FROM π-CALCULUS SPECIFICATION TO SIMULATION OF A MOBILE AGENT USING JINI

Andreea Barbu and Fabrice Mourlin

University of Paris 12, LACL,
61. avenue du Général de Gaulle
94010 Créteil, France
barbu@univ-paris12.fr, fabrice.mourlin@wanadoo.fr

KEYWORDS: π-Calculus, process algebra, mobile agents, Jini

Abstract

This paper presents a formalism related to mobile agents. It describes the higher order π-calculus, an extension of the π-calculus who was introduced by R. Milner and later developed by Sangiorgi. This formalism defines a mathematical framework that can be used to reason about mobile code; it varies greatly in its expressiveness, in the mechanism it provides to specify mobile code based applications and in its practical usefulness for the validation and the verification of such applications.
In this paper we show how this formalism can be used to represent the mobility and communication aspects of a mobile code environment called $HoPiTool$. We developed the $HoPiTool$, which is a Jini-based tool to implement mobile agents who respect an initial higher order π-calculus specification. We will also introduce the structure of the $HoPiTool$ with its most important elements.

1. INTRODUCTION - WHAT IS AN AGENT IN OUR FRAMEWORK?

Mobile agents are omnipresent in today's software applications and Java is a significant language to develop these agents. For a few years, Java Intelligent Network Interface (Jini) [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 implement high-level abstractions (such as service registration, discovery, and use) that Jini provides. 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 first to describe the modelling of mobile agents and second to implement a Java Tool, which is able to generate Jini code from our π-calculus specification. Mobile agents or transportable agents are codes, which move on the network to fulfil 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 [2] leads to a higher order π-calculus formalisation. We use the formal language higher order π-calculus for the specification of the mobile agent and we chose the Java/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. After a correct specification of our mobile system, we can generate Jini code with the help of our $HOPiTool$ and the generated files can communicate with each other in a network.
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. Figure 1. describes the framework components. This component scheme can be mapped easily to the 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.

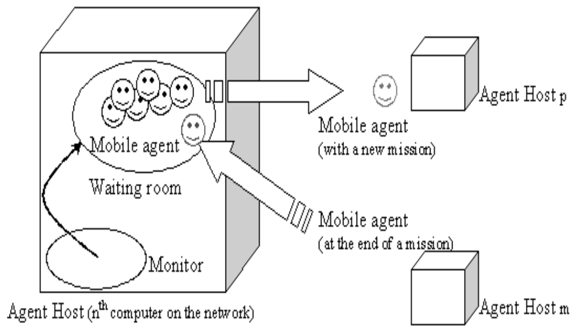
Mobile agent
Waiting room
Monitor
Agent Host (n^th computer on the network)
Mobile agent (with a new mission)
Agent Host p
Mobile agent (at the end of a mission)
Agent Host m

