Ciancarini, and C. Mascolo
Dipartimento di Scienze dell'Informazione
Università di Bologna
Mura Anteo Zamboni 7, I-40127 Bologna, Italy
phone: +39 051 354506
e-mail: {ciancarini,mascolo}@cs.unibo.it

Abstract

Modern Software Architectures often have to deal with mobile components. Therefore, the structure of these systems is dynamic and continuously changing.

We introduce MobiS, a coordination language based on multiple tuple spaces and show through an example how it can be used for the specification of software architectures containing mobile components. The flexibility of the language encodes mobility in the model, so the specification of mobile components and of reconfigurable systems is easy.

Due to the non-determinism of the coordination model the behaviors of components context-dependent can be specified and used to make assumptions on the kind of architecture the component can be put into.
Topic Area: Dynamic Architectures and Reconfiguration.

1 Introduction

The presence of mobile components in the modern software architectures is continuously increasing: this trend requires the introduction of languages able to specify the dynamics of the systems.

Software Architecture deals with: *"the structure of the components of a program/system, their interrelationships, and guidelines governing their design and evolution over time [7]."*

In presence of mobility the "evolution over time" of the system is a key issue: every specification language with the aim of specifying this kind of architectures must be able to define the behavior of the components with respect to the system.

In this paper we introduce a coordination language for the specification of software architectures with mobile components. The coordination mechanism based on multiple tuple space allows the specification of mobility aspect at the software architectural level.

As the structure of these kind of systems is dynamic, the language used for the specification must be flexible: the multiple tuple space based model provides active rules for the movement of the components, encapsulating the mobility aspects inside the language.

Many architectural description languages have been devised for the specification of software architecture, examples include Wright [1], Darwin [10], Cham [8]. Recently, the need of the specification of the dynamic aspects besides the static ones has increased. We associate this trend in the specification of dynamics to the need of coping with mobility and reconfigurability at the software architectural specification level.

The coordination language we present deals with the creation, movement and termination of architectural components, their communication activities, their distribution as well as the synchronization of their actions over time.

As far as software engineering is concerned, the decoupling between components and their coordination realized by the coordination languages can be regarded as fostering the distinction between components and their interconnection which is at the root of research in languages and methods for software architectures [12].

The paper is organized as follows: Section 2 contains an overview of the MobiS language and an example of specification. Section 3 shows how MobiS can be used to specify software architectures with mobile components. Finally in Section 4 we illustrate some related work and conclusions.

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 [4].

The figure 1.a shows the MobiS structure of the nested spaces while figure 1.b shows the tree structure corresponding to the spaces in figure 1.a.

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, as represented by a program tuple.

A space is modified by agents that are reactions that transform multi-sets of tuples in multi-sets of tuples.

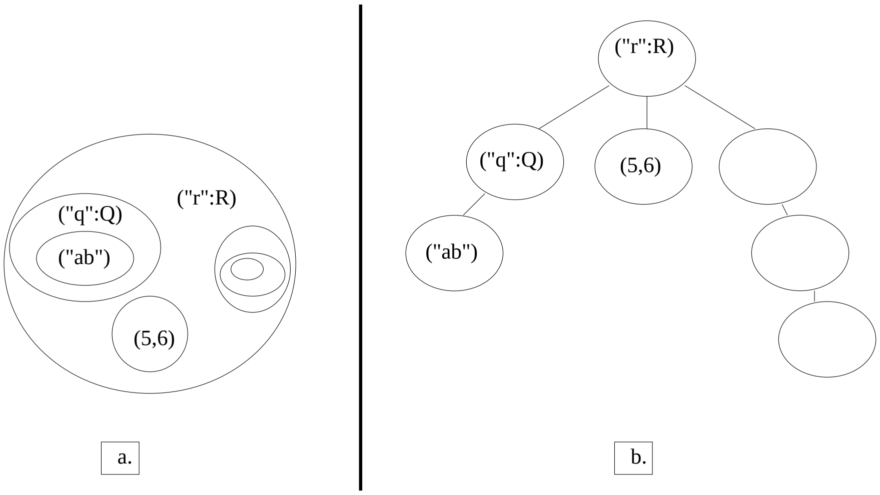

Figure 1: MobiS nested spaces and the tree of nested spaces

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_id* is a rule identifier and *rule* is the specification of the reaction (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 consists of consuming and 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 it its parent and so on.

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 components it contains.

3 Mobile Software Architecture and MobiS

Our purpose is to show how software architectures with mobile components can be specified using the MobiS language.

A component is specified in MobiS using a space. In this way each architectural component can be seen as a composition of different sub-components (i.e. sub-spaces) or, more abstractly, as a single component (i.e. a space containing only program and ordinary tuples).

The coordination model is a good framework to abstract from communication details: at the architectural level we would like to have an abstract view of the system: the tuple-based communication mechanism let the focus be put on the structure. On the other side, if the specification of the connection is important, it is possible to associate with the connector a space in order to define its particular behavior.

As the components are spaces (i.e. first class entities of the model) they can move over the system. The architecture is dynamic reconfigurable as the components makes their assumptions on the external environment and can consider different possible behaviors (due to the non-determinism of the application of the rules of the MobiS language), depending on what the environment offers.

The figure 2 contains an example of a Client-Server system where the Client and the Server exchange requests and replies. However the architecture is reconfigurable and when the network is busy the Client sends an Agent to the Server site in order to avoid heavy communication on the links. Then, the Agent and the Server communicate in the local server site. When the Agent has finished it goes back to the Client site.

This is an example of a dynamic architecture that changes its configuration depending on the context.

We can easily handle this situation using MobiS non-deterministic model that allows the components to make assumptions on the external environment and provide different behaviors with respect to the different contexts: a similar approach has been studied in [9].

