Mob: a Scripting Language for Programming Web Agents

Hervé Paulino[1], Luís Lopes[2], and Fernando Silva[2]

[1] Department of Informatics. Faculty of Sciences and Technology. New University of Lisbon.
email: herve@di.fct.unl.pt
[2] Department of Computer Science. Faculty of Science. University of Oporto.
email: {lblopes,fds}@ncc.up.pt

Abstract. Mobile agents are the latest software technology to program flexible and efficient distributed applications, since they are independent programs that travel over the network, focusing on local communication, rather than the usual communication paradigms. Most current systems implement semantics that are hard if not impossible to prove correct. In this paper we present Mob, a scripting language for web agents encoded on top of a process calculus and with provably sound semantics that allows interaction with programs written in many programming languages.

Keywords: mobile agents, Internet computing, distributed systems, process calculus

1 Introduction and Motivation

The introduction of the π-calculus [7, 13] and other related process calculi, in the early nineties, as a model for concurrent distributed systems, provided the theoretical framework upon which researchers could build solid specifications. The main abstractions in these calculi are processes, representing arbitrary computations and channels, representing places where processes synchronize and exchange data. Recent extensions of these models introduced another fundamental abstraction, sites, which denote places in a network where processes run. These extensions allowed, for the first time, the modeling of complex distributed systems with mobile resources [2, 6, 10, 12, 17].

The mobile resources supported by these recent extensions are a powerful abstraction for the development of mobile agent frameworks. Mobile agents add to regular agents the capability of traveling to multiple locations in the network, by saving their state and restoring it in the new host. As they travel, they work on behalf of the user, such as collecting information or delivering requests. This mobility greatly enhances the productivity of each computing element in the network and creates a powerful computing environment, focusing on local interaction. Thus, our mobile agents are independent programs that travel over the network, focusing in local communication, rather than the usual communication paradigms (e.g., client-server).

In this paper we present a scripting language, Mob, for programming mobile agents in distributed environments. The semantics of the language is based on the DiTyCO (Distributed TYped Concurrent Objects) process calculus [17]. The run-time for the language is provided by the current implementations of the DiTyCO [11]. In particular, we rely on it for interprocess communication and code migration over the network.

The development of mobile agents requires a software infrastructure that provides migration and communication facilities, among others. Current frameworks that support mobile agents are mostly implemented by defining a set of Java classes that must then be extended to implement a given agent behavior, such as Aglets [3], Mole [8], or Klava [9]. MOB, on the other hand, is a simple scripting language that allows the definition of mobile agents and their interaction, an approach similar to D'Agents [15]. However, MOB applications may interact with external services programmed in other languages than MOB. Furthermore, the language is compiled into a process-calculus based kernel-language, and its semantics can be formally proved correct relative to the base calculus. Therefore, in this sense, MOB features some language security. Correctness or type-safety results are difficult to produce for most of the current systems. For example, only a subset of Java programs can be proved to be correct and type-safe [14].

The interaction between MOB applications and external services, provided by MOB enabled hosts, can be programmed in many languages such as Java, C, TCL or Perl. The philosophy is similar to that used in the MIME type recognition. The runtime engine matches a file type against a set of internally known types and either launches the corresponding application or simply executes the code.

The remainder of the paper is structured as follows: section 2 describes the target language, TyCO, in which MOB is encoded; section 3 describes the MOB programming language; section 4 provides some MOB programming examples; section 5 describes the compilation of a MOB program in TyCO; and finally section 6 describes the on-going research and future work.

2 The Target Language - DiTyCO

Our target language is based on a process calculus, in the line of the asynchronous π-calculus, named DiTyCO [17]. The main abstractions of the (centralized) calculus are channels (communication endpoints), objects (collections of methods that wait for incoming messages at channels) and asynchronous messages (method invocations targeted to channels). It is also possible to define process definitions, parameterized on a set of variables, that may be instantiated anywhere in the program (this allows for unbounded behavior). The abstract syntax for the core language is the following:

P	::=	$\mathbf{0}$	terminated process
	|	$P \mid P$	concurrent composition
	|	**new** x P	new local variable
	|	$x!l[\tilde{v}]$	asynchronous message
	|	$x?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	object
	|	**def** $X_1(\tilde{x}_1) = P_1 \ \ldots \ X_n(\tilde{x}_n) = P_n$ **in** P	definition
	|	$X[\tilde{v}]$	instantiation
	|	**if** v **then** P **else** Q	conditional execution

where x represents a variable, v a value (a variable or a channel), X a process definition and, l a method label.

