Mobile properties and Temporal Logic

Bardu, A. and Mourlin, F.

Technical Report TR-2002-06

Laboratory of Algorithms, Complexity and Logic
University of Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
F-94010 CRÉTEIL Cedex – FRANCE
Tel: +33 (0)1 45 17 16 47
Fax: +33 (0)1 45 17 66 01
Mobile Properties and Temporal Logic

Andreea Barbu and Fabrice Mourlin

Department of Computing Science LACL, Paris12, Val de Marne University, Créteil, France
email: barbu@univ-paris12.fr, Fabrice.Mourlin@wanadoo.fr

Abstract

Our subject is the proof of properties of real communicating systems with mobile features. We use two different strategies to specify an example: SLP Protocol. Firstly, we described this system by the use of a process algebra called Higher Order π-calculus Language [2] and we build inference tree to validate our property. Secondly, we represent our system with a toolbox called UPPAAL [6] and we check our property with the help of a model checker based on computation tree logic (CTL) [5]. Finally, we use a simulation module to validate our property by experimentation.

Keywords: temporal logic, mobility, process algebra, π-Calculus.

1 Introduction

Mobility is a key concept in distributed computing and concurrent object-oriented languages. Since computers were connected by links, there was the idea to exploit these connections not only for exchange messages, but also to move entities. Starting from simple data, the mobility had an evolution that has led to move the execution control, the code and the execution environment. Mobile code is a code than can be transmitted across the network and executes on the other end; this term used to describe general-purpose executables that run in remote locations. There are two main approaches to represent mobility in process algebra. In the higher order (or process passing) paradigm, terms of the language (like processes) may be transmitted. In the first order (or name-passing) paradigm, mobility is achieved by allowing transmission of names. We are interested in mobile code and particularly in properties of mobile code. The first section of this work is dedicated to present a mobile protocol, the SLP protocol [1] and to give a textual describing of a property of this protocol. The second section presents a formal method to express mobility, the higher order π-Calculus [2] and we describe an application of the SLP protocol with help of the HOπ-Calculus. For proving the reachability of our property we construct an inference tree [3] that used just the rules we have stated for the operational semantics of the π-Calculus [4]. In
the next section we describe our property using the CTL (Computation Tree Logic [5]) and we check it with UPPAAL, [6] that is a tool box for modeling, simulation and validation of real-time systems. The last section gives a conclusion about this work concerning the comparison of the both approaches to describe and prove properties, and we look at the limits of each formal approaches.

2 Mobile feature of the SLP Protocol

An important area is mobile code for providing clients access to network services. By deploying mobile code for network service access, called service drivers, clients can download code for particular network services when needed, exactly as capsules and mobile code allow routers and other infrastructure elements to download code for processing protocols that were not previously encountered. Client application software interacts with the downloaded network service driver through a standardized programmable interface defined for the service.

The SLP [1] is an IETF standard track protocol that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks.

SLP can eliminate the need for user to know the technical features of network hosts. With the SLP, the user only needs to know the description of the service he is interested in. Based on this description, SLP is then able to return the URL of the desired service. SLP is a language independent protocol. Thus the protocol specification can be implemented in any language. The SLP infrastructure consist of tree types of agents:

1. UserAgent (UA) is a software entity that is looking for the location of one or more services,
2. ServiceAgent (SA) is a software entity that provides the location of one or more services,
3. DirectoryAgent (DA) is a software entity that acts as a centralized repository for service location information.

In order to be able to provide a framework for service location, SLP agents communicate with each other using eleven different types of messages. The dialog between agents is usually limited to very simple exchanges of request and reply messages.

- Service Request (SrvRqst)
  Message sent by UAs to SAs and DAs to request the location of a service.

- Service Reply (SrvRply)
  Message sent by SAs and DAs in reply to a SrvRqst. The SrvRply message contains the URL of the requested service.
• Service Registration (SrvReg)
  Message sent by SAs to DAs containing information about a service that is
  available.

• Service Deregister (SrvDeReg)
  Message sent by SAs to inform DAs that a service is no longer available.

