A Higher Order pi-Calculus Specification for a Mobile Agent in JINI

Barbu, A. and Mourlin, F.

A Higher Order $\pi$-Calcl Specifcation for a Mobile Agent in JINI

Andreea Barbu, Fabrice Mourlin

Address:
LACL, Département of Computer Science
61 avenue du Général de Gaulle 94010 Créteil Cedex, France

Tel. : 00 33 /0|1 45 17 65 96
Fax : 00 33 /0|1 45 17 66 01

E-Mail : barbu@univ-paris12.fr, Fabrice.Mourlin@wanadoo.fr

Topic Areas: Distributed computing technology, Parallel and Distributed systems, Internet computing; JAVA, JINI, CORBA, XML, Data Communications.
A HIGHER ORDER π-CALCULUS SPECIFICATION FOR A MOBILE AGENT IN JINI

ANDREEA BARBU, FABRICE MOURLIN

Abstract. Current technologies of distribution of code have arrived for a few years to a good degree of maturity, in particular with the appearance of distributed execution platforms in conformity with CORBA specifications and recently with the appearance of Java/Jini technologies [1]. This work proposes to bridge the gap between the requirements of mobile code generation for communicating systems and the constraints due to the writing of specifications describing the mobility. Milner’s Pi-calculus [2] is found to offer many features required to bridge this gap. It is powerful enough to be able to represent mobile systems and to reason on their behavior. The higher order Pi-calculus [1] gains even more expressive power by allowing mobile agents to pass as values in a communication. The mobile agents are very powerful and especially general-purpose. Java provides an ideal implementation platform, furnishing tools that help streamline complex software applications. Java’s Jini [1] framework facilitates mobile agent application development, providing key features for distributed network programming. This work provides a formal approach of higher order Pi-calculus where mobile agents are specified and whose prototype is generated in Jini.

1. Introduction

Mobile agents are omnipresent in today’s software applications and Java is a significant language to develop these agents. For a few years, Jini (Java Intelligent Network Interface) [1] has been more and more essential on the framework market, allowing the development in distributed networks. Jini is a tool based on Java and its enables us to realize, in a rather simple way, shared applications. Just as robots automate many aspects of manufacturing a computer, JINI automates and abstracts distributed applications’ underlying details. These details include the low-level functionality (socket communication, synchronisation) necessary to

Key words and phrases. π-Calculus, Jini, RMI, shared computing, mobile computing.
implement high-level abstractions (such as service registration, discovery, and use) that JINI provides.

The ideas behind JINI corresponding to Jim Waldo’s description (from Sun Microsystems) are simple: “JINI is designed so that chunks of code can find other chunks of code without people.” JINI was designed assuming that the network is not reliable. Things join the network and leave the network. There is no central control. Also, JINI blurs the distinction between hardware and software, dealing only with services.

The objective of this work is to describe the modelling of mobile agents or code in Java integrated within the JINI platform. Mobile agents or transportable agents, are codes which move on the network to fulfill a mission. For a better understanding of their behavior and to validate properties of them, such as for example, their return on their starting machine, it is necessary to formalize these aspects. A first realized study leads to a higher order Pi-calculus formalisation [2].

We use the formal language higher order Pi-calculus for the specification of the mobile agent and we chose the JINI framework for its development. It is very significant to accentuate the higher order aspect because, to be able to develop a mobile agent, we need a language which is able to express it like an essential characteristic.

A mobile agent is a piece of code which achieves a task required by an user. It must have a certain mobility to be able to move between various computers.

In our case study, we are going to focus on the mobile aspect of agents that can move around between different servers (a three rooms HOUSE (FIRSTROOM, ROOM1, LASTROOM) to accomplish a task given by a task manager (TASK - LOOKUP): “collect all the books you find on your way” TASK1(get(book)).

