The MOB core language and abstract machine (rev 0.2)

Hervé Paulino\(^1\) and Luís Lopes\(^2\)

\(^1\) CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
Portugal
e-mail: herve@di.fct.unl.pt

\(^2\) CRACS / Departamento de Ciência de Computadores
Faculdade de Ciências, Universidade do Porto
Portugal
e-mail: lblopes@dcc.fc.up.pt

Abstract

Most current mobile agent systems are based on programming languages whose semantics are difficult to prove correct as they lack an adequate underlying formal theory. In recent years, the development of the theory of concurrent systems, namely of process calculi, has allowed for the first time the modeling of mobile agent systems. Languages directly based on process calculi are, however, very low-level and it is desirable to provide the programmer with higher level abstractions, while keeping the semantics of the base calculus.

In this technical report we present the syntax and the semantics of a scripting language for programming mobile agents called MOB. MOB is service-oriented, meaning that agents act both as servers and as clients of services and that this coupling is done dynamically at run-time. The language is implemented on top of a process calculus which allows us to prove that the framework is sound by encoding its semantics into the underlying calculus. This provides a form of language security not available to other mobile agent languages developed using a more ah-doc approach.

Keywords: Mobile Computations, Service-Oriented, Process-Calculus, Programming Language, Run-Time System.

1 Introduction and Motivation

The Service-oriented programming departs from the object-oriented paradigm by separating the data from the processing. Services are thus provided in a transparent way for clients, requiring only knowledge of the contract (service’s interface). One of the main advantages of service-oriented programming is that they provide a framework on which to develop component-based systems. In such systems, inter-component communication is done through the contracts provided by each component. Most of the first service-oriented architectures were built resorting to DCOM \(^6\) or to CORBA \(^9\). Such systems have recently received a lot of attention for distributed systems, namely with the .NET \(^12\), Jini \(^19\) and Openwings \(^10\) platforms.

Another major technology for Web applications is that of Mobile Agents. Mobile agents are computations that have the ability to travel to multiple locations in a network, by saving their state and restoring it in a new host. This paradigm greatly enhances the productivity of each computing element in the network and creates a powerful computing environment, focusing on local interaction. In fact, 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 [10] and Voyager [3] that mostly extend Java classes to define an agent’s behavior. Providing a demonstrably sound semantics for these systems is rather difficult given the gap between the implementation and an adequate formal model. Moreover, since it is not possible to access the state of the Java Virtual Machine (JVM) these systems have a hard time implementing autonomous mobile agents, which occasionally have to move between sites carrying their state and resuming their execution upon arrival to their destination. Another approach, still in the same set, is that of scripting languages such as D’Agents [4] or Ara [11], that fully support agent migration but require specific virtual machine support.

Languages in the second set are based on formal systems, mostly some form or extension of the π-calculus [5, 8]. This process calculus provides the theoretical framework upon which researchers can build solid specifications for programming languages. Languages can thus be proved correct by design relative to some base calculus with a well established theory. Examples of such languages have been implemented in recent years, namely, JoCaml [2], TyCO [18], X-Klaim [1], Nomadic Pict [21], Acute [14] and Alice [15]. Although process calculi are ideal formal tools for the development of mobile agent frameworks, their constructs are very low-level and high-level idioms that provide more intuitive abstractions for programming are desirable.

Here, we introduce a scripting language called Mob that aims to provide both language security and, a user friendly, seamless, programming style more characteristic of the first set of languages. The main novelties introduced in Mob are as follows:

- Mob is service oriented, meaning that services, described as interfaces implemented by agents are the main abstractions of the language. Agents both provide and require services and they are bound to these dynamically as they move through the network;

- the Mob language has been encoded onto a calculus that extends the LSD (Lexically Scoped, Distributed) π-calculus [13] with basic objects, expressions, and a strong migration primitive. The LSDπ-calculus is, in turn, a form of the π-calculus extended with support for distributed execution and mobility of resources and, with a well-studied semantics. Although this is not the focus of this paper, we hope to use the encoding to prove the soundness of the operational semantics of the language. This is particularly important as it provides a form of language security, in the sense of being correct by design, not readily available in related languages;

- the encoding onto the process calculus provides a full specification of the front-end of the compiler for Mob. The output of this front-end are the Mob source programs written into equivalent programs in the TyCO language [18], a concrete implementation of the LSDπ model. This allowed us to use both the compiler and the run-time system previously developed for TyCO, respectively, as the back-end of the Mob compiler and as the basis for the run-time system for Mob;

- the TyCO run-time is implemented on top of the JVM and thus takes advantage of its portability while keeping full control of the state of the virtual machine. This provides full support for agent migration while keeping portability across distinct hardware platforms;

- a user friendly scripting programming style provides the high-level abstractions desirable for programming mobile agents as derived constructs from the core language, thus preserving the semantics;

- extensions to the core virtual machine in the form of external calls can be used to interact with other services (not implemented in Mob, namely SMTP, FTP, HTTP or SQL for databases), as well to execute as programs written in other programming languages, such as Java, Python or Perl. In this view, Mob can be used as a coordination language allowing high-level programming of mobile applications.
In the remainder of this paper we introduce the syntax and semantics of the MOB programming language.

2 General Overview

Before describing the language and its semantics, we give a short overview of the main features of MOB. We want to provide a simple to use programming language based on the high-level type-free programming feel of scripting languages. However, we also want to supply the means to develop clean, modular and structured code, and therefore we adhere to an object-oriented programming paradigm.

Agents and services are the two main abstractions in the language. Conceptually, we view an agent as a special object, with a run-time associated, that can move from host to host in a network and that provides/requires services to/from that network. Agents are handled in MOB programs much like objects in object-oriented languages: the constructor agent is used to define an agent abstraction, and new is used to create a new instance of an agent.

Data-types are defined with the usual class constructor. Objects are instances of these classes and, unlike agents, have no run-time associated. In MOB, objects are first class entities and are also created with the constructor new.

Agents may implement services (declared through the provides keyword), and may, simultaneously, be clients for the services provided by other agents, by requiring a service (the requires keyword). There is no distinction between clients and servers.

Checking that an agent correctly uses or correctly implements the interface of a service is done at compile time by connecting to a network name resolver. The types inferred by the MOB compiler are matched with those assumed for the service in the resolver. If the agent implements a non-registered service, the interface provided becomes the de facto interface for that service. This level of type verification provides some form of program security, namely in method invocation across agents.

To access a service, a programmer is required to get a binding for an agent that provides it. The binding is obtained dynamically using the bind primitive that asks the network resolver for an agent that provides the required 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 (similar to the one found in Telescript [20]). 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 in a transparent way to users.

Since agents supply services, they must be able to handle multiple incoming requests. To cope with such a demand we have designed agents to be multi-threaded. For example, each remote method invocation is handled by a dedicated thread. This design justified also the inclusion of explicit thread creation (fork) and synchronization (join, wait and notify).