Figure 1. The software architecture of our framework

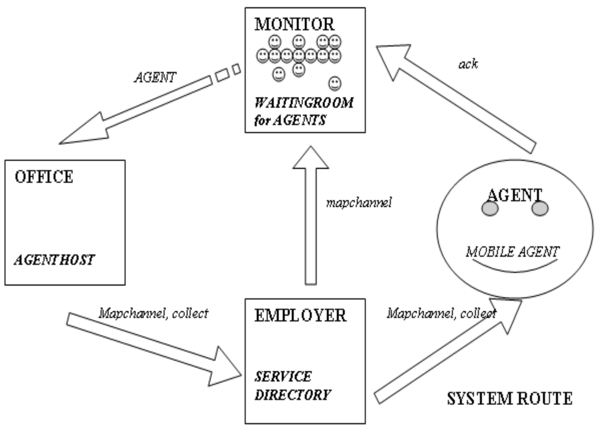

Figure 2. Our Route System

1.1. Case study

Our objective is to simulate a $MOBILEAGENT$ that wants to get a job from a service collector called $EMPLOYER$. The $MOBILEAGENT$ operates in a $ROUTE$ system composed by a finite number of agent hosts, which we call $OFFICES$. Each $OFFICE$ is able to make jobs and routes available by registering these to the $EMPLOYER$ system. The $EMPLOYER$ contacts the $MOBILEAGENT$ to give him a certain job to do (in our case it will be $collect$) and a route map called $Mapchannel$. With this information, the $MOBILEAGENT$ contacts directly the $OFFICE$, moves and gets the desired information from him (import of SQL orders into its bag). A $MOBILEAGENT$ can ask also for services and contact many $EMPLOYERS$. It is possible to have more than one $EMPLOYER$ service for a big $ROUTE$ system. We can specialize many $EMPLOYERS$ in order to make the offers of services manageable. To simplify matters we will not discuss this option in our paper. The $MONITOR$ is a waiting room, where all the $MOBILEAGENTS$ wait for some jobs. The $MONITOR$ is also the starting and the ending machine for the $MOBILEAGENTS$. After carrying out the job, an acknowledgement ack will be sent back to the monitor. The higher order aspect in our case study is represented through the $MOBILEAGENT$ that contains the job to do ($collect$), the route map ($Mapchannel$) or even another agents. Figure 2. depicts this simulation. In the context of our mobile agent framework, the agent host(s) provides Jini ”collect” services. The mobile agent(s) is the Jini client. It can have a lease for on particular node and this one can be modify during the action of the agent. Our ”collect” service provides a set of SQL statements which have to be executed later on a database. Jini services register with one or more Jini lookup-services by providing a service proxy for perspective clients. In turn, clients query the lookup service(s) for particular services. For detail information about Jini structure look at [1, 3].

2. 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 [3] 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 π-Calculus that was introduced by Milner, Parrow and Walker in [6] and later refined by Milner [5] 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. The Higher Order π-Calculus (HOπ) is an extension of the first order π-Calculus introduced by D.Sangiorgi [7]. 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. A higher-order π-Calculus process is given by the following syntax:
$P \ ::= \sum \alpha_i.P_i \mid P_1|P_2 \mid P_1 + P_2 \mid \nu\, x.P \mid [x = y]P \mid$

$D\langle\widetilde{K}\rangle \mid X\langle\widetilde{K}\rangle$
$\alpha ::= x(\widetilde{U}) \mid \overline{x}\langle\widetilde{K}\rangle$
Where X is an agent (process) variable, $\langle\widetilde{K}\rangle$ stands for any tuple of agent or (channel) name, and $\langle\widetilde{U}\rangle$ stands for any tuple of variable or (channel) name. The constants D are defined as $D \overset{def}{=} (\widetilde{U})P$. Constants are to be seen as functions whose parameters can be processes or other functions 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 $P_1|P_2$, the two components can proceed independently and interact via shared names or processes.
- $\nu\, x.P$ is called the restriction and means that the scope of name x is restricted to P.
- in the sum $P_1 + P_2$ either P_1 or P_2 can interact with other processes.
- The matching $[x = y]P$ denotes the activation of a process which is selected by other processes on depend of a condition ($[x = y]$).

The difference between first-order and higher-order π-Calculus resides in the fact that parameters can be channels and/or processes in the higher-order π-Calculus, while in firs-order π-Calculus only channels can be passed as parameters.
Example: in $\overline{x}\langle P\rangle.Q|x(X).X$, once the interaction between the two processes has taken place, the resulting process is $Q|P$. Indeed, process $x(X).X$ was waiting for X to be sent along channel x, i.e., it was waiting for a process X defining its subsequent behavior.
The operational semantics is given in terms of a labeled transition system. There are three labels for the transitions: the silent step τ, the input action $x\langle\widetilde{K}\rangle$ and the output action $\overline{x}\langle\widetilde{K}\rangle$.