2. Case study

Our objective is to simulate a ROBOT which moves in a HOUSE with three rooms. Let us consider the following situation where a ROBOT receives a task (to collect all the books) from the task manager (TASKLOOKUP). With this task it will cross the first gate entry to get in the FIRSTROOM. As soon as it is inside, the gate entry will be closed to prevent the other ROBOTS to get in the room. The ROBOT will carry out the task (TASK1(get(book))) and after, it will cross
the second gate gate\(_1\) to pass in the second room ROOM\(_1\). As soon as it is inside, the first gate entry will open to let pass the other ROBOTS, on the other hand the gate gate\(_1\) will close. After having passed ROOM\(_1\) and LASTROOM in the same way and by collecting all the books, the ROBOT will leave the HOUSE by the last gate exit and it will return to the task manager. The higher order aspect in our case study is represented through the move of the agent ROBOT and agent TASK\(_1\). The two agents are passed on channels like a simple name. The diagram 1. depicts this simulation.

![Diagram of a house system](image)

**Figure 1.** Our House system

2.1. **Specification Part.** The recent developments of data processing in a network confront the field of analysis and checking of parallel systems with new and complex questions. One concept which is the core of this evolution is that of mobility (of code, and more general of calculus over a network whose communication topology is dynamically changing).

A specification language must be powerful enough to express the mapping of a simulation model to any protocol on any target mobile architecture. The Pi-calculus expresses the move of data but also of code. This offers a solid base to make use of the JINI generation. The Higher Order paradigm is a construct where mobility is achieved by allowing agents to be passed as values in a communication. The prototypical calculus in the first-order paradigm is the \(\pi\)-Calculus that was introduced by Milner, Parrow and Walker in [5] and later refined by Milner [6] with the addition
of sorts and of communication of tuples (Polyadic π-Calculus). The π-Calculus is a way of describing and analyzing systems consisting of agents which interact among each other, and whose configuration or neighborhood is continually changing. This model of concurrent computation based upon the notion of naming. The most primitive entity in π-Calculus is a name which refers to a link or a channel. Semantics is done in terms of a reduction system and a version of labelled transitions called commitment. The Higher Order π-Calculus (HOπ) is an extension of the first order π-Calculus introduced by D. Sangiorgi [4]. This calculus enriches the π-Calculus with explicit higher order communications. In the HOπ-Calculus not only names, but also agents of arbitrarily high order, can be transmitted. The syntax of the HOπ-Calculus is an extension of the syntax of the first order π-Calculus. Let be \( a, b, \ldots, x, y, \ldots \), a set of names and \( P, Q, \ldots \), a set of processes. The syntax of the HOπ-Calculus is an extension of the syntax of the π-Calculus [5]:
\[
P ::= \bar{x}(K).P | x(U).P | \tau.\overline{P} | \nu.x.P | P + Q [x = y]P
\]
where \( K \) is an agent or name, and \( U \) is a variable or name and:

- \( \bar{x}(K).P \) can send the name or process \( K \) via the name \( x \) and continue as \( P \).
- \( x(U).P \) can receive any name or variable \( U \) and continue as \( P \) with the received name substituted for \( U \).
- in the composition \( P|Q \), the two components can proceed independently and interact via shared names or processes.
- \( \tau.\overline{P} \) is an invisible action to \( P \). Can be an internal action of a process.
- \( \nu.x.P \) is called the restriction and means that the scope of name \( x \) is restricted to \( P \).
- in the sum \( P + Q \) either \( P \) or \( Q \) can interact with other processes.
- The matching \([a = b]P\) denotes the activation of a process which is selected by other processes on depend of a condition \((a = b)\).

The rules to define the basic Pi-calculus are subdivided in two groups: the rules of the structure congruence and the transition and communication rules. They are described by Milner [5].

The extension of the operational semantics is given by Sangiorgi [4]:
ComHO_1 : 

\[ \frac{P \models K \quad P', Q \models (U) \quad Q}{P \cup Q \models [U]} \]

ComHO_2 : 

\[ \frac{P \models (U) \quad P', Q \models K \quad Q}{P \cup Q \models [K/U]} \]

For the two rules of the operational semantic it is necessary to notice that the amount of numbers of parameters of \( U \) is the same as the numbers of parameters of \( K \): \( \text{Arith}_{HA} p(U) = \text{Arith}_{HA} p(K) \) with \( A \) a set of actions \( x, y, \ldots \) and \( P \) a set of agents \( P, Q, \ldots \). This restriction is necessary to be enrich with a type control over the signature of \( U \) and \( K \). The adding of the type control [7] allows us to use the overloading into the agent definition.