Objects in MOB can be accessed simultaneously by any number of threads, therefore a scheme to allow for exclusive access is required. We provide such a scheme with two instructions lock and unlock that allows a primitive form of mutual exclusion in data access.

Interaction with external services is provided by MOB through the exec instruction. In general, exec is used to implement extensions to the core MOB language to support more functionality. These external services may be implemented in other languages, such as Java, C, TCL or Perl, or allow interaction with network services, such as WWW queries, FTP transactions, or e-mail communication.

3 The Syntax

We now present the syntax for the core language. The full form of the language is obtained by providing derived constructs for higher-level programming, while keeping the underlying semantics
Program ::= D P ; exit
D ::= requires S
| agent X(\$) provides S requires S M
| service S {\$}
| class X(\$) M
P ::= I ; P
| ε
I ::= go (v)
| return (v)
| join (x)
| wait (x)
| notify (x)
| lock (x) | unlock (x)
| if (v) { P } else { P }
| while (v) { P }
| break
| exit
| x = V
| o.x = v
V ::= new X (\$)
| fork { P }
| bind (S v) | bind(S)
| host ()
| exec (v)
| o.m (v)
| e
| o.x
M ::= {m_1(\$_1) { P_1 } . . . m_n(\$_n) { P_n }}
e ::= v | e bop e | uop e
v ::= o | c | null
o ::= x | self
c ::= bool | int | string
bop ::= + | − | * | / | % | ^
| && | || | == | !=
| < | > | <= | >=
uop ::= ! | −

Table 1: Syntax of the MOB programming language
Program $\in$ Program A Mob program
$D \in$ Definition Class, agent and service definitions, and service requirement
$X, Y \in$ Class Class or agent identifiers
$S \in$ Service Service identifiers
$m \in$ MethodId Method labels
$P \in$ InstructionSeq Sequence of program instructions
$I \in$ Instruction Program instruction
$V \in$ AssignValue Assignable value
$M \in$ Method Set of methods
$e \in$ Expression Basic expression
$x, y \in$ Var Variable identifier
$c \in$ Constant Constant value
$v \in$ LangValue Variable or constant
$o \in$ Target Object or agent
$bop \in$ BinOp Binary operations
$uop \in$ UnOp Unary operations

Table 2: Phrase categories

| agent | provides | requires | class | service | main | new |
|-------|----------|----------|-------|---------|------|-----|
| go    | bind     | fork     | join  | wait    | notify | lock |
|       |          |          |       |         |       | unlock|
| if    | else     | while    | break | return  | exit  | self |
|       |          |          |       |         |       | null |
|       | !        | {        | }     | (       | )    | ;    |
| +     | −        | *        | /     | %       | ==   | !    |
|       |          |          |       |         | >    | >=   |
|       |          |          |       |         | <=   | &&   |
|       |          |          |       |         | ||   |      |

Table 3: Reserved words

of the core language.

As defined in the grammar in table 1 a Mob program is syntactically a sequence of definitions (D) followed by a sequence of instructions (P) terminated by exit. Tables 2 and 3 present, respectively, the phrase categories and the set of reserved words, that may not be used as identifiers, required by the syntax of a Mob program.

We choose only to allow the assignment of language values to class and agent attributes. The assignment of an element in $V$ to an attribute must use an intermediate auxiliary variable. This will be overcame in the full form of the language, with the inclusion of syntactic sugar. Here, it greatly simplifies the definition of the language’s semantics, since we do not have to define duplicate rules for the assignment of the elements in $V$ to either variables and attributes.

Constant identifiers in Mob are divided in the following classes: booleans, elements of $\text{Bool} = \{ \text{true}, \text{false} \}$ ranged over by $\text{bool}$; integers, elements of $\text{Int}$ ranged over by $\text{int}$ and, strings, elements of $\text{String}$ ranged over by $\text{string}$, defined by the regular expression: "^[\"\n]*)$n. Hosts in Mob are represented with strings.

Syntactic Restrictions

The concrete syntax of Mob imposes some syntactic restrictions over the syntax of a program:

- service definitions must precede service requirements that, in turn, must precede the definitions of classes and agents;
- an agent must implement the main method;
• the return instruction can only appear within the body of a method;
• the break instruction can only appear inside the body of a while instruction;
• the go and exit instructions can only appear inside the body of a method in an agent definition;
• the method identifiers \( \tilde{m} \) in \{ \( m_1(\tilde{x}_1) \{ P_1 \} \ldots m_n(\tilde{x}_n) \{ P_n \} \) \}, and in service \{\tilde{m}\} are pairwise distinct;
• the parameters \( \tilde{x} \) in an agent \((\text{agent } X(\tilde{x}) \text{ provides } \tilde{S} \text{ requires } \tilde{M})\), a class \((\text{class } X(\tilde{x}) \ M)\) or a method \((m(\tilde{x})\{P\})\) definitions are pairwise distinct;
• a variable \( x \) is bound in \( P \) with an assignment \((x = V; P)\), where \( V \) is a language construct that may appear on the left of an assignment; is bound in \( M \) in an agent definition \((\text{agent } X(\tilde{x}) \text{ provides } \tilde{S} \text{ requires } \tilde{M})\) or class definition \((\text{class } X(\tilde{x}) \ M)\) if it is one of the \( \tilde{x} \); is bound in \( P \) in a method \((m(\tilde{x})\{P\})\) if it is one of the \( \tilde{x} \):
• an agent identifier \( X \) is bound in \( \tilde{D}, \tilde{D}', \ M \) and \( P \) with a statement \( \tilde{D} \text{ agent } X(\tilde{x}) \text{ provides } \tilde{S} \text{ requires } \tilde{M} \ M \tilde{D}' \ P; \)
• a class identifier \( X \) is bound in \( \tilde{D}, \tilde{D}', \ M \) and \( P \) with a statement \( \tilde{D} \text{ class } X(\tilde{x}) \ M \tilde{D}' \ P; \)
• a service identifier \( S \) is bound in \( \tilde{D} \) and \( P \) with a statement \( \text{service } S \ \{ \tilde{m} \} \tilde{D} \ P; \)
• the sets of free variables, free agents, free classes and free services are defined accordingly. Well formed Mob programs are closed for variables, agent identifiers, class identifiers and service identifiers.

4 The Mob Abstract Machine

We provide the semantics for a Mob network in the form of an abstract state transition machine. A Mob network is composed by a set of hosts, which are abstractions for network nodes. Hosts define the boundaries where computations take place in a Mob network. The computing units of the Mob language are agents. There may be several agents running concurrently in a given host at any given time. In our approach there is no distinction between clients and servers, any agent may behave as a client requesting a service while also providing services to others. This is achieved by implementing multi-threaded agents to handle multiple requests concurrently. The threads in an agent share the same heap space whilst having independent control data-structures.