Output action: $\overline{x}\widetilde{K}.P \xrightarrow{\bar{x}\langle\widetilde{K}\rangle} P$ means that after having sent message $\widetilde{K}$ (tuples of channels or processes) over channel x, process $\overline{x}\widetilde{K}.P$ behaves like P.

Input Action: $x(\widetilde{K}).P \xrightarrow{x\langle\widetilde{U}\rangle} P\{\widetilde{U}/\widetilde{K}\}$ means that if message $\widetilde{U}$ (tuples of channels or processes) is sent over channel x, then the process $x(\widetilde{K}).P$, waiting for a process or channel name on x, receives it and instantiates the process or channel name to $\widetilde{U}$, it then behaves like P, where all occurrences of $\widetilde{K}$ are replaced by $\widetilde{U}$. $\langle.\rangle$ stands for real parameters, while $(.)$ stands for formal parameters.

Interaction between two processes:

$$\frac{P \xrightarrow{(\nu\widetilde{K})\bar{x}\langle\widetilde{K}\rangle} P', Q \xrightarrow{x\langle\widetilde{K}\rangle} Q'}{P|Q \xrightarrow{\tau} \nu\widetilde{K}(P'|Q')}$$

$\widetilde{K} \cap fn(Q) = \emptyset$ means that if an output action causes P to becomeP', and the corresponding input action causes Q to become Q', then P and Q in parallel become P' and Q' in parallel, and the private (bound) process $\widetilde{K}$ emitted by P becomes a private (bound) process of $P'|Q'$.
Equivalences: Bisimulation usually identifies processes with the same external behavior. Higher-order bisimulation identifies higher-order processes if their interactions with the environment are the same and if their internal processes are bisimilar.
The case study HOπ specification is given below:
$ROUTE =$

$\nu\,(out, out_{AH}, out_A, ack, service)$
$OFFICE(out, mapchannel, collect)$
$\mid EMPLOYER(out, out_{AH}, out_A)$
$\mid MOBILEAGENT(out_A, service, ack)$
$\mid MONITOR(out_{AH}, ack)$

$OFFICE(out) =$
$\nu(mapchannel, collect, agent)\,\overline{out}(mapchannel, collect)$
$.\,mapchannel(agent).\,OFFICE(out, mapchannel, collect)$

$EMPLOYER(out, out_{AH}, out_A) =$
$\nu(channel, service)\; out(channel, service)$
$.\,\overline{out_{AH}}(channel)$
$.\,\overline{out_A}(channel, service)$
$.\,EMPLOYER(out, out_{AH}, out_A)$

$MONITOR(out_{AH}, ack) =$
$\nu x\; out_{AH}(x)\,.\,\overline{x}(MOBILEAGENT)\,.\,ack$
$.\,MONITOR(out_{AH}, ack)$

$MOBILEAGENT(out_A, service, ack) =$
$\nu(ch, serv)\; out_A(ch, serv)\,.\,\overline{ack}$
$.\,MOBILEAGENT(out_A, service, ack)$

We specified as follows:

1. $ROUTE$: describes our whole system with all the components.
2. $OFFICE$: represents service-hosts, which make available different services (jobs) and notify their availability to the $EMPLOYER$ service.
3. $MOBILEAGENT$: represents our mobile agents that are able to migrate to an $OFFICE$ in order to

apply a job. The job is: $collect$ a database information and add it to its ”bag”.

4. $EMPLOYER$ plays a kind of reference book of all the tasks or jobs which are available on the local network.

5. $MONITOR$ is the waiting platform for the mobile agents.

We use gates like out, out_A, out_{AH} to specify the communication channel between the agents, employers, offices and monitor. In this specification the mobility is described by the channel parameters.

3. IMPLEMENTATION PART

The specification and validation platform architecture is functionalities directed. The Figure 3. below emphasizes not only the functionalities but also the dependencies of data. The blocks represent the functional entities, whereas the ochre forms are strategic data.

