A Service-Oriented Language for Programming Mobile Agents

Hervé Paulino
Departamento de Informática,
Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa, Portugal
herve@di.fct.unl.pt

Luís Lopes
Departamento de Ciência de Computadores,
Faculdade de Ciências,
Universidade do Porto, Portugal
lblopes@dcc.fc.up.pt

ABSTRACT

In this paper we present MOB, a service-oriented scripting language for programming mobile agents in distributed systems. The main feature of the language is the integration of the service-oriented and the mobile agent paradigms. MOB is encoded onto a process calculus with a well studied semantics which provides us with a tool to prove the soundness of the language relative to the underlying calculus.

Keywords

Mobile Agents, Service-Oriented, Programming Language

1. INTRODUCTION AND MOTIVATION

Service-Oriented programming departs from the object-oriented paradigm by de-coupling data and processing. Services are provided in a transparent way for clients, requiring only knowledge of the contract (the service's interface). One of the main advantages of service-oriented programming is that it provides a framework on which to develop component-based systems. These have recently received a lot of attention for distributed systems, namely with the .NET [14], Jini [19] and Openwings [9] platforms.

Another major technology for Web applications is that of *mobile agents*. Mobile agents are computations that have the ability to travel through a network, by halting their execution, saving their state and then restoring it in a new host. In this paradigm, mobile agents move towards the resources (e.g., data, servers) and interact locally, unlike the usual communication paradigms (e.g., client-server), that require costly remote sessions to be maintained.

Programming languages for mobile agents come in two flavors: those *designed by hand* and those based on formal systems. In the first set we have systems such as Aglets [7], Mole [16] and Voyager [4] that mostly extend Java classes to define an agent's behavior, and scripting languages such as D'Agents [5] or Ara [13]. Providing a demonstrably sound semantics for these systems is rather difficult given the gap between the implementation and an adequate formal model.

Languages in the second set are based on formal systems, mostly some form or extension of the π-calculus [6, 8]. This allows researchers to build solid specifications for these programming languages and prove the languages correct *by design* by providing adequate encodings onto a base calculus. Examples of such languages have been implemented in recent years, namely, JoCaml [3], TyCO [18], X-Klaim [2], Nomadic Pict [20], Acute [10] and Alice [15].

In [12] we first introduced MOB, a programming language for developing applications based on mobile agents. Here we extend MOB with another main abstraction, *services*, thus uniting two major abstractions for Web applications. The main novelties introduced in this paper, in particular relative to [12], are as follows: a) services, described as interfaces implemented by agents, and (mobile) agents are the main abstractions of the language. Agents provide and require services and are bound to these dynamically as they move through the network; b) the MOB language has been encoded in TyCO a form of the π-calculus extended with distribution and mobility of resources and with a well-studied semantics [1,18]. This provides a tool to prove the soundness of the operational semantics of the language, thus providing a form of language security not readily available in the other languages discussed; c) the encoding onto the process calculus provided a full specification of the front-end of the compiler for MOB and allowed us to use the TyCO run-time system to run compiled MOB programs.

2. THE LANGUAGE

One of the main abstractions of MOB is the **agent**. Agents may be viewed as special objects, with a run-time associated, that provide/require services to/from the network (other agents). Agents are handled in MOB programs much like objects are handled in object-oriented languages.

The **service** is another main abstraction. An agent **implements** services and, simultaneously, **requires** services provided by other agents. There is absolutely no distinction between clients and servers. To access a service, a programmer must get a binding for an agent that provides it. The binding is obtained dynamically through the primitive **bind** that asks a network name service for an agent that provides the service. When the binding is received, interaction through method invocation can happen.

Agents may move through the network and this is controlled explicitly, at high-level, by the programmer using a primitive **go**. The movement of an agent involves moving an entire virtual machine and its state to the target host in the network. The execution resumes on arrival at the target host, without intervention from other agents.

Agents are multi-threaded. Thread creation and synchronization is supported through the **fork** and **join** primitives. Another form of synchronization, on data-structures, is provided by the instructions **lock** and **unlock** that support a simple form of mutual exclusion in data access.