Before defining the structure of the network, of agents and of threads we first introduce the syntactic categories and auxiliary functions.

4.1 Syntactic Categories and Data-Structures

Besides the syntactic categories of the language, the abstract machine requires a new set of categories defined in table 4.

The abstract machine has two layers: agents and threads. A network is described as a set of agents running concurrently plus a resolver for agents and services. Agents are described as collections of threads running concurrently and sharing the agent’s resources, namely its code and address space. Agents are abstractions for autonomous programs running on network hosts, that interact with the network by spawning new agents or moving between hosts. Inter-agent interaction is performed by invoking methods. The abstract machine requires some syntactic categories and data-structures to be defined:

\footnote{We denote the syntactic definition of methods in the language and their internal representation in the abstract machine with the same letter \( \tilde{M} \). We choose to do so because both relate to the same information, although with different representations.}
Table 4: Syntactic categories of the MOB virtual machine

- an **Agent Key** is an element of the set \( \text{AgentKey} \subseteq \text{String} \), ranged over by \( a, b \), and represents a unique, network-wide, key for an agent;

- a **Host** is an element of the set \( \text{Host} \subseteq \text{String} \), ranged over by \( h \), and represents a unique, network-wide, host identifier;

- an **Instruction** is an element of the set \( \text{Instruction} \), ranged over by \( I \), and represents a MOB instruction. A sequence of instructions separated by ; is denoted by \( P \in \text{InstructionSeq} \);

- a **Method** represents a set of methods and is a map of the form \( \text{Method} = \text{MethodId} \mapsto \text{Var}^* \times \text{InstructionSeq} \), ranged over by \( M \), and represents the methods in a class or an agent;

- the **Code** repository for an agent is a map defined as \( \text{Code} = \text{Class} \mapsto \text{Bool} \times \text{Var}^* \times \text{Method} \times \text{Code} \times \text{Service}^* \), ranged over by \( C \) and represents all the code required by a class or agent. The boolean value makes the distinction between the two (true = agent, false = class);

- a **Heap Reference** is an element of the set \( \text{HeapRef} \), ranged over by \( r \), and is an abstraction for an address in the address space of an agent. Heap references in MOB are qualified with the key of their hosting agent (e.g., reference \( r \) in the heap of agent \( a \) should be interpreted as \( r@a \)) and thus are unique in the network. To ease the reading of the rules we omit the qualifier of a heap reference when it is accessed from within its hosting agent. The value \( \text{null} \in \text{HeapRef} \) represents an undefined heap reference;

- a **Thread Reference** is an element of the set \( \text{ThreadRef} \subseteq \text{HeapRef} \), ranged over by \( t \), and represents a reference to a thread. Note that this subset includes \( \text{null} \);

- a **Constant** is an element of the set \( \text{Constant} = \text{Bool} \cup \text{Int} \cup \text{String} \), ranged over by \( c \), and represents a primitive value of the language;

- a **Value** is an element of the set \( \text{Value} = \text{Constant} \cup \text{HeapRef} \), ranged over by \( u \);

- an **Environment** is a map defined as \( \text{Bindings} = \text{Var} \mapsto \text{Value} \), ranged over by \( B \), that represents a map from identifiers in the code to constants or references in the heap. We will represent the binding from an identifier \( x \) to a value \( u \) as \( x : u \);
• a **Closure** is an element of the set \( \text{Closure} = \text{Bool} \times \text{Bindings} \times \text{Class} \), ranged over by \( K \), and represents the closure for an instance of a class or an agent located in an address space. The boolean value makes the distinction between instances of classes and agents;

• a **Heap** is a map defined as \( \text{Heap} = \text{HeapRef} \mapsto \text{ThreadRef} \times (\text{Closure} \cup \text{Value}) \), ranged over by \( H \), and represents the address space of an agent. The contents associated with heap references may be accessed with mutual exclusion using locks. The thread reference in the image of a reference indicates which thread holds the lock to the contents. Unlocked references hold \( \text{null} \) has their thread reference;

• a **Code Stack** is an element of the set \( \text{CodeStack} = \text{Stack}((\text{Bindings} \times \text{InstructionSeq})) \), ranged over by \( Q \), and represents the stack of blocks of code, used as a mechanism to implement **while** loops. The block of the top of the stack is the one currently being executed by the machine. As soon as it terminates, it is popped from the stack and the execution continues with the code of the block found at the top of the stack. We refer to elements of the stack as code-blocks;

• a **Pool of Running Threads** is an element of the set \( \text{Pool(RunningThread)} \), ranged over by \( T \), where \( \text{RunningThread} = \text{ThreadRef} \times \text{CodeStack} \times \text{HeapRef} \) represents a flow of execution. In the definition of a thread \( (t, q, r) \), the thread reference \( t \) is a location in the heap to which the thread is bound. The reference \( r \) is a heap reference where the thread may place a result;

• a **Suspended Thread** is a map defined as \( \text{SuspendedThread} = \text{HeapRef} \mapsto 2^\text{RunningThread} \), ranged over by \( W \), and represents threads suspended (waiting) on heap references;

• a **Pool of Agents** is an element of the set \( \text{Pool(Agent)} \), ranged over by \( A \), where \( \text{Agent} = \text{AgentKey} \times \text{Host} \times \text{Code} \times \text{Heap} \times \text{Pool(RunningThread)} \times \text{SuspendedThread} \) represents a multi-threaded autonomous computation. We write an agent \( (a, h, C, H, T, W) \) as \( a(h, C, H, T, W) \) thus exposing the agent’s key;

• a **Service Type** is an element of the set \( \text{Type} \), ranged over by \( \alpha \);

• a **Name Resolver**, \( R \), is composed by two maps, \( \text{NameService} = \text{AgentNameService} \times \text{ServiceNameService} \). The first, defined as \( \text{AgentNameService} = \text{HeapRef} \mapsto \text{Host} \), ranged over by \( \text{ANS} \), represents a network-wide name resolver for locating agents. The second, defined as \( \text{ServiceNameService} = \text{Service} \mapsto \text{Type} \times 2^{\text{HeapRef}} \), ranged over by \( \text{SNS} \), represents a network-wide name resolver for obtaining the type and implementations of a service;

• a **Network** is an element of the set \( \text{Network} = \text{Pool(Agent)} \times \text{NameService} \), ranged over by \( \mathcal{K} \), and represents a MOB network computation.