In order to give the flavor of the way in which MobiS specifications can be written we show the formalization of the Client component in table 1.

The Client space contains an ordinary tuple indicating the name of the client (“*name*”, k) where k is the formal parameter containing the name. It also contains the tuple

Client

$$Client = \left\{\!\left| \begin{array}{l} (\text{``}name\text{''}, k), (\text{``}put\text{''} : PUT), (\text{``}reqlist\text{''}, r), (\text{``}get\text{''} : GET), \\ (\text{``}move\text{''} : MOVE), (\text{``}create\text{''} : CREATE), (\text{``}servername\text{''}, s) \end{array} \right|\!\right\}$$

$$PUT = \{\!| \ (\text{``}reqlist\text{''}, r), ?((\text{``}name\text{''}, k), (\text{``}idle\text{''}) \ |\!\} \xrightarrow{(t) \leftarrow f(r)} \{\!| \ \uparrow(\text{``}req\text{''}, i, t), (\text{``}wait\text{''}) \ |\!\}$$
where $f(x) = (head(x))$

$$GET = \left\{\!\left| \begin{array}{l} \uparrow(\text{``}reply\text{''}, i, r), (\text{``}reqlist\text{''}, t), \\ (\text{``}name\text{''}, i), (\text{``}wait\text{''}) \end{array} \right|\!\right\} \xrightarrow{(j) \leftarrow f(t)} \left\{\!\left| \begin{array}{l} (\text{``}requlist\text{''}, j), (\text{``}reply\text{''}, i, r), \\ (\text{``}idle\text{''}) \end{array} \right|\!\right\}$$
where $f(x) = (diff(x, head(x)))$

$$CREATE = \{\!| \ \uparrow(\text{``}networkbusy\text{''}), (\text{``}name\text{''}, i), (\text{``}reqlist\text{''}, r) \ |\!\} \xrightarrow{(a) \leftarrow f(r,i)} \{\!| \ (a * AGENT) \ |\!\}$$
where $f(x, y) = (a)$

$$MOVE = \{\!| \ (a * AGENT), (\text{``}servername\text{''}, k) \ |\!\} \xrightarrow{(j) \leftarrow f(a,k)} \{\!| \ \uparrow(j * AGENT) \ |\!\}$$
where $f(x, y) = (concat(x, y))$

$$GETAG = \{\!| \ \uparrow(a * AGENT)(\text{``}name\text{''}, k), \mathbf{ask}(prefix(a, k) \ |\!\} \longrightarrow \{\!| \ (a * AGENT) \ |\!\}$$

Table 1: Specification of the Client component

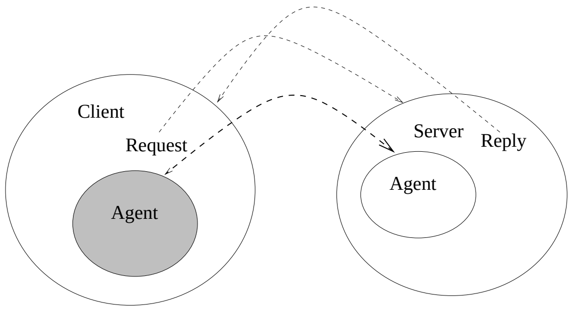

Figure 2: MobiS nested spaces and the tree of nested spaces

$(\text{``}reqlist\text{''}, r)$ of the list of the requests for the server, the name of the server $(\text{``}servername\text{''}, s)$, and some program tuples that refer to rules specified below in the table.

The rules PUT and GET handle the communication with the server when the network is not busy. The rule PUT emits in the external space (the network) a request extracting it from the requests list. The rule GET gets the reply from the Server (i.e. it checks if a reply directed to the Client is present on the network), and stores it in the local space updating the requests list (it throws the first request in the queue as it has already been served).

When the net is busy the rule $CREATE$ generates an $AGENT$ space storing the requests in it. The rule $MOVE$ moves the Agent into the network. It also changes the name of the Agent appending the name of the Server to it in order to indicate the destination of the Agent.

The last rule is $GETAG$ that gets the Agent from the network when it come back after having finished its work on the Server site.

The Client can choose the communication protocol depending on its context: when the network is not congestioned it sends requests and wait for replies, while when the network is very busy it build an Agent and sends it to the Server site to exploit local computation.

4 Related Work and Conclusions

Modern software architectures often deal with mobile components and there is a need of formalization of new architectural patterns based on mobility. MobiS has an interesting application in the specification of mobility aspects: the coordination allows flexible moving of components and extensibility of the architecture.

The MobiS language encodes mobility as a feature in its model and the specification of architectures in presence of mobile components is immediate. The space is a first class entity of the language: spaces represent components and can be nested, in this way compositional mechanisms can be exploited.

New formal languages are being proposed for the specification of mobility features; a short list includes Bauhaus [3], Ambit [2], Join Calculus [6], Klaim [5], and Mobile Unity [11].

Our purpose has been is to focus on the architectural aspects of systems with mobile components and to reason about mobility at the architectural level.

We are studying how security aspects can be introduced and analyzed at the software architecture level. We are also reasoning on the possible assumptions that every mobile component can make on the other components of the architecture: MobiS model of scoping of the reactions can be exploited for this kind of reasoning.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[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] 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.

[5] 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.

[6] 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.

[7] D. Garlan and D. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.

[8] 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.

[9] P. Inverardi, A. Wolf, and D. Yankelevich. Checking assumptions in components dynamics at the architectural level. 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 46–63, Berlin, Germany, September 1997. Springer-Verlag, Berlin.

[10] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, April 1985.

[11] 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.

[12] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.