• Service Acknowledge (SrvAck)
  A generic acknowledgment that is sent by DAs to SAs as a reply to SrvReg
  and SrvDeReg messages.

• Attribute Request (AttrRqs)
  Message sent by UAs to request the attributes of a service.

• Attribute Reply (AttrRply)
  Message sent by SAs and DAs in reply to a AttrRqs. The AttrRply contains
  the list of attributes that were requested.

• Service Type Request (SrvTypeRqs)
  Message sent by UAs to SAs and DAs requesting the types of services that
  are available.

• Service Type Reply (SrvTypeRply)
  Message by SAs and DAs in reply to a SrvTypeRqs. The SrvTypeRply
  contains a list of requested service types.

• DA Advertisement (DAAdvert)
  Message sent by DAs to let SAs and UAs know where they are.

• SA Advertisement (SAAdvert)
  Message sent by SAs to let UAs know where they are.

It is interesting to prove properties of applications to ensure that the developed
codes respect strictly their initial specifications. First we look at a property of this
protocol. Let a system be composed of an UserAgent(UA), a ServiceAgent(SA)
and a DirectoryAgent(DA). The UserAgent(UA) asks the DirectoryAgent for a
service called "print", and the DirectoryAgent must be able to deliver this service.
This is one of many properties of this system. We look just this because it is
basic and simple to understand. In order to prove properties of applications we
are interested in a formal specification language having the capacity to express
mobility: the HO\pi-Calculus.

3 Formal method for mobility

A mobile system is a system with a dynamically changing communication topol-
ogy. We are interested in an approach to represent mobility in process algebra.
That is the Higher Order paradigm, 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 [9] and later refined by Milner [8] with the addition of sorts and of communication of tuples (Polyadic $\pi$-Calculus). The $\pi$-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 $\pi$-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 $\pi$-Calculus (HO$\pi$) is an extension of the first order $\pi$-Calculus introduced by D.Sangiorgi [2]. This calculus enriches the $\pi$-Calculus with explicit higher order communications. In the HO$\pi$-Calculus not only names, but also processes and parameterized processes of arbitrarily high order, can be transmitted. The syntax of the HO$\pi$-Calculus is an extension of the syntax of the first order $\pi$-Calculus.

The syntax of the HO$\pi$-Calculus is an extension of the syntax of the $\pi$-Calculus [7]:

\[ P ::= \xi(K).P \mid x(U).P \mid P|Q \mid \tau.P \mid \text{new } x \ P \mid P + Q \mid 0 \]

where $K$ is an agent or name, and $U$ is a variable or name.

The extension of the operational semantics is given by Sangiori [7]:

**ComHO$_1$** :
\[
\frac{P \xrightarrow{\xi K} P', Q \xrightarrow{\xi(U)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{K/U\}}
\]

**ComHO$_2$** :
\[
\frac{P \xrightarrow{x(U)} P', Q \xrightarrow{\xi K} Q'}{P|Q \xrightarrow{\tau} P'|K/U\}Q'}
\]

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 $K$: $\text{Arity}_{A,P}(U) = \text{Arity}_{A,P}(K)$ with $A$ a set of actions $x, y, ...$; and $P$ a set of agents $P, Q, ...$ This restriction is necessary to be enrich with a type control over the signature of $U$ and $K$. The adding of the type control [14] allow to use the overloading into the agent definition.

Our example is based on the publication of a simple print service. It uses just a single parameter. Also we do not use (AttrRqst), (AttrRply),(SrvTypeRply), (SrvTypeRqst), (DAAdvert), (SAAdvert), (UAAdvert), because the specification will be to difficult to understand. We describe a SLP system by using the HO$\pi$-Calculus, and we consider an UserAgent (UA) that asks the DirectoryAgent (DA) for a service, called "print". A ServiceAgent (SA) can
register and deregister services by the DirectoryAgent (DA) which must send the requested available service to the UserAgent if it is available. The other exchanges of communication between the SLP agents, will be ignored by our system to simplify matters. The $DA_{Mem}$ and the $IdleDA_{Mem}$ agents are used as a memory for the DirectoryAgent (DA). The DA saves information by the $DA_{Mem}$ and the $DA_{Mem}$ by the $IdleDA_{Mem}$. The $IdleDA_{Mem}$ is necessary for the deregister of the services.