From an operational point of view, centralized DiTyCO computations evolve for two reasons: object-message reduction (i.e., the execution of a method in an object in response to the reception of a message) and, instantiation of definitions. These actions can be described more precisely as follows:

$$x?\{\ldots, l(\tilde{x}) = P, \ldots\} \mid x!l[\tilde{v}] \rightarrow \{\tilde{v}/\tilde{x}\}P$$

The message $x!l[\tilde{v}]$ targeted to channel x, invokes the method l in an object $x?\{\ldots, l(\tilde{x}) = P, \ldots\}$ at channel x. The result is the body of the method, P, running with the parameters $\tilde{x}$ substituted by the arguments $\tilde{v}$. For instantiations we have something very similar.

$$\textbf{def } \ldots\ X(\tilde{x}) = P\ \ldots \textbf{ in } X[\tilde{v}] \mid Q \rightarrow \textbf{def } \ldots\ X(\tilde{x}) = P\ \ldots \textbf{ in } \{\tilde{v}/\tilde{x}\}P \mid Q$$

A new instance $X[\tilde{v}]$ of the definition X is created. The result is a new process with the same body, P, as the definition but with the parameters $\tilde{x}$ substituted for the arguments $\tilde{v}$ given in the instantiation.

This kernel language constitutes a kind of assembly language upon which higher level programming abstractions can be implemented as derived constructs.

The full, distributed, calculus grows from the centralized version by adding a new layer of abstraction representing a network of *locations*, identified by r, s, where processes run.

N	::=	$\mathbf{0}$	terminated network
	|	$N \parallel N$	concurrent composition
	|	**new** $x@s\ N$	new local variable
	|	**def** $D@s$ **in** N	definition
	|	$s[P]$	location with running process

This additional layer does not however introduce new reduction operations in the calculus. In fact, reduction can only be performed locally at locations, either by communications or instantiations as described above.

As can be observed from the above syntax, all resources are lexically bound to the locations they are created on. Thus, a message or object located at some channel $x@s$ must first move to location s to in order to reduce. Similarly, an instantiation of a definition $X@s$ must move to location s in order to reduce.

To preserve the lexical bindings of resources, every time one moves to another location, all its free identifiers (references for resources it uses) are translated on-the-fly. This is represented by a transformation σ_{rs} meaning "translation of identifiers when moving from location r to location s".

The lexical scope on resources together with the requirement of local reduction induce the following rules for resource migration:

Message Migration	$r[x@s!l[\tilde{v}]] \rightarrow s[x!l[\tilde{v}\sigma_{rs}]]$
Object Migration	$r[x@s?M] \rightarrow s[x?M\sigma_{rs}]$
Remote Instantiation	**def** $X@s(\tilde{x}) = P$ **in** $r[X@s[\tilde{v}]] \rightarrow$ **def** $X@s(\tilde{x}) = P$ **in** $s[X[\tilde{v}\sigma_{rs}]]$

One final word is required on: (a) lexical scope, and; (b) local reduction, since they are design goals for the language. Lexical scope is an important property since it provides the compiler and run-time system with important information on the origin of a resource. This is important namely for safety reasons (e.g., does the resource come from a trusted location ?) and for implementation reasons (e.g., where do we allocate the data-structures for it ? Do they move around in the network ?).

Local reduction is also of the utmost importance. Client-server interactions for example occur within a location, with much lower overheads than in the standard Client-Server model where interactions required maintaining remote sessions open and the exchange of many messages drastically reducing the available bandwidth of a network. In the novel paradigms for Web Computing [1], client applications move to server locations where they

interact with a local session. They return to their original location after the local session is complete.

The following programming example illustrates the use of these primitives and derived constructs. We use the (**let**/**in**) derived constructs, defined in [16], for synchronous method calls.

```
def Cell (self, value) =
    self ? {
        read (replyto) = replyto![value] | Cell [self, value]
        write (newValue) : Cell [self, newValue]
    }
in def
    IntegerCell(self, value) = Cell [self, value]
    StringCell(self, value) = Cell [self, value]
in
    new c IntegerCell[c, 4] | let i = c!read[] in io!printi[i]
```

The general Cell template stores a value and features read and write methods to retrieve or change its contents. Type specific templates, such as IntegerCell, based on the general Cell template will be used further on on the paper, to encode values in DiTyCO. In this example, an IntegerCell is created to store value 4 and the read method is used to retrieve this value in order to print it to the standard output. Notice that Cell is polymorphic on value.