### 4.2 Auxiliary Definitions

Function **tryAccess** checks if, in a heap \( H \), the access to the value located at \( r \) is granted to a thread identified by \( t \).

\[
\text{tryAccess} : \text{Heap} \times \text{HeapRef} \times \text{ThreadRef} \mapsto \text{Bool}
\]

\[
\text{tryAccess}(H, r, t) = \begin{cases} 
\text{true}, & \text{if } H(r) = (t, \_) \text{ or } H(r) = (\text{null}, \_)

\text{false}, & \text{if } H(r) = (t', \_) \text{ and } t' \neq t
\end{cases}
\]

Function **tryLock** tries to grant the lock for a value located at \( r \), in a heap \( H \), to a thread identified by \( t \). Function **tryUnlock** tries to release the lock for a value located at \( r \) in a heap \( H \). The result of both functions is a heap modified (or not) by the operation, and a boolean value indicating if the operation was successful. Both these functions are atomic.
tryLock : Heap × HeapRef × ThreadRef → Heap × Boolean
tryLock(H, r, t) = \{(H + \{r : (t, K)}), true\}, if tryAccess(H, r, t) = true 
and H(r) = (\_ K)
\{(H + \{r : (t, u)}), true\}, if tryAccess(H, r, t) = true 
and H(r) = (\_ u)
(H, false), if tryAccess(H, r, t) = false

tryUnlock : Heap × HeapRef × ThreadRef → Heap × Boolean
tryUnlock(H, r, t) = \{(H + \{r : (null, K)}), true\}, if tryAccess(H, r, t) = true 
and H(r) = (\_ K)
\{(H + \{r : (null, u)}), true\}, if tryAccess(H, r, t) = true 
and H(r) = (\_ u)
(H, false), if tryAccess(H, r, t) = false

Remember that unlocked references have null as their locking reference.
Function code returns the code for a method m from a given class or agent representation:

\[
\text{code} : (\mathrm{Var}^* \times \mathrm{Method} \times \mathrm{Service}^*) \times \mathrm{MethodId} \rightarrow \mathrm{Var}^* \times \mathrm{InstructionSeq}
\]

\[
\text{code}((\emptyset, M, \emptyset), m) = M(m)
\]

Function codeIn returns the code closure for a set of methods. The result is a code repository, built from another received as argument, that is composed of the code for all the classes referenced in the set of methods.

\[
\text{codeIn} : \mathrm{Code} \times \mathrm{Method} \rightarrow \mathrm{Code}
\]

\[
\text{codeIn}(C, M \cdot (\emptyset, x = \text{new} X(\emptyset); P)) = \{X : C(X)\} + \text{codeIn}(C, M \cdot (\emptyset, P))
\]
\[
\text{codeIn}(C, M \cdot (\emptyset, I; P)) = \text{codeIn}(C, M \cdot (\emptyset, P))
\]
\[
\text{codeIn}(C, M \cdot (\emptyset, \epsilon)) = \text{codeIn}(C, M)
\]
\[
\text{codeIn}(\emptyset) = \emptyset
\]

Function evalSeq returns the evaluation of a sequence of expressions, each element being evaluated by the eval function. The evaluation requires the knowledge of the state of the heap H, the heap reference of the thread computing the expression x, and its environment B. The evaluation is fairly standard, however a particularity requires some closer attention. A variable may contain a heap reference whose value is another reference, which forces the resolution of the indirection.

The value of a constant is given by the built-in val function, and the result of the relational and arithmetic built-in operations is given by the bop and uop built-in functions.

\[
\text{evalSeq} : (\mathrm{Heap} \times \mathrm{ThreadRef} \times \mathrm{Bindings} \times \mathrm{Expression}^*) \rightarrow \mathrm{Value}^*
\]

\[
\text{evalSeq}(H, t, B, v \ \emptyset) = \text{eval}(H, t, B, v) \ \text{evalSeq}(H, t, B, \emptyset)
\]
\[
\text{evalSeq}(H, t, B, \epsilon) = \epsilon
\]

\[
\text{eval} : (\mathrm{Heap} \times \mathrm{ThreadRef} \times \mathrm{Bindings} \times \mathrm{Expression}) \rightarrow \mathrm{Value}
\]

\[
\text{eval}(H, t, B, c) = \text{val}(c)
\]
\[
\text{eval}(H, t, B, null) = null
\]
\[
\text{eval}(H, t, B, x) = \text{val}(c)
\]
\[
\text{eval}(H, t, B, x) = \text{null}
\]
\[
\text{eval}(H, t, B, x) = t'
\]
\[
\text{eval}(H, t, B, x) = r
\]
\[
\text{eval}(H, t, B, x) = u
\]
\[
\text{eval}(H, t, B, \text{bop} \ \epsilon') = \text{bop}(u, u')
\]
\[
\text{eval}(H, t, B, \text{uop} \ \epsilon) = \text{uop}(u)
\]
Function \(\text{copySeq}_{ab}\) returns a copy of the closures for a sequence of values located in the heap of an agent \(a\), plus all the code they require. The new references created to duplicate the given closure are located in the target agent \(b\). The function takes as arguments the code repository \(C\) and heap \(H\) of the original agent \(a\), and the sequence of values to be copied \(\bar{u}\). The copy of each value is computed by function \(\text{copy}_{ab}\).

\[
\begin{align*}
\text{copySeq}_{ab} : & \text{Code} \times \text{Heap} \times \text{Value}^* \rightarrow \text{Code} \times \text{Heap} \times \text{Value}^* \\
\text{copySeq}_{ab}(C, H, u \bar{u}) & = (C' + C', H' + H', u'\bar{u}') \\
\text{copySeq}_{ab}(C, H, c) & = (\emptyset, \emptyset, c)
\end{align*}
\]

where \(\text{copySeq}_{ab}(C, H, u) = (C', H', u')\) and \(\text{copySeq}_{ab}(C, H, \bar{u}) = (C'', H'', \bar{u}'')\)

\[
\begin{align*}
\text{copy}_{ab} : & \text{Code} \times \text{Heap} \times \text{Value} \rightarrow \text{Code} \times \text{Heap} \times \text{Value} \\
\text{copy}_{ab}(C, H, r@a) & = (\emptyset, \emptyset, r@a) \\
\text{copy}_{ab}(C, H, r@a') & = (\emptyset, \emptyset, r@a') \\
\text{copy}_{ab}(C, H, c) & = (\emptyset, \emptyset, c)
\end{align*}
\]

if \(H(r@a) = (\text{false}, \{x : \bar{u}\}, X)\)

where \(\text{copySeq}_{ab}(C, H, \bar{u}) = (C', H', \bar{u}')\) and \(r'@b \in \text{HeapRef fresh}\)

if \(H(r@a) = (\text{true}, \{x : \bar{u}\}, X)\)

if \(H(r@a) = (\_ u)\)

if \(a' \neq a \neq b\)

The \text{run} function places a set of pool of running threads in concurrent execution.

\[
\begin{align*}
\text{run} : & 2^{\text{RunningThread}} \rightarrow \text{Pool(RunningThread)} \\
\text{run}((t_1, Q_1, T_1), \ldots, (t_n, Q_n, T_n)) & = (t_1, Q_1, T_1) | \cdots | (t_n, Q_n, T_n)
\end{align*}
\]