The specification of our case study is given by our system \( HOUSE \) and all elements are in a parallel relation to each other to be able to communicate.

\[ HOUSE = TASKAGENT(go) \]
\[ |(v \text{ input})| (TASKLOOKUP(export, input, go) | TASKLOOKUP_mem(input)) \]
\[ ENTRY(entry, entrystate_1, entrystate_2) \]
\[ FIRSTROOM(entry, get, put, entrystate_1, entrystate_2, gatestate_1, gatestate_2, gatestate_3, gateName) \]
\[ GATE_1(entrystate_1, entrystate_2, gatestate_1, gatestate_2, gatestate_3, gateName) \]
\[ ROOM_1(entry, get, put, entrystate_1, entrystate_2, gatestate_1, gatestate_2, gatestate_3, gateName) \]
\[ GATE_2(entrystate_1, entrystate_2, gatestate_1, gatestate_2, gatestate_3, gateName) \]
\[ LASTROOM(entry, get, put, entrystate_1, entrystate_2, gatestate_1, gatestate_2, gatestate_3, gateName) \]
\[ EXIT(entry, exitstate_1, gateName) \]

\[ TASKAGENT(go) = \exists TASK_1(entry, book). TASKAGENT(go) \]

\[ TASK_1(entry, obj) = get(obj). TASK_1(entry, obj) \]

\[ TASK_2(entry, thing) = put(thing). TASK_2(entry, thing) \]

\[ TASKLOOKUP(export, input, go) = (v k) go(T[k]). TASKLOOKUP_mem(input) \]
\[ TASKLOOKUP_mem(input) = (v l) input(T[l]). TASKLOOKUP(export, input, go) \]