Interaction with external services is supported with **exec** which is used to implement extensions to the core MOB lan-

$$
\begin{array}{rcl}
P & ::= & \textsf{agent}\ X(\tilde{x})\ \textsf{implements}\ \tilde{S}\ \textsf{requires}\ \tilde{S}\ M \\
 & \mid & \textsf{service}\ S\ \{\tilde{m}\} \ \mid\ \textsf{requires}\ \tilde{S} \ \mid\ \textsf{class}\ X(\tilde{x})\ M \\
 & \mid & \textsf{join}\ (x) \ \mid\ \textsf{go}\ (e) \ \mid\ \textsf{lock}\ (x) \ \mid\ \textsf{unlock}\ (x) \\
 & \mid & \textsf{if}\ (e)\ \{\ P\ \}\ \textsf{else}\ \{\ P\ \} \ \mid\ \textsf{while}\ (e)\ \{\ P\ \} \\
 & \mid & \textsf{for}\ (x\ \textsf{in}\ e)\ \{P\} \ \mid\ \textsf{break} \ \mid\ \textsf{return}\ (e) \\
 & \mid & \textsf{exit}() \ \mid\ x = e \ \mid\ P\ ;\ P \\
M & ::= & \{m_1(\tilde{x}_1)\ \{P_1\}\ \ldots m_n(\tilde{x}_n)\ \{P_n\}\} \\
e & ::= & e\ bop\ e \ \mid\ uop\ e \ \mid\ \textsf{self} \ \mid\ x \ \mid\ e.x \\
 & \mid & c \ \mid\ \textsf{null} \ \mid\ C \\
C & ::= & \textsf{new}\ X(\tilde{e}) \ \mid\ \textsf{fork}\ \{\ P\ \} \ \mid\ \textsf{bind}\ (S) \ \mid\ \textsf{host}() \\
 & \mid & \textsf{exec}\ (\tilde{e}) \ \mid\ C.m\ (\tilde{e}) \ \mid\ \{\tilde{e} : \tilde{e}\} \ \mid\ [\tilde{e}]
\end{array}
$$

Figure 1: Syntax of the Mob programming language.

guage to support external functionality. These external services may be implemented in other languages, such as Java, C, TCL or Perl, or access network services, such as WWW queries, FTP transactions, or e-mail communication.

The remainder of the language constructs provide fairly standard support for control flow, expressions and external calls. Due to space constraints we will briefly describe the syntax and semantics of the language. The full definition of the language and its operational semantics may be found in [11]. The syntax for a Mob program is presented in figure 1. The language defines a set reserved words identified in bold-face. The main syntactic categories are: constants (booleans, integers and strings) ranged over by c; variables, ranged over by x; agent and class identifiers, ranged over by X; service identifiers, ranged over by S; method names and method collections, ranged over by m and M, respectively; expressions, ranged over by e; commands, ranged over by C; arrays and maps, ranged over by $\{\tilde{e} : \tilde{e}\}$ and $[\tilde{e}]$, respectively, and; instructions, ranged over by P. A sequence of elements of a given grammatical category α is denoted by $\tilde{\alpha}$. The concrete syntax of Mob imposes some syntactic restrictions to the above.

We exemplify the syntax with a small example of a server and a mobile client for a simplified Time service. The server provides the service Time with a single method getTime(). Note that the **main** method may be empty since Mob agents run as daemons and some external action is required to terminate their execution.

```
service Time { getTime }
agent TimeServer() implements Time {
   main () { }
   getTime() { return exec ("getTimeApplication");}
}
new TimeServer();
```

The client requires this service and when run, it takes a sequence of hosts, and performs a cycle in which it moves to each of them, setting their time accordingly with the central time from the TimeServer.

```
agent TimeClient(hostList) requires Time {
   main () {
      timeServer = bind (Time);
      for (h in hostList) {
         go (h);
         exec ("setTimeApplication", "" ^timeServer.getTime());
      }
}
new TimeClient([host1,...,hostn]);
```

The semantics for the Mob core language is provided in the form of an abstract machine that describes state transitions between network configurations. Each network configuration is composed of a set of agents running concurrently plus a name service for binding services to agents. Each agent has its own heap, code, a set of threads running concurrently and a set of threads suspended on resources. Besides their own local environment, threads share the agent's heap and use it to synchronize their access to resources.