4.3 The Initial and Final States

Based on the above definitions, we may write the syntax for a network as follows:

\[
\begin{align*}
\text{N} & := \text{A}, \text{R} & \text{Network} \\
\text{A} & := \text{A} | \text{A} & \text{Concurrent agents} \\
| & a(h, C, H, T, W) & \text{Running agent} \\
| & 0_a & \text{Terminated agent} \\
\text{T} & := T | T & \text{Concurrent threads} \\
| & (t, Q, r) & \text{Running thread} \\
| & 0_t & \text{Terminated thread}
\end{align*}
\]

For the sake of simplicity we assume that agents run in a static network with no failures. In other words, the set of available hosts, \(\text{Host}\), is constant. Here we describe the abstract machine from the point of view of the execution of one agent. Thus, when we start running an agent, the network may already have a pool of agents \(\text{A}\) running concurrently and distributed among the network nodes in the set \(\text{Host}\), together with the resolver \(\text{R}\):

\[
\text{A, R}
\]
We launch a program \((\mathcal{D} P)\) in the network by encapsulating its code \((P)\) in an agent that is placed in a host specified by the user. The code repository for the program is collected at compile-time with function \(\text{codeCollect}\) that we will define ahead. Thus, when the agent is launched into the network it already contains all the code it requires. The initial state of the execution of a program with code \((\mathcal{D} P)\) is thus:

\[
\text{a}(h, \text{codeCollect}(\mathcal{D}), \emptyset, \text{launch}(\emptyset, P, \text{null}), \emptyset) | A, R
\]

where \(a\) is a fresh agent key, \(h\) is the local host and \(\text{launch}(B, P, r)\) is a macro that creates a new thread with an environment \(B\) (here \(\emptyset\)), a code \(P\) (here \(\mathcal{D}\)) and a return reference \(r\) (here \(\text{null}\)):

\[
\text{a}(h, C, H, \text{launch}(B, P, r) | T, W) | A, R \quad \text{def} = \quad \text{a}(h, C, H + \{t : (t, \text{null})\}, (t, (B, P), r) | T, W) | A, R \quad t \in \text{ThreadRef fresh}
\]

Note that no heap reference is associated with the agent \(a\), since a program does not provide any methods, nor has attributes. Moreover, \(a\) is not registered in \(R\), and thus is not accessible to the network. The registry is a precondition for an agent to migrate (further detail in rule \([\text{Go}]\)), and thus, \(\mathcal{D}\) programs cannot migrate, just agents.

Agents are daemons by default and must be explicitly terminated by the \texttt{exit} instruction, which produces the terminated agent \(0_A\). Thus, at the end of the program running in agent \(a\), the configuration of the network will be of the form:

\[
0_A | A', R'
\]

Such an agent can thus be garbage collected and produce the state:

\[
A', R'
\]

### 4.4 Code Collection

The compile-time code collection is defined by function \(\text{codeCollect} : \text{Definition} \times \mathbb{N} \rightarrow \text{Code}\) that returns a code repository with all the code required by a sequence of class and agent definitions. We present the function in a case by case analysis.

A service specifies an interface implemented by some \(\mathcal{D}\) agent. Service definitions are used to supply information to the type-system. Type-checking of a \(\mathcal{D}\) program is performed at compile-time by matching the inferred types for services required or implemented by the agents with their definitions kept in the resolver. If the service is required by the program or if the program implements a known service in the network then its inferred type must match the interface for the service kept in the resolver. If the service is introduced for the first time by the program (an interface for it does not yet exist in the resolver) then the type inferred for the service will become the adopted interface for the service as registered in the resolver. So, when the agent is created, the \(\text{SNS}\) map is updated by adding the reference of the agent to every entry associated to a one of the implemented services. Anyway, these are handled at compile time and there is no need for them in the abstract machine.

\[
\text{codeCollect(service } S \{\mathcal{D}\} \mathcal{D}) = \text{codeCollect}(\mathcal{D})
\]
Simple classes define abstract data-types and we call their instances objects. The entry in the code repository associated with this definition contains a closure with slots for the code for all the classes and agents that are required by this class, its attributes, the code for its methods, and an empty sequence of implemented services, since an object does not provide any services. Remember that false indicates that the entry contains the code of a class.

\[
\text{codeCollect}(\text{class } X) \{ m_1(\xi_1) \{ P_1 \} \ldots m_n(\xi_n) \{ P_n \} \} \bar{D} = \\
\{ X : (\text{false}, \bar{\xi}, \bar{\mu}', \text{codeIn}(\bar{\mu}'), \epsilon) \} + \text{codeCollect}(\bar{D})
\]

where \( M' = \{ m_1 : (\xi_1, P_1), \ldots, m_n : (\xi_n, P_n) \} \).

Some classes are special in the fact that they represent full computations. We call their instances agents, and we use a different keyword to differentiate them. Otherwise, the definition of an agent is very much like that of a regular class, it contains the code for all the classes and agents required, the attributes, the code for the methods and indicates which services are provided by the agent.

The requires keyword supplies information to the type-system, indicating which services are required by the agent and that their uses must be checked against the definitions in the SNS. The provides keyword does not only supply information to the type-system, but also states which service entries must be updated whenever an instance of the agent is created (rule [NewAgent]). To hold this information when necessary, we keep this sequence in the agent’s code closure.

\[
\text{codeCollect}(\text{agent } X) \text{ provides } \bar{S} \text{ requires } \bar{S}' \{ m_1(\xi_1) \{ P_1 \} \ldots m_n(\xi_n) \{ P_n \} \} \bar{D} = \\
\{ X : (\text{true}, \bar{\xi}, \bar{\mu}', \text{codeIn}(\bar{\mu}'), \bar{S}) \} + \text{codeCollect}(\bar{D})
\]

where \( M' = \{ m_1 : (\xi_1, P_1), \ldots, m_n : (\xi_n, P_n) \} \).

Note that agents may not always implement or require services and thus both \( \bar{S} \) or \( \bar{S}' \) may be empty sequences.