3 The Source Language - Mob

The Mob programming language is a simple, easy-to-use, scripting language. Its main abstractions are *mobile agents* that can be grouped in *communicators* allowing hierarchies to be formed, group communication and synchronization. The abstract syntax for the kernel of the language is as follows:

$$
\begin{aligned}
\text{Program} &::= \text{AgentDef} \mid \text{AgentDef Program} \mid \text{InstructionList} \mid \text{InstructionList Program} \\
\text{AgentDef} &::= \textbf{agent}\ id\ \textbf{\{}\ \text{AttributeDef Init Do Iterators}\textbf{\}} \\
\text{AttributeDef} &::= [] \mid id\ \textbf{;}\ \text{AttributeDef} \\
\text{Init} &::= \textbf{init}\ \text{CodeBlock} \\
\text{Do} &::= \textbf{do}\ \text{CodeBlock} \\
\text{Iterators} &::= [] \mid \textbf{next}\ \text{CodeBlock}\ \textbf{previous}\ \text{CodeBlock} \\
\text{CodeBlock} &::= \text{Instruction} \mid \textbf{\{}\ \text{InstructionList}\ \textbf{\}} \\
\text{InstructionList} &::= [] \mid \text{Instruction}\ \textbf{;}\ \text{InstructionList} \mid \text{Instruction}\ \textbf{\textbackslash n}\ \text{InstructionList} \\
\text{Instruction} &::= \text{NewComm} \mid \text{NewAgent} \mid \text{Statement} \mid id = \text{Command} \mid \text{Command} \\
\text{NewComm} &::= id = \textbf{communicator}\ \text{StringLiteral} \mid \textbf{communicator}\ \text{StringLiteral} \\
\text{NewAgent} &::= id = \textbf{agentof}\ id\ \text{Attributes} \mid \textbf{agentof}\ id\ \text{Attributes}
\end{aligned}
$$

The language defines a set of reserved words for constructs and built-in attributes, here written in boldface.

3.1 The Agent Abstraction

The development of a Mob mobile agent implies two stages: the first (**init** section) consists on a setup that runs prior to the agent's actual execution. Usually it is used to assign initial values to the agent's attributes. The second (**do** section) defines the agent's behavior throughout its journey.

A MOB agent features several built-in attributes: **email**: email address of the agent's owner; **owner**: identification of the agent's owner; **home**: home hosts of the agent; **itinerary**: the agent's itinerary; **strategy**: definition of a strategy of how the itinerary must be traveled; and **sindex**: the index in the itinerary of the current host. Attributes **owner**, **email** and **home** are read-only, while all the others can be altered during the agent's execution.

Although MOB features several strategies for traversing the itinerary, namely: list, tree, circular and scatterjoin, it allows the programmer to define new ones. An agent's itinerary is seen as an object that can be managed through two iterators **next** and **previous**.

Beside the built-in attributes, an agent may feature as many attributes as the programmer wishes. Their usefulness is to hold values to be retrieved when the agent migrates back home. The following example presents the skeleton of a MOB agent definition, Airline, that includes a new user-defined attribute, price.

```
agent Airline {
    price;
    init { price = 0 }
    do { // Implementation of the agent's actions/behavior }
}
```

Now that an agent behavior is defined an undetermined number of agents can be created. The following example creates an agent named airline owned by johndoe and with **home** hosts host1 and host2. One can also launch several agents at once using the **-n** flag. airlineList will contain the returned list of agent identifiers Notice that the attribute initialization supplied in the agent constructor will not override the ones in the **init** section.

```
airline = agentof Airline -u "johndoe" -h "host1 host2"
airlineList = agentof Airline -n 10 -u "johndoe" -h "host1 host2"
```

Each agent must be associated to an owner, defined in an entry of the Unix-like file named passwd. An entry of such a file must contain the user's login, name, password and group membership. Following the Unix policy for user management, users may belong to groups defined in the groups file, sharing their access permissions. Each MOB enabled host must own both files in order to authenticate each incoming agent. As featured in FTP servers, an agent can present itself as anonymous for limited access to local resources.

3.2 The Communicator Abstraction

Communicators are conceptually equivalent to MPI communicators [4] and allow group communication and synchronization. As presented in the grammar, the **communicator** construct only requires the list of agents (may be empty) that will start the communicator. Other agents may join later.