We index the names of the messages with the names of the sends agents, in order to indicate that is possible to generalize the system, for instance we can look the behavior of a system with several UAs, DAs and SAs. Figure 1 gives an overview over the messages in a very simple system that contains an UserAgent, a ServiceAgent, a DirectoryAgent and a Memory and a IdleMemory for the DirectoryAgent.

![Diagram of agent interactions](image_url)

**Figure 1:** Messages between the agents in a very simple system.

**System** =

$$UA_1(Rqst_{UA_i})|SA_1(Reg_{SA_i}, DeReg_{SA_i}, Ack_{DA_i})|DA_1(Reg_{SA_i}, DeReg_{SA_i}, Rqst_{UA_i}, Ack_{DA_i})|DA_{Mem}(in, out)|IdleDA_{Mem}(out, in)$$
\[ UA_1(SrvRqst_{UA_1}) = \\
(new \ SrvRply_{UA_1}) \overline{SrvRqst_{UA_1}}(Service(print, msg), SrvRply_{UA_1}) \\
.SrvRply(S(name, f)).UA_1(SrvRqst_{UA_1}) \]

\[ SA_1(SrvReg_{SA_1}, SrvDeReg_{SA_1}, SrvAck_{DA_1}) = \\
SrvReg_{SA_1}(Service(print, f)).SrvAck_{DA_1} \\
.SrvDeReg_{SA_1}(Service(print, f)).SrvAck_{DA_1} \\
.SA_1(SrvReg_{SA_1}, SrvDeReg_{SA_1}, SrvAck_{DA_1}) \]

\[ DA_1(SrvReg_{SA_1}, SrvDeReg_{SA_1}, SrvRqst_{UA_1}, SrvAck_{DA_1}) = \\
[(SrvReg_{SA_1}(S(name, f)).input(S(name, f)).SrvAck_{DA_1}) \\
+ (SrvDeReg_{SA_1}(S(name, f)).reset. SrvAck_{DA_1}) \\
| SrvRqst_{UA_1}(S(name, f), SrvRply_{UA_1}). \overline{name}(SrvRply_{UA_1})] \\
.DA_1(SrvReg_{SA_1}, SrvDeReg_{SA_1}, SrvRqst_{UA_1}, SrvAck_{DA_1}) \]

\[ DA_{Mem}(input, inputIdle) = \\
input(S(name, f)).\overline{name}(SrvRply_{UA_1}). \overline{SrvRply_{UA_1}}(S(name, f)) + reset] \\
\cdot [(inputIdle(S(name, f)). IdleDA_{Mem}(input, inputIdle)) \]

\[ IdleDA_{Mem}(input, inputIdle) = \\
inputIdle(S(name, f)). \overline{name}(SrvRply_{UA_1}). \overline{SrvRply_{UA_1}}(S(name, f)) + reset] \\
\cdot [(input(S(name, f)). DA_{Mem}(input, inputIdle)) \]

Formal methods may be used to specify and model the behavior of a system and to verify that the system design and implementation satisfy system functional and safety properties. With the notion of transition graphs and the operational semantics for operators in the HOπ-Calculus we can demonstrate the relation between them. The operational rules will allow us to prove or disprove the correctness of the arcs connecting agent expressions in any transition graphs we construct. We do this by constructing inference trees using just the rules we have stated for the operational semantics. At the root of each successful tree will be the transition we are trying to prove. Each node in the tree will consist of a transition labelled by the rule which was used to derive it. A node may only refer to transitions already proved correct higher up in the tree.