\[ ROBOT(export) = (v m) export(T[m]). (T[m]). ROBOT(export) + \tau.0 \]
\[ ENTRY(entry, entrystate_1, entrystate_2) = entry[ROBOT(export)]. entrystate_1(closed) \]
\[ ENTRYCHECK(entry, entrystate_1, entrystate_2) \]
\[ ENTRYCHECK(entry, entrystate_1, entrystate_2) = entrystate_2(z). \]
(z = open)\text{ENTRY}(entry,\text{entrystate}_1, \text{entrystate}_2) +
\text{z = closed}\text{ENTRYCHECK}(entry,\text{entrystate}_1, \text{entrystate}_2)
\text{FIRST ROOM}(entry, \text{get, put, entrystate}_1, \text{entrystate}_2, \text{gate}_1\text{state}_1, \text{gate}_1\text{ename}) =
\text{entry}(R(e)) (R(e)| \text{book} \oplus \text{eraser}) + \text{put}(y).
\text{FIRST ROOMCHECK}(entry, \text{get, put, entrystate}_1, \text{entrystate}_2, \text{gate}_1\text{state}_1, \text{gate}_1\text{ename})
\text{ENTRY}(entry, \text{gate}_1\text{state}_1, \text{gate}_1\text{state}_2, \text{gate}_1\text{state}_3, \text{gate}_1\text{ename}) =
\text{gate}_1\text{state}_1(\text{open}), \text{gate}_1\text{ename}(\text{gate}_1).
\text{GATE}_1\text{CHECK}(entry, \text{get, put, entrystate}_1, \text{entrystate}_2, \text{gate}_1\text{state}_1, \text{gate}_1\text{ename}) =
\text{gate}_1\text{state}_1(\text{k}).
\text{GATE}_1\text{CHECK}(entry, \text{get, put, entrystate}_1, \text{entrystate}_2, \text{gate}_1\text{state}_1, \text{gate}_1\text{ename}) =
\text{gate}_1\text{state}_1(\text{open}).
\text{ROOM}(entry, \text{get, put, gate}_1\text{state}_1, \text{gate}_2\text{state}_1, \text{gate}_1\text{state}_3, \text{gate}_1\text{ename}) =
\text{gate}_1\text{state}_2(\text{open}), \text{gate}_1\text{state}_2(\text{closed}).
\text{ROOMCHECK}(entry, \text{get, put, gate}_1\text{state}_1, \text{gate}_2\text{state}_1, \text{gate}_1\text{state}_3, \text{gate}_1\text{ename}) =
\text{gate}_2\text{state}_1(\text{f}), [(\text{f = open})\text{gatename}(\text{d}) \text{R}(e)) \text{gate}_1\text{state}_2(\text{open}).
\text{GATE}_2\text{CHECK}(entry, \text{get, put, gate}_1\text{state}_1, \text{gate}_2\text{state}_1, \text{gate}_2\text{state}_5, \text{gate}_1\text{ename}) =
\text{gate}_2\text{state}_1(\text{open}), \text{gate}_2\text{state}_1(\text{closed}).
\text{CHECK}_3(\text{gate}_2\text{state}_1, \text{gate}_2\text{state}_4, \text{gate}_2\text{state}_5, \text{gate}_1\text{ename}) =
\text{gate}_2\text{state}_2(\text{t}).
\text{CHECK}_4(\text{gate}_2\text{state}_1, \text{gate}_2\text{state}_4, \text{gate}_2\text{state}_5, \text{gate}_1\text{ename}) =
\text{gate}_2\text{state}_3(\text{w}).
\text{LAST ROOM}(entry, \text{get, put, gate}_2, \text{gate}_2\text{state}_4, \text{gate}_2\text{state}_5, \text{gate}_1\text{ename}) =
\[ gate_2(R(e)) . (R(e) | get(book) + put(y)) . gate_2 \text{state}_2(closed) . gate\text{name}(r) . \mathcal{P}(R(e)) . \]
\[ gate_2 \text{state}_2(open) . \text{LASTROOM}(get, put, gate_2, gate_2 \text{state}_4, gate_2 \text{state}_5, gate\text{name}) \]
\[ \text{EXIT}(exit, gate\text{name}) = gate\text{name}(exit) . \text{exit}(R(e)) . \text{EXIT} \]

We specified as follows:

1. *HOUSE*: it describes our whole system with all the components.
2. *TASKAGENT*: it corresponds to the mobile service which notifies its availability to the *TASKLOOKUP*.
3. *TASK_1* and *TASK_2* describe the service in our case get(book) (collect all the books) or put(book) (deposit books).
4. *TASKLOOKUP*: plays a kind of reference book of all the tasks which are available on the local network.
5. *TASKLOOKUP* men: are linked to the *TASKLOOKUP* and manages its knowledge.
6. *ROBOT*: it describes the mobile agent which is able to migrate to accomplish its task *TASK_1*.
7. *FIRSTROOM, ROOM_1* and *LASTROOM* represent services which make available "things" like books, pens, eraser that *ROBOT* can collect.
8. *ENTRY, GATE_1, GATE_2, EXIT*: are also services which make available their gates to allow the access to the rooms.
9. *ENTRYCHECK, FIRSTROOMCHECK, GATE_1, CHECK, CHECK_1, CHECK_2, ROOM_1CHECK, CHECK_3, CHECK_4*: are technical agents, we need these to be able to make choices and to terminate the given task.

We use gates like *entry* to specify the communication channel between the agents *ENTRY* and *FIRSTROOM*. In this specification the mobility is described by the channel parameters. In our system, the *ROBOT* asks the *TASKLOOKUP* for a task and after receiving of this *TASK(get(book))*, it will be send through the first gate *entry* in the *FIRSTROOM*, etc. We use also constants like *open* and *closed* to simulate our gate state.

In any case, the system can be modelled with several *ROBOTS* which carry out different tasks. Our restriction in this system is that only one *ROBOT* can stop in
a room to do its work. After he’s gone, the room becomes free to welcome other agents.