The requires keyword can be also used by itself to indicate which are the services required by a program. Once again this only provides information to the compile time type checking, and thus there is no need to pass it to the run-time. Here we also present the base case for the recursion.

\[
\text{codeCollect}(\text{requires } \bar{S}) = \text{codeCollect}(\bar{D})
\]

\[
\text{codeCollect}(\epsilon) = \emptyset
\]

4.5 The Congruence Rules

The computation in the abstract machine is driven by a set of reduction rules that operate over the thread or the agent at the most left in the respective pool. Thus, in order to be able to commute, associate and garbage collect threads and agents in their pools, we need a set of congruence rules. These will allow for the re-writing of both pools, into semantically equivalent ones, where the configuration is accordingly to the reduction rules to be applied. The congruence rules for a pool of agents are:

\[
\begin{align*}
\text{[AgentSwap]} & \quad A \mid A' \equiv A' \mid A \\
\text{[AgentAssoc]} & \quad A \mid (A' \mid A'') \equiv (A \mid A') \mid A''
\end{align*}
\]

The congruence rules for threads for threads are:
Rule [ThreadInAgent] allows the use of the congruence rules for threads in the layer of agent states:

\[ \text{ThreadInAgent} \quad T \equiv T' \]

4.6 The Reduction Rules

Each Mob instruction requires at least one machine transition to be processed. The rules are written using the usual forms in the definition of operational semantics. The rules are of two forms: the first are denoted by

\[ A \rightarrow A' \]

and operate simply over a pool of threads. They are used whenever the operation to be performed does not modify the name resolver \( R \). The second, denoted by

\[ A, R \rightarrow A', R' \]

include the resolver and are used whenever the operation requests data from the resolver or modifies its contents in some way. To widen the scope of the first form of reductions to whole network we define rule rule [AgentRed]. This allows us to define rules focused on one agent alone, whenever the remainder of the network is not affected.

\[ \text{AgentRed} \quad A \rightarrow A'' \]

Rule [Cong] allows reduction to occur under structural congruence:

\[ A \equiv A', A', R \rightarrow A'', R'' \quad A'', R'' \equiv A'''' \]

Next we provide the rules for the language constructs.

Creation of Objects and Agents

The instantiation of a regular class creates a new object. It reserves a block of heap space for a closure representing the object. The closure holds the values of the attributes, a special attribute self, that is a reference to the object itself, and keeps a link for the code of the class.

\[ \text{NewObject} \quad \text{evalSeq}(H, t, B, \bar{v}) = \bar{u} \quad C(X) = (\text{false}, \bar{x}, \ldots, \epsilon) \quad x' \in \text{HeapRef fresh} \]

\[ a(h, C, H, t, (B, x = \text{new } X(\bar{v}) P :: Q, r) :: T, W) \rightarrow a(h, C, H + \{x' : (\text{null}, (\text{false}, \{\text{self} : x', \bar{x} : \emptyset\}, X))\}, (t, (B + \{x : x'\}, P :: Q, r) :: T, W) \]

Agents in Mob are similar to objects, but they have an execution unit associated to them. A new agent is placed in the network’s pool of agents. In the beginning, its location is the same as the agent that created it. It is initiated with a heap containing its closure at \( x' \) (as in the [NewObject] rule). A new thread is created to execute the code of the agent’s main method with the agent’s environment \( B' \), given by the attributes and self. main is a required method that
defines the agent’s initial behavior, an approach common to many programming languages. The parent agent keeps a binding to the reference $r'@b$ of the created agent in $x$.

The code repository for the new agent is composed of the code required by the values given as argument to the constructor ($C'$), plus all the code required by the agent definition ($\{X : C(X)\}$).

\[\text{[NewAgent]}\]
\[
\begin{align*}
\text{evalSeq}(\text{H, t, B, } \bar{v}) &= \bar{u} \quad \text{copySeq}_{\text{ab}}(\text{C, H, } \bar{u}) = (C', H', \bar{u}') \quad C(X) = (\text{true}, \bar{x}, \bar{M}, S_1 \cdots S_n) \\
\text{code}(\text{C(X, main)}) &= (\epsilon, P') \quad B' = \{\text{self : r', x : }\bar{u}'\} \quad b \in \text{AgentKey} \text{ and } r'@b \in \text{HeapRef fresh} \\
\text{SNS}(S_i) &= (\alpha_i, K_i) \cdots \text{SNS(S_n)} = (\alpha_n, K_n) \quad K_i = \{r@i | i \in \{1, \ldots, n\}\} \cdots K_n = \{r@n | i \in \{1, \ldots, m\}\}
\end{align*}
\]

where $k$ denotes the number of services implemented by the agent, and $n$ and $m$ denote, respectively, the number of implementations of services $S_1$ and $S_n$ in the network.

Note that both maps of the resolver are updated. The reference $r'@b$ holding the agent’s closure will be the key in the SNS map to locate the agent’s current host ($r'@b : h$). Every entry of the SNS map corresponding to each of the agent’s implemented services given as $K_1, \ldots, K_n$, will be updated with $r'@b$. e. g., $\{S_1 : (\alpha_1, K_1 + \{r'@b\})\}$ for the implemented service $S_1$.

\section*{Multi-threaded Agents}

The \texttt{fork} instruction allows the explicit creation of a new thread by the programmer. The new thread inherits the environment of its creator and a handle is returned to the caller. This handle is associated to a newly created heap reference, that contains a null value and is used for inter-thread synchronization. The synchronization is achieved by granting the thread exclusive access to itself (see [JOIN] rules).

\[\text{[Fork]}\]
\[
\frac{t' \in \text{ThreadRef fresh}}{a(h, C, H, \{t. (B, x = \texttt{fork} \{P\}) :: Q, r\} | T, W) \rightarrow a(h, C, H, \{t'. (t', \texttt{null})\}, (t. (B + \{x : t'\}) :: Q, r) | (t', (B, P'), \texttt{null}) | T, W)}
\]

A thread can suspend waiting for the completion of another thread using the instruction \texttt{join}. The instruction uses the thread’s handle returned by a previous \texttt{fork} statement. While it is running, a given thread has a reference in the heap associated to it ($t'$). In this scenario, any other thread that tries to perform the \texttt{join} operation will suspend on $t'$.

\[\text{[JoinSuspend]}\]
\[
\frac{\text{eval}(\text{H, t, B, x}) = t' \quad H(t') = (t', \texttt{null}) \quad t' \neq t}{a(h, C, H, \{t. (B, \texttt{join}(x) :: Q, r) | T, W) \rightarrow a(h, C, H, W + \{t' : (t. (B, P) :: Q, r)\})}
\]

If the thread on which the synchronization is performed is no longer running, the reference associated to it is no longer locked. In this case, the operation succeeds and the execution continues.

\[\text{[Join]}\]
\[
\frac{\text{eval}(\text{H, t, B, x}) = t' \quad (t = t' \lor H(t') = (\texttt{null, null}))}{a(h, C, H, \{t. (B, \texttt{join}(x) :: Q, r) | T, W) \rightarrow a(h, C, H, (t. (B, P) :: Q, r) | T, W)}
\]

If the code currently under execution terminates and the stack has no more elements, the thread has run out of code to execute and terminates, as in rule [END]. To give a more expressive writing of the reduction rules involving synchronization, we define the \texttt{notify} macro. The macro represents a thread that wakes up all the threads suspended on reference $r$.

\[\text{notify(r) def } = (\texttt{null, } e, r)\]
When a thread terminates its execution it uses this macro to wake up all the threads suspended on it.

\[
\text{[END]} \quad a(h, C, H, (t, (B, e), \text{null}) \mid T, W) \rightarrow a(h, C, H, \text{notify}(t) \mid T, W)
\]