We are looking for a very simple system that contains an UA, a SA, a DA and a \( DA_{Mem} \). The UA asks the DA for a service "print". The SA can, in our example, only subscribe a service called "print" at the DA. The DA puts this service in its memory \( DA_{Mem} \) which delivers the UA with the requested service. We prove that the transition is achieved: "If UA asks for the service "print" and if SA subscribes a DA, then UA obtains the service "print"."
System =
\[ UA(Rqst_{UA}) | SA(Reg_{SA}, De_{Reg_{SA}}, Ack_{DA}) \]
\[ | DA(Reg_{SA}, Rqst_{UA}, Ack_{DA}) | DA_{Mem}(in, out) \]

\[ UA(SrvRqst) = \]
\[ (new \ SrvRply) \overline{SrvRqst}(Service(print, msg), SrvRply) \]
\[ . SrvRply(S(name, f)) . UA(SrvRqst) \]

\[ SA(SrvReg, SrvAck) = \]
\[ SrvReg(Service(print, f)) . SrvAck . SA(SrvReg, SrvDeReg, SrvAck) \]

\[ DA(SrvReg, SrvRqst, SrvAck) = \]
\[ [(SrvReg(S(name, f)).input(S(name, f)).\overline{SrvAck}) \]
\[ | SrvRqst(S(name, f), SrvRply).\overline{name(SrvRply)}]. DA(SrvReg, SrvRqst, SrvAck) \]

\[ DA_{Mem}(input) = \]
\[ input(S(name, f)) . name(SrvRply).\overline{SrvRply}(S(name, f)) . DA_{Mem}(input) \]

Using the Com\(HO_1\) et Com\(HO_2\) rules we can establish the correctness of our property. We consider: \(UA|SA|DA|DA_{Mem}\)

After the application of Com\(HO\) for the parallel communication between the channels \(SrvRqst(S(name, f), SrvRply)\) and \(\overline{SrvRqst}(Service(print, msg), SrvRply)\) and the channels: \(\overline{SrvReg(Service(print, f))}\) and \(SrvReg(S(name, f))\), what means that UA asks the DA for the service "print" and SA subscribes the service "print" at the DA, our system becomes:

\[ SrvRply(S(name, f)) . UA(SrvRqst) \]
\[ | SA(SrvReg, SrvAck) \]
\[ | input(Service(print, f)) | Service(SrvRply) | . DA(SrvReg, SrvRqst, SrvAck) \]
\[ | input(S(name, f)) . Service(SrvRply). SrvRply(Service(print, f)) . DA_{Mem}(input) \]

Using the Com\(HO\) for the parallel channels \(input(Service(print, f))\) and \(input(S(name, f))\) which means that DA puts this information (that the registered service is "print") in its memory \(DA_{Mem}\), we get the following transition:

\[ SrvRply(S(name, f)) . UA(SrvRqst) \]
\[ | SA(SrvReg, SrvAck)|DA(SrvReg, SrvRqst, SrvAck) \]
\[ | SrvRply(Service(print, f)) . DA_{Mem}(input) \]
We see that the channels $SrvRply(S(name, f))$ and $SrvRply(Service(print, f))$ can communicate, that means that UA receives the requested service "print" from the $DA_{Mem}$. After this transitions and the applications of the $ComHO$ the system becomes: $UA|SA|DA|DA$ with $S := service$ and $name := print$.

4 Temporal properties of SLP Protocol

Computation Tree Logic (CTL) is a branching time [10] temporal logic which provides correct behavior of parallel systems by expressing properties concerning occurrence of events in time. Different operators and modalities can be used to express important properties such as invariance, eventuality and precedence [11]. Given a correctness property, i.e. a temporal logic formula, there are two principal ways of using temporal logic. One is to apply an automated synthesis method using a decision procedure to determine the satisfiability of our property. When the method succeeds it generates a synchronization skeleton of the system events [12]. The second one, on which we will focus attention, uses a model checking based method which verifies the truth of the correctness property in the structure representing the parallel system. This structure is generally a labelled transition system.