2.2. Implementation Part. The word "agent" has found its way into a number of technologies. It has been applied to aspects of artificial intelligence research and to constructs developed for improving the experience provided by collaborative online social environments. It is a branch on the tree of distributed computing. What can we use agents for? Mobile agents come in a variety of flavors and perform numerous functions:

(1) **An information agent** searches for information residing on remote nodes and reports back to the source.

(2) **A computation agent** seeks underutilized network resources to perform CPU-intensive processing function.

(3) **A communication agent** couriers messages back and forth between clients residing on various network nodes.

Let’s now look at a mobile agent framework that consists of two main components. This framework uses the JINI/RMI platform given by [BM02] and implemented by [3]. The first component is the mobile agent ROBOT, that is an entity with some job to do. The second component is the mobile agent host TASKLOOKUP, the service that provides the mobile agents’ execution platform. In a distributed environment, we can have one-to-many agent hosts as well as one-to-many agents. To be an active agent platform, a given node in the system must have at least one active agent host.

These two components map quite nicely to the JINI model. JINI, at the highest level, provides the infrastructure that enables clients to discover and use various services. JINI also provides a programming model for developers of JINI clients and services. In the context of this mobile agent framework, the agent host(s) provides JINI services. The mobile agent, the ROBOT, is the JINI client. JINI services, FIRSTROOM, ROOM1, LASTROOM, TASK1, register one or more JINI services by providing a service proxy for perspective clients. In turn, clients query the lookup service(s) for particular services that might be of interest. The JINI services TASK1, FIRSTROOM, ROOM1 and LASTROOM will register at one or more JINI Lookups (in our case at the JINI Lookup Service 1 (TASKLOOKUP)
and JINI Lookup Service 2 (registers the ROOMS). Figure 2. depicts this process and considers the notations of our higher order Pi-calculus specification for the JINI platform.

![Diagram](image)

**Figure 2.** Service registration and discovery.

2.2.1. **Agent host construction.** In our case study we will represent the agent hosts through FIRSTROOM, ROOM1, LASTROOM, TASK12. Each of them publishes a service at the corresponding JINI Lookup service. The ROOMS will publish a service like "let the ROBOT passing trough a gate" and they will make use of the gates entry, gate1, gate2, and exit. The agent TASK12 will publish the service get(book).

The first step in building the agent host is to create a remote interface, the service template that agents will look for via the JINI lookup service (1 and 2). The **AgentHostRemoteInterface** provides one method, **acceptAgent()**, which agents call to travel to the implementing agent host:

```java
public interface AgentHostRemoteInterface extends Remote {
    public void acceptAgent (AgentInterface ai) throws RemoteException;
}
```
The beauty of JINI is that objects can publish several interfaces that provide multiple services. For instance, if we had a distributed data warehouse, we might have an agent host that provides a local data access service. In this instance, a data-mining agent might look for a host that provides the data access service and move to that host to perform localized mining operations. Therefore, we can have agents with different missions share hosts that provide multiple services.

The second step in building the agent host is to provide an implementation of this remote interface that is the actual JINI service $T_\text{ASK}_1$ with $\text{get}(\text{book})$. We have provided a $\text{MobileAgentHost}$ class that implements the $\text{AgentHostRemoteInterface}$. Figure 3. shows the class diagram for $\text{MobileAgentHost}$. The class extends the $\text{java.rmi.server.UnicastRemoteObject}$ class, which allows clients to obtain a remote reference and call its methods. The $\text{MobileAgentHost}$ also implements the $\text{ServiceIDLListener}$ interface, which is passed as unique $\text{ServiceID}$ object via the $\text{serviceIDNatify()}$ method when the service first registers with a JINI lookup service (1 and 2).