For a complete formal specification of the semantics of the language we refer the reader to [11].

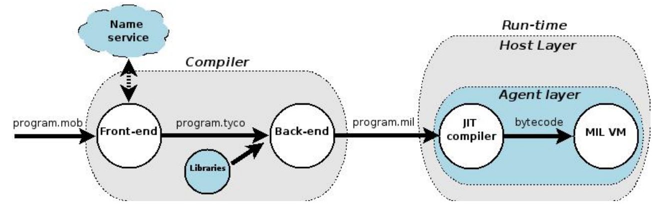

Figure 2: Compiling and running a Mob program.

3. THE COMPILER AND RUN-TIME

The compiler is divided in two main stages (figure 2). First, the *front-end* of the compiler takes the Mob source code and outputs the corresponding code in the TyCO language, as specified by the encoding of Mob in TyCO. The front-end of the compiler performs type inference on the Mob source code and in particular finds the types for both the implemented and the required services for agents. At this point the name service is contacted and a type-check is performed. The types inferred by the compiler are matched with those assumed for the service in the network. If the agent implements a service yet not registered, the interface provided becomes the *de facto* interface for that service. If type-checking succeeds, the source Mob program is transformed into a program written in the TyCO language.

The *back-end* of the Mob compiler is just the compiler for the TyCO language. It takes the TyCO code generated by the front-end and produces code written in an intermediate language called MIL (Multi-threaded Intermediate Language [17]), which is compiled *just-in-time* by the run-time system before being executed. This scheme allows the run-time to check type information in the MIL code before running it.

The run-time system is divided into two layers. The top layer is the *host layer* and is implemented as a service that must run on every Mob-enabled host in a network. This layer is responsible for managing agents within a host and supporting their mobility. More specifically, it provides the means to create, execute, marshall, send, unmarshall and receive agents.

The second layer is the *agent layer* and implements agents and the name service (also an agent). All are executed with the TyCO run-time system. Each agent is implemented with two concurrent threads: the first is an instance of the MIL virtual machine that runs the MIL code for the agent; the second handles network communication for the MIL virtual machine.

4. RELATED WORK

Of the languages more closely related to Mob none adheres to the service-oriented paradigm.

JoCaml [3] is a programming language based on the Join-calculus that provides support for distributed and mobile agent based applications. The language uses a custom virtual machine and supports the migration of trees of compu-

tations (an agent and its tree of sub-agents) between hosts in a network. Just as in Mob, type-checking is mostly done at compile time except for interaction with other modules which is done dynamically. The required type information is annotated in the source program.

The X-Klaim [2] language is an implementation of the Klaim model with ad-hoc extensions to incorporate higher-order constructs, asynchronous reading of tuple-spaces and hierarchical structured networks. Programs in X-Klaim are compiled into Java classes that resort to a package, Klava, to run. Mobile agents in X-Klaim are processes with a single execution flow, rather than the multi-threaded agents found in Mob. This makes the migration of multi-threaded agents a complex and user aware operation.

Nomadic Pict [20] is perhaps the closest to ours in that it grows from another process calculus based language, Pict, and adding primitives for programming mobile computations such as agent creation, agent migration and asynchronous communication between agents. Nomadic Pict also focuses on verification and a proof of correctness for an instance of the infrastructure has been achieved. Despite the similarities our aim is to provide a scripting language that abstracts away from network location dependent information. In this respect we feel that Mob, even in its core language is higher level than Nomadic-Pict.

Acute [10] is a programming language for mobile agents built on top of Objective Caml. The language provides type-safety through a partially static/dynamic type checking scheme. Moving computations is achieved through an atomic operation that captures a collection of threads in a structure (a *thunk*) that can afterwards be marshalled and moved across the network. This contrasts to Mob where marshalling of objects or agents is transparent to the programmer. In the case of agents the primitive *go* implements the marshalling required for sending the agent to another node in the network. The Mob service on that node will be responsible for unmarshalling the agent and restart it. In this respect Acute provides a finer, lower level, control over migration and marshalling than Mob.