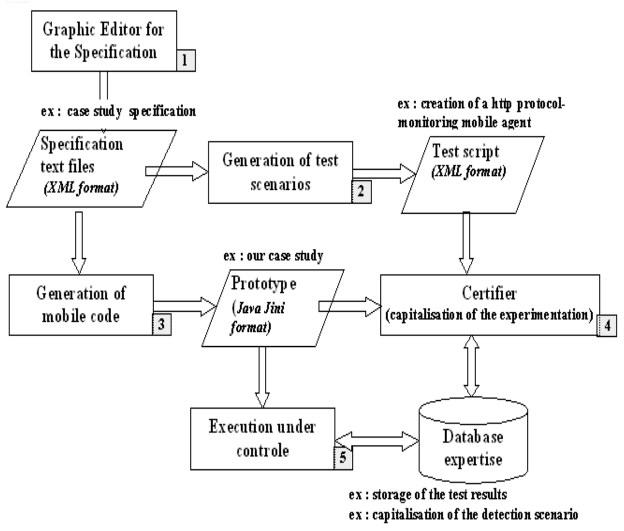
Graphic Editor for the Specification
1
ex : case study specification
Specification text files (XML format)
Generation of test scenarios
2
ex : creation of a http protocol-monitoring mobile agent
Test script (XML format)
ex : our case study
Generation of mobile code
3
Prototype (Java Jini format)
Certifier (capitalisation of the experimentation)
4
Execution under controle
5
Database expertise
ex : storage of the test results
ex : capitalisation of the detection scenario

Figure 3. from Specification to Implementation

Our $HoPiTool$ consist theoretically of five parts:

- The graphic editor for the specification.
  The objective is to represent by respecting the Unified Modelling Language (UML) notation, a specification where mobility is an essential characteristic. We propose five visualization windows for our editor:
  - an XML visualization of our specification, where the XML file will be validate by the given HOπ DTD
  - a π-calculus visualization
  - an HTML visualization in order to be able to show the documentation of each mobile agent
  - an UML visualization like collaboration diagrams.
  - a Java/Jini source code visualization

  From these charts, files of textual description are generated with the XML format (under control that the diagrams are sufficiently complete). The internationalization(we use the given methods by Java) of the graphic interface aspects is an essential constraint. To developed global software, one of the great commandments of internationalization is to separate text, labels, messages, and other locale-sensitive objects from the core source code. This helps to maintain a single source code base for all language versions of our product. It also facilitates translations because all localizable resources are identified and isolated.
  The major idea is to have a rigorous framework to control two important stages: the generation of test scenario and the generation of code.

- The generation of test scenarios
  From XML files describing a π-calculus specification, which means an ensemble of mobile agents communicating through the services that they specified, the goal of this functionality is to generate test scenarios respecting the criterions given by the specification.
  These criterions are compared to the specification not operational, but denotational. The objective is to build tests which are used by the Certifier entity. Also, this test-set must answer to constraints of coherence, that means to satisfy a common goal such as: ”execute all the sequences”. In other words our first goal is to build structural ”white box” and not functional tests called more commonly ”black box”.
  Each output file describes a script of tests and also the essential measurement points for the work executed by the Certifier.

- The generation of the mobile code
  This functionality is initiated by the graphic interface and it allows to obtain a code prototype from valid HOπ specifications. From XML files describing a π-calculus specification a whole of source files is generated in order to obtain a prototype. The aim is to gain ability to do simulations and prototype validations through the tests which we defined before.

More, HTML documents describing the documentation are generated by our tool, which takes in consideration the comments written within the specification. A specification consists of agents declarations. The whole of agents can be divided into two categories: mobile and non mobile. This constraint can be detected when an agent is send over a communication channel to an another agent. The mobile aspects take shape through the use of a technical framework called Java Intelligent Network Interface (Jini) from Sun. The fundamental element is the utilization of Remote Method Invocation (RMI), which allows us to consider the network like object oriented. The services and clients in a Jini architecture can be integrated easier within the object interface. Jini disposes also of identifications and collect services, that is a import future when we want to develop a mobile agent system. The security questions will be treat through Java Cryptography Architecture (JCA) Norm from Sun.