The $\text{MobileAgentHost}$ constructor takes the $\text{agentObject}$ object reference stored as member data and passed to arriving agents via $\text{doWork()}$ method. The constructor itself performs two basic functions. First, it creates a $\text{LookupDiscoveryManager}$ to locate the JINI lookup service 1 and 2. Second, it creates a $\text{JoinManager}$, with the $\text{LookupDiscoveryManager}$ as a parameter, to add this lookup service (for

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{mobile_agent_host_class_diagram.png}
\caption{Mobile agent host class diagram}
\end{figure}
example $\text{TASK}_1$) to the JINI service federation. In the $\text{acceptAgent()}$ method's implementation, the $\text{MobileAgentHost}$ binds incoming agent to an $\text{AgentThread}$:

```
public void acceptAgent(AgentInterface ai) throws RemoteException
{
    AgentThread at = new AgentThread(ai);
    at.start();
}
```

In turn, an inner class instance, $\text{AgentThread}$, is created to run the bounded agent by calling its $\text{doWork()}$ method, passing the $\text{LookupDiscoveryManager}$ and agent object references. For each arriving agent a new thread is created.

2.2.2 Agent construction. In our case study the JINI client is the mobile agent $\text{ROBOT}$ which collects the books $\text{get(book)}$ by looking at the $\text{ROOMS}$ through the gates $\text{entry, gate}_1, \text{gate}_2, \text{exit}$. The $\text{ROBOT}$ receives five proxy objects from the JINI Lookup Service 1 and 2 and so it becomes able to communicate with the services through the proxy. The first step in building an agent is to create an interface for remote services. For this, we create an $\text{AgentInterface}$ that extends the $\text{Serializable}$ interface. The $\text{Serializable}$ interface marks the implementer as a serializable entity, or one that can be sent across the wire:

```
public interface AgentInterface extends Serializable
{
    public void doWork(LookupDiscoveryManager ldm, Object workObject);
}
```

The $\text{AgentInterface}$ consists of a $\text{doWork()}$ method that is called when an agent arrives on a given host. This method takes two parameters. The first parameter is a reference to the $\text{LookupDiscoveryManager}$ maintained by the current host. The agent uses this reference if and when it decides to look for new service providers, such as when it wants to travel to a new agent host. The second parameter is an optional (possibly null) object parameter, which contains data necessary for the agent to complete its job. The second step in agent construction is providing an $\text{AgentInterface}$ implementation for which an abstract $\text{MobileAgent}$ class was
created. This class's constructor builds a service template that locates services of type `MobileAgentHostInterface`. It also provides three additional methods: `doWork()`, `moveToRandomHost()`, and `getMobileAgentHosts()`:

1. To perform an agent-specific task, subclasses override the abstract `doWork()` method.
2. When the agent wants to move, subclasses call the `moveToRandomHost()` method, which performs the following three steps:
   - Gets a list of the currently available mobile agent hosts with a call to `getMobileAgentHosts()`.
   - Randomly selects a host from this list.
   - Moves to a new host by calling the `acceptAgent()` method. If the call on the selected host fails, selects a new host.
3. To obtain a list of currently available agent hosts, subclasses call the `getMobileAgentHosts()` method. This process requires the following steps:
   - Call `getRegistrar()` to obtain a current list of lookup services.
   - Iterate through each lookup service to find services that match the desired template; in this case, `AgentHostRemoteInterfaces`. Note that we might have duplicated the same agent host registered with multiple lookup services. The `myROBOTServiceTemplate` object, a `ServiceTemplate` class instance, passed to the lookup() method initializes in the `MobileAgent` constructor.
   - Add each matching service to a vector of `AgentHostRemoteInterfaces`.

Figure 4. depicts the interactions between the `MobileAgent`, `MobileAgentHost`, and the Jini lookup service 1 and 2.

3. Conclusion

This work represents our results and underlines the connection between the constraints of specification of higher order Pi-calculus and the mobile generation of code for the communicating systems. The specification part of our system (emphasizing mobility) was carried out by the use of the higher order Pi-calculus and for the part of execution (the migration of agents in a network) we used the Java-Jini framework. Our approach is validated by the construction of a prototype respecting the
same constraints as in the specification and the tests which we passed constituting the final point of our validation.

References


[4] D. Sangiorgi: From $\pi$-Calculus to Higher-Order $\pi$-Calculus — and back, work supported by the ESPRIT BRA project "CONFER".

