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

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.

The presence of mobile components in the modern software architectures often have to deal with mobile components. Therefore, the structure of these systems is dynamic and continuously changing.

We introduce MobiŚ, 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.

1.1 Overview of MobiŚ

MobiŚ is a specification language based on multiple tuple spaces. MobiŚ specifications are hierarchically structured: a MobiŚ specification denotes a tree of nested spaces that dynamically evolves in time. It is an enhanced version of PoliŚ [4].

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

A MobiŚ space is a first class entity, and can move. Formally, a MobiŚ 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.
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

![Figure 1: MobiS nested spaces and the tree of nested spaces](image-url)
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 congested 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.

Table 1: Specification of the Client component

\[
\begin{align*}
\text{Client} = \{ & \{ \text{"name"}, k \}, \{ \text{"put"}, \text{PUT} \}, \{ \text{"reqlist"}, r \}, \{ \text{"get"}, \text{GET} \}, \\
& \{ \text{\"move\", \MOVE} \}, \{ \text{\"create\", \CREATE} \}, \{ \text{\"servername\", s} \} \}
\end{align*}
\]

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

\[
\begin{align*}
\text{PUT} = \{ & \{ \text{\"reqlist\", r} \}, \{ \{ \text{\"name\", k}, \{ \text{\"idle\"} \} \}, \{ \{ \text{\"req\", i}, \{ \text{\"wait\"} \} \} \} \\
& \text{where } f(x) = (\text{head}(x))
\end{align*}
\]

\[
\begin{align*}
\text{GET} = \{ & \{ \text{\"reply\", i}, \{ \text{\"reqlist\", t} \}, \{ \{ \text{\"req\", i}, \{ \text{\"wait\"} \} \} \} \\
& \text{where } f(x) = (\text{diff}(x, \text{head}(x))]
\end{align*}
\]

\[
\begin{align*}
\text{CREATE} = \{ & \{ \{ \\text{\"network\", b} \}, \{ \text{\"name\", i} \}, \{ \text{\"reqlist\", r} \} \}, \{ \{ \text{\"name\", i}, \{ \text{\"idle\"} \} \} \} \\
& \text{where } f(x, y) = (x)
\end{align*}
\]

\[
\begin{align*}
\text{MOVE} = \{ & \{ \text{\"a\", \AGENT} \}, \{ \text{\"servername\", k} \}, \{ \{ \text{\"a\", \AGENT} \} \} \\
& \text{where } f(x, y) = (\text{concat}(x, y))
\end{align*}
\]

\[
\begin{align*}
\text{GETAG} = \{ & \{ \text{\"a\", \AGENT} \}, \{ \text{\"name\", k}, \text{\"reply\", i, t} \}, \{ \{ \text{\"a\", \AGENT} \} \}
\end{align*}
\]

4 Related Work and Conclusions

Modern software architectures often deal with mobile components and there is a need for 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 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