Alice is a programming language based on Oz and its implementation, Mozart [15] (itself based on Standard ML). The functionality provided by Alice is similar to that of Acute. The front-end of the Alice compiler produces Oz intermediate code so that the Oz run-time, Mozart, is used to run Alice applications. The approach is similar to the one we use since the front-end of the Mob compiler produces TyCO code that is then executed with TyCO's run-time system.

5. CONCLUSIONS

We have produced an implementation of the Mob language compiler and its run-time system. Currently we are working to prove the soundness of the Mob language. In other words, given an encoding map $[\![\cdot]\!]$, from Mob abstract-machine states into TyCO programs, we wish to prove the following conjecture:

Conjecture (Soundness) Let N and N' be network configurations in the Mob abstract-machine. If $N \rightarrow N'$ (*reduces to* in Mob) then, $[\![N]\!] \equiv [\![N']\!]$ (*is congruent to* in TyCO) or $[\![N]\!] \rightarrow [\![N']\!]$ (*reduces to* in TyCO).

6. REFERENCES

[1] A. Ravara et all. Lexically Scoping Distribution: What You See Is What You Get. In *F. of Global Computing*, volume 85(1) of *ENTCS*. Elsevier Science, 2003.

[2] L. Bettini, R. D. Nicola, and R. Pugliese. X-Klaim and Klava: Programming Mobile Code. In *TOSCA 2001*, volume 62. Elsevier Science, 2001.

[3] S. Conchon and F. L. Fessant. Jocaml: Mobile Agents for Objective-Caml. In *ASA/MA'99*, pages 22–29. IEEE Computer Society, 1999.

[4] G. Glass. Overview of Voyager: ObjectSpace's Product Family for State-of-the-art Distributed Computing. Technical report, CTO ObjectSpace, 1999.

[5] R. S. Gray. Agent Tcl: A Transportable Agent System. In *CIKM'95 Workshop on Intelligent Information Agents*, 1995.

[6] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *ECOOP'91*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[7] D. B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.

[8] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[9] Openwings. Openwings: A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems. `http://www.openwings.org`, 2001.

[10] P. Sewell et all. Acute: High-Level Programming Language Design for Distributed Computation. `http://www.cl.cam.ac.uk/users/pes20/acute/`.

[11] H. Paulino and L. Lopes. Mob Core Language and Virtual Machine. Technical Report DCC-2005-05, DCC - FC and LIACC, Universidade do Porto, 2005. `http://www.dcc.fc.up.pt/Pubs/TR05/dcc-2005-05.pdf`.

[12] H. Paulino, L. Lopes, and F. Silva. Mob: A Scripting Language for Mobile Agents Based on a Process Calculus. In *ICWE 2003*, volume 2272 of *LNCS*, pages 40–43. Springer-Verlag, 2003.

[13] H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *MA'97*, volume 1219 of *LNCS*, pages 316–323. Springer-Verlag, 1997.

[14] D. Platt. *Introducing Microsoft .NET, Third Edition*. Microsoft Press, 2003.

[15] G. Smolka. Concurrent Constraint Programming Based on Functional Programming. In *Programming Languages and Systems*, volume 1381 of *LNCS*, pages 1–11. Springer-Verlag, 1998.

[16] F. Straber and J. Baumann. Mole - A Java Based Mobile Agent System. In M. M., editor, *Special Issues in Object Oriented Programming*, pages 301–308, 1997.

[17] V. Vasconcelos and L. Lopes. A Multithreaded Assembly Language and Virtual Machine. Unpublished.

[18] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *HLCL'98*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[19] J. Waldo. The Jini Architecture for Network-Centric Computing. *Commun. ACM*, 42(7):76–82, 1999.

[20] P. T. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile

Agents. *IEEE Concurrency*, 8(2):42–52, 2000.