3.3 Instructions

MOB features a rather small but fully functional set of instructions. Most of the statements included in MOB are common in all scripting languages (**for**, **while**, **if**, **foreach** and **switch**), the only difference lies in the **try** instruction, a little different from the usual error catching instructions found in, for instance, TCL. Its syntax is similar to the try/catch exception handler instruction of Java, allowing specific handling of different types of local run-time system exceptions. MOB provides instructions to define a mobile agent's behavior and its interaction with other agents and external services. These commands can be grouped in the following main sets:

1. agent manipulation: **clone**.
2. mobility: **go**.
3. check-pointing: **savestate** and **getstate**.
4. inter-agent communication: asynchronous (**send**, **recv**, **recvfrom**), synchronous (**bsend**, **brecv**, **brecvfrom**), communicator-wide (**csend**) and multicast (**msend**). There are variants of these functions for use with the HTTP and SMTP protocols (e.g., **httpcsend**; **smtprecv**). These variants are useful to bypass firewalls that only allow connections to ports of regular services.
5. managing communicators: **cjoin** and **cleave**.
6. execution of external commands: **exec**. This functionality allows the execution of commands external to the Mob language. The Mob system features a set of service providers that enable communication through known protocols, such as HTTP, SMTP, SQL and FTP. The interaction with these providers is possible through **exec**'s protocol flag.
7. input/output: Mob's input/output instructions are implemented as syntactic sugar for the **exec** instruction. **open** filename could also be written as **exec** -p fs open filename.

4 Programming with Mob

Now that the language syntax is presented, this section introduces simple Mob programming examples.

We intend to develop two agents: one, airline, capable of querying each host of its itinerary for the price of one airline ticket from Lisbon to Las Vegas; and a second, hotel, capable of querying the hosts of its itinerary for a single's room in a Las Vegas hotel.

In the airline example, the **init** section will set the itinerary as the first ten results of a query to a search engine, and price as zero. The actual program starts with a query to a hypothetical ticketsDB database for the price of the tickets. Note that the syntax of the query will be defined by the implementation of the ticketsDB server and not by the language. The execution of **exec** is protected by a **try** instruction. If no exception is caught the program continues and newprice and price are compared, otherwise nothing is done. Once the end of the program is reached the agent migrates to the next host in its itinerary (default strategy) and restarts the execution of the program. When all of the itinerary has been processed, the agent migrates to one of the hosts defined in the **home** attribute.

In the hotel example, to enhance the efficiency the search is divided among several agents, all members of a ghotel communicator. This provides group communication to spawn new cheaper prices among the agents.

In order to avoid a needless search of an hotel if there are no available airline tickets to Las Vegas, the airline agent can interact with the ghotel communicator through the **csend** command, and inform all the agents from ghotel that they can finish their execution and return home.

```
agent Airline {
    price // declaration of the price attribute
    init {
        itinerary = exec -p http -n 10 www.search_engine.com "airline company" // query engine for itinerary
        price = 0 // initialize price attribute
    }
```

```
    do {
        try { // protect database access with a exception handling mechanism
            newprice = exec -p sql ticketsDB ""price" "Lisbon" "Las Vegas"" // query database for ticket price
            if (newprice < price || price == 0)
                price = newprice
        }
        catch // exception caught
            write log "Could not access database in " + hostname // cannot access database
        if (sindex+1 == [lsize itinerary] && price == 0) // is the search over and no ticket is found?
            csend ghotel "stop" // no ticket found, send "stop" message to ghotel communicator
    }
}
airline = agentof Airline -u "johndoe" -h "host1 host2" // create a new agent
```

In this second example, the **exec** to the search engine is now done outside the agent's definition. The result is scattered among the 10 agents launched. Notice that each agent joins the ghotel communicator in the **init** section and that all the agents terminate their execution and return home if they receive the stop message from the airline agent. Also notice that, every time a cheaper price is found it is spawned to the communicator.

```
agent Hotel {
    price // declaration of the price attribute
    init {
        strategy = "list" // definition of the agent's strategy
        cjoin ghotel // join the ghotel communicator
    }
    do {
        if ([recvfrom airline] == "stop") // did the airline agent terminate without finding any tickets?
            go -h home// search is over, migrate home
        try { // protect database access with a exception handling mechanism
            newprice = exec -p sql hotelDB "price" "single room" // query database for hotel room price
            if (price < newprice) {
                price = newprice
                csend ghotel price // spawn new price to the communicator
            }
            newprice = recv // probe and receive (if any) a new price from the communicator
            if (price < newprice)
                price = newprice
        }
        catch // exception caught
            exec -p smtp email "Could not access database in " + hostname // could not access database
    }
}
ghotel = communicator// create communicator
list = exec -p http -n 50 www.search_engine.com "hotel Las Vegas" // query search engine
for (i = 0; i < 50; i = i+10)
    agentof Hotel -i [lrange list i 10] // create new agents with a sublist of list as the itinerary
```

5 The Compilation Scheme

In this section we briefly sketch how a Mob program can be encoded into the DiTyCO language and run-time. Using the airline example, the agent is encoded into a AirlineAgent extension of the general agent definition Agent. This extension overrides the init and do methods and introduces a new attribute, price.