- The certifier
  This entity takes a Java prototype as entry but also a whole of test-scripts. Every test is apply over the prototype in order to determine if the prototype validates this test. We can distinguish two cases: the prototype satisfies the test and this simulation will enrich the data base or the prototype doesn't satisfy the test, an anomaly is lifted and a log file is saved. The log files are used in a "post-mortem" analysis phase, which is the basis of the Software Validation and Verification Report (SVVR). The analysis is not a part of our project.

- Execution under control
  Every execution of a generated prototype is realized under control. Every anomaly compared with the acquired experience is done through a call-back alert from the sites where the conflict was detected.

Our more important technical choices are the Java 1.4.2 Swing Framework, input, output in XML (with external validation in form of a DTD file), Java Intelligent Network Interface Jini 1.2., Remote Method Invocation (RMI), Oracle 9i. We use Extensible Markup Language (XML) to define and describe the higher order specification of our mobile agent. Our $HOPiTool$ reads and generates Jini code from the XML file.
In our tool the XML file is read and parsed by a Simple API for XML (SAX) parser and an instance of $AgentFactory$ class is created for each input XML file. A Document Type Definition (DTD) document describes how these XML files should be structured and the agent factory needs to have some knowledge of this DTD in order to know how to handle. Our $AgentFactory$ corresponds therefore to a Java code generator from an XML input file, which formally describes a π-calculus language specification.
We decided to use SAX instead of Document Object Model (DOM) because SAX requires little memory, it does not construct an internal representation (tree structure) of the XML data. Instead, it simply sends data to the application as it is read.
We defined a basic Document Type Definition (DTD) that describes the higher order π-calculus syntax and it is used to validate the XML file of our case study. This DTD states that an agent XML file contains a series of agents which play together in a mobile system. Each agent has a name, a number of arguments and a definition of his form and his objective. Based on the DTD we edit an XML file, which defines our case study, the world system, introduced before. We have defined each agent with the correspondents attributes and arguments. Based on this XML description of our mobile system and with help of our $HOPiTool$ we are able to generate Jini code for a further communication of the agents in a network.
We use intensively design pattern such that: $AbstractFactory$ pattern, $Composite$ pattern, $Template$ pattern and so on. This way of coding is an insurance for a better productively and maintainable code. We have defined four packages: $hopitool.gui$, $hopitool.lang$, $hopitool.parsing$ and $hopitool.unification$. The $hopitool.lang$ package provides an interface that describes the requirements for all the construction of Higher Order π-calculus language. Several classes like: $Call$, $Sequence$, $Parallel$, $Send$, $Receive$, $Choice$, $Restriction$, $Zero$ represent the instruction set in HO-π-Calculus and the communication gates are represented by the class $Channel$. The $PiCalculusHandler$ class contains the semantics of the Higher Order π-calculus language. The communication and the call of agents use parameters and expressions. The semantics of this data transfer is based on an unification algorithm [4]. In the current version of our tool we use a first order unification algorithm because all the agent names are considered as bound variables.
The second package $hopitool.unification$ contains several classes for a $Template$ design pattern or the unification concept. The $AbstractTerm$ class represents the root class of a $Composite$ design pattern. This pattern is applied for modelling a higher order π-calculus term. Its structure is always finite but its depth and its arity are not constant. We implement a finite first order unification to check the association between two terms. This operation is very useful for a $Call$ statement and also for every communication, it means for $Send$ and $Receive$ instructions.
The $Template$ design pattern is also used because it