We use the UPPAAL tool box [6] for proving the correctness of our property. UPPAAL, developed jointly by Uppsala University and Aalborg University, is a tool box for modeling, simulation and verification of real-time systems. A typical application area include the communication protocols, and this was one of the reason for choosing this tool. It is designed mainly to check invariant and reachability properties by exploring the state-space of a system. A system in UPPAAL is composed of concurrent processes, each of them modeled as an automaton. The automaton has a set of locations. Transitions are used to change location. To control how to fire these transitions, it is possible to have a guard and a synchronization. A guard is a condition on the variables saying when the transition is enabled. The synchronization mechanism in UPPAAL is a handshaking synchronization: two processes take a transition at the same time, one will have a $a!$ and the other a $a?$, a being a the synchronization channel. When taking a transition actions are possible: assignment of variables. We take a system that contains two UserAgents, two ServiceAgents (one that can register a "print" service and one that can register a "mail" service), two DirectoryAgents and the correspondingly $DA_{Memorys}$ and $DA_{IdleMemorys}$.

The UserAgent $UA_1$ asks the DirectoryAgent $DA_1$ for the service "print". The DirectoryAgent can send the service to the UserAgent only if the ServiceAgent $SA_1$ has previously subscribed the service "print" at the DirectoryAgent. The modeling of UserAgent is given by Figure 2, for the DirectoryAgent by Figure 3.
and for the memory of the DirectoryAgent by Figure 4.

Figure 2: Modeling of UserAgent with UPPAAL-Tool: process UA

Figure 3: Modeling of DirectoryAgent with UPPAAL: process DA

Figure 4: Modeling of MemoryDirectoryAgent with UPPAAL: process DAMem
We describe the system and then verify that our property is satisfied. We describe the property with help of CTL:

- The states before the UA receives the service "print":
  \[ P_1 = E <> (\text{UA.wait-print and DA.Rqst-print and DAMem.save-print}) \]

- The states after the receiving of the service "print":
  \[ P_0 = E <> (\text{UA.wait-use and DA.startDA and DAMem.save-Memprint}) \]

\( P_1 \) means that there exists a path where (UA.wait-print and DA.Rqst-print and DAMem.save-sprint) holds in the next state, and \( P_0 \) means that there exists a path where (UA.wait-use and DA.startDA and DAMem.save-Memprint) holds in the next state. UA.wait-use, DA.startDA, DAMem.save-Memprint, UA.wait-print, DA.Rqst-print and DAMem.save-sprint represent names of states of our agents. The figure 5 shows the print screen of the validation for our property by the UPPAAL tool box. The two properties are satisfied.

![Established direct connection to local server.](image)

Figure 5: Status of the property.

5 Conclusion

This work explains how to specify a real protocol with mobile features, called SLP and also how to validate a specific property. We encounter two main limits. The first one was with the use of π-Calculus language and its operational semantics. We have had to add two rules to a traditional semantics [2] and we built an inference tree for proving that a transition "print" was achieved. This approach is not automatic and it is essential to explore the whole algebraic term before to conclude that a specific state is achieved. This
kind of proof is too specific. We just looked into an evaluation of a global term and we cannot conclude anything about the other possible evaluations. Also, our inference tree describes one UserAgent, one ServiceAgent and one DirectoryAgent. But, if we change the initial configuration, this tree has to be built from scratch.

Thus we used a model checking technique and temporal logic to establish a more general property about all the evaluations of a term. We used UPPAAL tool to do that, but we encountered a second limit: a sequence of actions is described by an automaton and between two automata there is no exchange of data. There is only one possible operation: the synchronization. In that case, we were not able to express the mobile aspect of the "print" service.

Both approaches stress the lack of a temporal logic for higher order process specification. An extension of CTL seems to be a right approach. Some work exist on that subject [15], but higher order process are not taken into account and the mobility is only applied with names. Also, there is no tool and the proof are built manually.

A second solution consists of translating a higher order process specification into a first order description and to establish properties on a logical model (built for the previous description). But in that case, it is also necessary to extend CTL language to consider exchange of names.

A future work is to develop a tool which will allow us to obtain a prototype. This one will be written with JINI toolkit (Java Intelligent Network Infrastructure) [13], starting from the specification of the HOπ-calculus. Using the Jini specification we could preserve the notion of mobility in our system.

References


[2] D. Sangiorgi: From π-Calculus to Higher-Order π-Calculus – and back, work supported by the ESPRIT BRA project "CONFER".