The following code illustrates the encoding of the airline agent into DiTyCO. The full encoding of the Mob language into DiTyCO may be found in [5].

```
def AirlineAgent(self, name, ..., homes, ..., price) =
    self ? {
        init() =
            {- Encoding of the init section. -}
            AirlineAgent[self, name, ..., homes, ..., price] |
            self!do[]

        do() =
            {- Encoding of the do section. -}
            AirlineAgent[self, name, ..., homes, ..., price]

        {- all the methods inherited from Agent -}
    }
in
```

The instantiation of the Airline definition in Mob corresponds of an instantiation of the AirlineAgent definition in DiTyCO. Continuing with the airline example, the following agent construct

```
airline = agentof Airline -u "johndoe" -h "host1 host2"
```

is encoded in the following DiTyCO code.

```
(1) new var10 StartList[var10] |
    new var11 AddToList[var11, "host2", var10] |
    new var1 AddToList[var1, "host1", var11] |
    ...
(2) new airlineUser StringCell[airlineUser, "johndoe"] |
    ...
    new airlineHomes ListCell[airlineHomes, var1] |
    ...
    new airlineUserDefined0 IntegerCell[airlineUserDefined0, 0] |
(3) new airline AirlineAgent[airline, airlineUser, ..., airlineHomes, ..., airlineUserDefined0] |
    airline!init[]
```

The **agentof** encoding of the airline agent is divided in three sections: the first (1) is dedicated to constructing all the lists required by the AirlineAgent definition (e.g., itinerary, homes, lists for managing incoming and outgoing messages, ...); the second (2) for building Cell objects for each user accessible attributes; and finally (3) creating the airline object and starting its execution, by invoking its init method.

After this static encoding into DiTyCO the Mob program may be compiled and executed using the DiTyCO run-time engine.

6 Conclusions and Future Work

Mob is currently under implementation. All the features, excluding external services and exception mechanisms, are fully encoded in DiTyCO. Future work will focus on the encoding and development of external services, such as, recognition/execution of programs in several high-level languages, building itineraries through external search engines, database communication, and network communication through known protocols, such as SMTP, FTP, or HTTP.

Once the first prototype is ready, case studies will be programmed to provide a base for discussion of the language's strong and weak points. Work will also be done in security (agents and hosts), and in providing an integrated tool for programming, debugging and monitoring of the agents.

Acknowledgments. This work is partially supported by FCT's project MIMO (contract POSI/CHS/39789/2001) and the CITI research center.

References

1. Fuggetta A., Picco G. P., and Vigna G. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
2. Fournet C. and Gonthier G. et al. A Calculus of Mobile Agents. In *International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.
3. Lange D. Programming Mobile Agents in Java. In *WWCA 1997*, pages 253–266, 1997.
4. MPI Forum. The MPI Message Passing Interface Standard. www-unix.mcs.anl.gov/mpi/, 1994.
5. Paulino H., Lopes L., and Silva F. Encoding and Compiling Mob in DiTyCO. To appear.
6. Vitek J. and Castagna G. Seal: A Framework for Secure Mobile Computations. In *Workshop on Internet Programming Languages*, 1999.
7. Honda K. and Tokoro M. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.
8. Straber K., Baumann J., and Hohl F. Mole - A Java Based Mobile Agent System. In Mühlhäuser M., editor, *Special Issues in Object Oriented Programming*, pages 301–308, 1997.
9. Bettini L., De Nicola R., and Pugliese R. Klava: a Java Framework for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
10. Cardelli L. and Gordon A. Mobile Ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.
11. Lopes L., Silva F., Figueira A., and Vasconcelos V. DiTyCO: Implementing Mobile Objects in the Realm of Process Calculi. In *5^{th} Mobile Object Systems Workshop (MOS'99)*, June 1999.
12. Abadi M. and Gordon A. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Computer and Communications Security (CCS'97)*, pages 36–47. The ACM Press, April 1997.
13. Milner R., Parrow J., and Walker D. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.
14. Drossopoulou S., Eisenbach s., and Khurshid S. Is the java type system sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.
15. Gray R. S. Agent Tcl: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, 1995.
16. Vasconcelos V. TyCO Gently. DI/FCUL TR 01–4, Departamento de Informática da Faculdade de Ciências de Lisboa, July 2001.
17. Vasconcelos V., Lopes L., and Silva F. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.