The [\text{NOTIFYTHREAD}] rule wakes up every thread suspended on the given reference. \(W(r)\) is the set of threads suspended on the reference \(r\).

\[
\text{[NOTIFYTHREAD]} \quad a(h, C, H, \text{notify}(r) \mid T, W) \rightarrow a(h, C, H, T \mid \text{run}(W(r)), W_{\text{dom}(W) - \{r\}})
\]

Explicit synchronization is also supported in \text{MOB} by the \text{wait} and \text{notify} instructions, that The first allows a thread to suspend on a reference, while the second is the language support to create to a \text{notify} thread, and thus wake every thread suspended on the given reference.

\[
\text{[WAIT]} \quad \text{eval}(H, t, B, x) = r' \quad r' \neq t \quad a(h, C, H, (t, (B, \text{wait}(x); P) :: Q, r) \mid T, W) \rightarrow a(h, C, H, T, W + \{r' : (t, (B, P) :: Q, r)\})
\]

\[
\text{[NOTIFY]} \quad \text{eval}(H, t, B, x) = r' \quad a(h, C, H, (t, (B, \text{notify}(x); P) :: Q, r) \mid T, W) \rightarrow a(h, C, H, \text{notify}(r) \mid (t, (B, P) :: Q, r) \mid T, W)
\]

**Agent Movement and Discovery**

An agent may move to another host, changing the topology of the distributed computation, rule [\text{Go}]. The original host will proceed without the agent, and the later will resume its execution concurrently with the agents at the target host. In order to migrate an agent must be registered in the ANS map. The reference associated to the agent in ANS is discovered by following the binding for the \text{self} identifier. This can be done, since the \text{go} instruction can only appear inside the body of an agent definition’s method.

\[
\text{[Go]} \quad \text{eval}(H, t, B, v) = h' \quad \text{B(self)} = r' \otimes a \quad r' \otimes a \in \text{dom(ANS)} \quad h' \in \text{Host} \quad a(h, C, H, (t, (B, \text{go}(v); P) :: Q, r) \mid T, W) \mid A, (\text{ANS}, \text{SNS}) \rightarrow \\
\quad a(h', C, H, (t, (B, P) :: Q, r) \mid T, W) \mid A, (\text{ANS} + \{r' \otimes a : h'\}, \text{SNS})
\]

An agent may invoke a method in another agent only if it has a binding for the target agent’s closure. Agent discovery in \text{MOB} is service-oriented, meaning that agents are discovered for the services they implement. The instruction \text{bind} consults the network resolver and retrieves a heap reference, \(r' \otimes b\), associated with an agent that implements a service \(S\) and is presently running in host \(h\). Note that an agent cannot obtain a binding for itself.

\[
\text{[Bind]} \quad \text{eval}(H, t, B, v) = h' \quad \text{SNS}(S) = (\alpha, \{r \otimes a_1, \ldots, r \otimes a_n\}) \quad \exists r' \otimes b \in \{r \otimes a_1, \ldots, r \otimes a_n\} : \text{ANS}(r' \otimes b) = h' \land b \neq a \\
a(h, C, H, (t, (B, x = \text{bind}(S; v); P) :: Q, r) \mid T, W) \mid A, (\text{ANS}, \text{SNS}) \rightarrow \\
a(h, C, H, (t, (B + \{x : r' \otimes b\}, P) :: Q, r) \mid T, W) \mid A, (\text{ANS}, \text{SNS})
\]

However, sometimes the host where the agent is running is irrelevant and is not taken as consideration when the reference is picked. In both these rules, the criteria used in choosing an agent is left to the implementation.

\[
\text{[BindAny]} \quad \text{SNS}(S) = (\alpha, \{r \otimes a_1, \ldots, r \otimes a_n\}) \quad \exists r' \otimes b \in \{r \otimes a_1, \ldots, r \otimes a_n\} : b \neq a \\
a(h, C, H, (t, (B, x = \text{bind}(S; P) :: Q, r) \mid T, W) \mid A, (\text{ANS}, \text{SNS}) \rightarrow \\
a(h, C, H, (t, (B + \{x : r' \otimes b\}, P) :: Q, r) \mid T, W) \mid A, (\text{ANS}, \text{SNS})
\]
Current Host

The next rule returns the host where the agent is running.

\[
[H] \quad a(h, C, H, (t, (B, x = \text{host}() ; P) :: Q, r) | T, W) \rightarrow a(h, C, H, (t, (B + \{x : h\}, P) :: Q, r) | T, W)
\]

Local Method Invocation

Method invocation in objects can only be done within an agent. All objects are encapsulated within agents and thus, invoking a method in an object located in the address space of some other agent is not possible, unless the target agent’s interface provides some means to access the object. Methods of the agent itself can of course be invoked both from within the agent, and from other remote agents. Rule [LOCALINVOCATION] applies to the scenario of local invocations, both in objects and in agents. We guarantee this restriction by qualifying the reference with its location.

The method invocation simulates a call stack by suspending the current thread, and by creating a new one, bound to the same heap reference (t), to execute the body of the method. The environment of the new thread is obtained from the target object’s environment modified with the values assigned to the method’s parameters. The result location is a fresh heap reference \( r'' \), locked by the current thread and holding no value \((r'' : (t, \text{null}))\). The current thread is then suspended on that reference \( r'' \), waiting for the result. Its environment is modified by the binding of variable \( x \) to \( r'' \), so that \( x \) holds the returned value once the current thread resumes its execution. By associating the new thread to the same heap reference as the one that invokes the method, the former gains access to all the resources locked by the latter.

\[
[\text{LOCALINVOCATION}]
\begin{align*}
\text{evalSeq}(H, t, B, \vartheta) &= \emptyset \\
B(o) &= r'@a \\
\text{tryAccess}(H, r'@a, t) &= \text{true} \\
H(r'@a) &= (\_, (\_, B', X)) \\
\text{code}(C(X), m) &= (\_, P') \\
\vartheta &= \text{HeapRef fresh} \\
a(h, C, H, (t, (B, x = \text{o.m(\vartheta)} ; P) :: Q, r) | T, W) \rightarrow \\
a(h, C, H + \{r'' : (t, \text{null})\}, (t, (B + \{x : \text{a} \}, P'), r''), W + \{r'' : (t, (B + \{x : r''\}, P) :: Q, r)\})
\end{align*}
\]

Note that elements of the heap of the form \( r'' : (t, \text{null}) \) (with \( r \notin \text{ThreadRef} \) and \( t \neq \text{null} \)) denote uniquely references waiting for results. This will be important when encoding MOB to the target process calculus.