allows us to delay a more complex algorithm for a higher order unification. This kind of concept is essential when an agent asks for a unknown service, but where its signature is already known by this agent. That algorithm is under development because, some constraints have to be checked before to compute this algorithm.
Further the class $Term$ represents a non terminal node of a term in an $AbstractTerm$ instances. It is used to model all functions or agents in an Higher Order π-calculus term. It is used for the unification step of terms. The class $Variable$ represents a general variable in a specification. Instances of that class are built during the analysis of the input XML file. It concerns : all the terms which appear in a Higher Order π-calculus expression and all the Higher Order π-calculus parameters of local declared names. It is used for the unification step of terms.
The class diagram for the creation of all the objects are given below. The $Factory$ classes are responsible for the creation of this classes, and the behavior is given by the class itself.
The $hopitool.parsing$ package contains all the classes used for the analysis and the management of the symbol table. Some classes are used to a validating check. This classes represents the data structure of some lines of the parsing table. It contains all the data which describe this information in the specification language.
The $hopitool.gui$ package contains all the viewers which belong to the graphical part of the tools. The graphical user interface (called GUI) has several features: each window created with $HOPiTool$ has the five tabs presented in the first part of the ”Implementation Part”. Some dialog boxes are used for diagnostics such as a dialog for each previous tab, a help manual for developer, a start guide for the specifier. All entities are internationalized and some external text property files are already prepared for English, French, German and Spanish. This package is divided into several sub packages which corresponds to the views. Also, it is easier to add another one. In the current version, the GUI is used to display some information about the specification. When the user click on a tab the update of the corresponding view is computed. A given specification is observable only in a $HOPiTool$ window. In the next version each view will support interactions from the user. All modifications have to be propagated on to the other views.

4. CONCLUSION

This work underlines the connection between the constraints of specification of higher order π-calculus and the mobile generation of code for the communicating systems. Our approach is validated by the construction of a prototype respecting the same constraints as in the specification. However, the $HOPiTool$ represented here, is only a part of our implementation an only the ”GUI”, ”generation of mobile code” modules are yet implemented. The generated Jini code is in our case study the generation of for java classes: $OFFICE$ class, $EMPLOYER$ class, a $MOBILEAGENT$ class and a $MONITOR$ class. These classes can communicate with each other in a network. An example for a framework for agent-based system in Jini is given by S. Li in [3] and it is called $Paradigma$. With help of this framework we can implement our Jini environment.
Also we are working on a test generator, which creates test sequences from initial π-calculus specification. Our intention is to obtain a test set, which will validate the generated code from $HOPiTool$. The test generator is configured with several criterions, which are essential to stop recursive unfolding in the agent definition.
The validation engine will be the final step of our study: it is an observer of the effects of the tests on the Jini prototype, generated by $HOPiTool$. Our intention is therefore to create an environment that allows the implementation of a mobile system of agents in Jini proceeding from a correct specification of the system given by the higher order π-calculus specification.

5. References

[1] Arnold, K.; A. Wollrath, B. O’Sullivan, R. Scheifler and J. Waldo. 1999 ”The Jini Specification”, *Addison-Wesley*.

[2] Barbu, A. and F. Mourlin. 2003 ”Higher Order π-Calcul Specification for a Mobile Agent in Jini”. In *W. Dosch, editor, 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD’03)*. pages 250-256, ACIS.

[3] Li, S. and al. 2000 ”Professional Jini” published by *WROX Press Inc*. ISBN 1861003552

[4] Lloyd, J. W. 1987 ”Foundations of logic programmic”. *Symbolic Computation - Artificial Intelligence*. Springer Verlag.

[5] Milner, R. 1999 ”Communicating and Mobile Systems: the π-Calculus”. *Cambridge Univerity Press*.

[6] Milner, R.; J. Parrow and D. Walker. 1993 ”A calculus of mobile processes, part I/II”. *Journal of Information and Computation*, 100:1-77.

[7] Sangiorgi, D. 1992 ”From π-Calculus to Higher-Order π-Calculus – and back”, In Proc. *TAPSOFT ’93.*, LNCS 668, Springer Verlag.