Rule [LOCALINVOCATIONLOCKED] states that if the object on which the method its to be invoked is locked by another thread, the current thread suspends on that object.

\[
[\text{LOCALINVOCATIONLOCKED}]
\begin{align*}
B(o) &= r'@a \\
\text{tryAccess}(H, r'@a, t) &= \text{false} \\
a(h, C, H, (t, (B, x = \text{o.m(\vartheta)} ; P) :: Q, r) | T, W) \rightarrow \\
a(h, C, H, T, W + \{r' : (t, (B, x = \text{o.m(\vartheta)} ; P) :: Q, r)\})
\end{align*}
\]

The return instruction terminates the execution of the current thread, places the result in the dedicated heap reference \( r \), releasing its lock, and spawns a notify thread to wake up the thread waiting for the result. Thus, the image of \( r \) in the heap will now hold the returned value (null, u). The notify thread will cause any thread in \( W \) waiting on \( r \) to resume, simulating the call stack.

\[
[\text{LOCALRETURN}]
\begin{align*}
\text{eval}(H, t, B, v) &= u \\
a(h, C, H, (t, (B, \text{return}(v) ; P) :: Q, r) | T, W) \rightarrow \\
a(h, C, H + \{r : (\text{null, u})\}, \text{notify}(r) | T, W)
\end{align*}
\]
Remote Method Invocation in an Agent

As in local method invocations, remote method invocations always launch a new thread \((t')\) in the target agent to execute the corresponding code\(^2\). The difference lies in the fact that the result slot of the thread, \(r''@a\), is now a heap reference from the heap of the calling agent. Moreover, this thread in the case of remote invocations, does not execute the body of the method, but rather triggers a local invocation. This allows for the application of the \([\text{LOCALINVOKE}]\) reduction rule to execute the method locally at the remote agent.

The values assigned to the method’s parameters are passed by value, except agents that are passed by reference\(^3\). A copy of the arguments must be sent to the target agent and since this may include objects, a closure with the values and the classes they use must be constructed, using function \(\text{copySeq}\). The local invocation performed at the target agent has arguments \(\exists\), bound to the clones of the original values assigned to the arguments in the calling thread.

\[
\begin{align*}
\text{eval}(H, t, B, \forall) &= u \quad \text{copySeq}(C, H, u) = (C', H', u') \\
(a(h, C, H, (t, (B, x = o.m(\forall); P) :: Q, r)) | T, W) & | \quad b(h', C', H', T, W') | A, R \rightarrow \\
(a(h, C, H + \{r'' : (t, null)\}, T, W + \{r'' : (t, (B + \{x : x''\}, P) :: Q, r)\}) | b(h', C' + C'', H' + H'', \text{launch}(\{\text{self} : r', \exists : \emptyset\}, x = \text{self}.m(\exists); \text{return} x, r''@a | T, W')) | A, R
\end{align*}
\]

The return value from a remote method invocation must be placed in a reference in the heap of the calling agent. This value may include objects, and thus a closure with the value and the classes it uses must be constructed. Finally, a \(\text{notify} \) thread is placed in the pool of threads of the calling agent, that will trigger the \([\text{NOTIFY}]\) rule and awake the thread that performed the invocation.

\[
\begin{align*}
\text{eval}(H, t, B, v) &= u \quad \text{copySeq}(C, H, u) = (C', H', u') \\
(a(h, C, H, (t, (B, \text{return}(v); P) :: Q, r@b)) | T, W) & | \quad b(h', C', H', T, W') | A, R \rightarrow \\
(a(h, C, H, T, W) | b(h', C' + C'', H' + H'' + \{r : (null, u')\}, \text{notify}(r) | T, W')) | A, R
\end{align*}
\]

Exclusive Access

Values in the heap may be shared by several threads, therefore it is necessary to supply a mechanism to ensure that a thread may gain exclusive access to a given value. The \(\text{lock} \) instruction gives exclusive access to a reference to the current thread. The operation is only allowed if no other thread has exclusive access over the reference. Note that although the inspected agent is the only one under the rule’s scope, we qualify \(r'\) with its location. This is to point out that exclusive access operations can only be performed on references owned by the agent.

\[
\begin{align*}
\text{eval}(H, t, B, x) &= r''@a \quad \text{tryLock}(H, r''@a, t) = (H', \text{true}) \\
a(h, C, H, (t, (B, \text{lock}(x); P) :: Q, r)) | T, W) & | a(h, C, H', (t, (B, P) :: Q, r)) | T, W)
\end{align*}
\]

A thread that tries to obtain the lock of a locked reference suspends on the reference.

\[
\begin{align*}
\text{eval}(H, t, B, x) &= r''@a \quad \text{tryLock}(H, r''@a, t) = (H, \text{false}) \\
a(h, C, H, (t, (B, \text{lock}(x); P) :: Q, r)) | T, W) & | a(h, C, H, T, W + \{r''@a : (t, (B, \text{lock}(x); P) :: Q, r)\})
\end{align*}
\]

\(^2\)From a practical point of view, the maximum number of threads allowed for one agent is implementation-dependent. Note that remote invocations are not anonymous, the invoking agent may be identified, since the agent’s name qualifies the reference \(r''@a\). This means that precautions to avoid abusive use from other agents may be achieved by adding a set of new preconditions to the rule. This can be used for instance to avoid denial-of-service attacks.

\(^3\)Passing them by value would constitute a new form of migration.
Instruction **unlock** returns the public access to a given heap reference and notifies every thread suspended on it, so that they may resume their execution.

\[
\text{[Unlock]}
\begin{align*}
\text{eval}(H, t, B, x) &= x' @ a \\
\text{tryUnlock}(H, x' @ a, t) &= (H', \text{true}) \\
\text{a}(h, C, H, (t, (B, \text{unlock}(x) : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H', \text{notify}(x' @ a) | (t, (B, P) :: Q, r) | T, W)
\end{align*}
\]

If a thread tries to free an object without having exclusive access to it, the operation is ignored.

\[
\text{[UnlockIgnore]}
\begin{align*}
\text{eval}(H, t, B, x) &= x' @ a \\
\text{tryUnlock}(H, x' @ a, t) &= (H, \text{false}) \\
\text{a}(h, C, H, (t, (B, \text{unlock}(x) : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, P) :: Q, r) | T, W)
\end{align*}
\]

**Control Flow**

The machine defines a basic set of instructions dedicated to control the flow of execution (**if**, **while**, and **break**). The **if** instruction requires two reduction rules, selecting the branch according to the boolean value resulting of the evaluation of value \(v\). Each of them executes the code of the selected branch followed by the instruction’s continuation (\(P\)).

\[
\text{[IfTrue]}
\begin{align*}
\text{eval}(H, t, B, v) &= \text{true} \\
\text{a}(h, C, H, (t, (B, \text{if } (v) \text{ \{ } P' \text{ \} } : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, P') :: Q, r) | T, W)
\end{align*}
\]

\[
\text{[IfFalse]}
\begin{align*}
\text{eval}(H, t, B, v) &= \text{false} \\
\text{a}(h, C, H, (t, (B, \text{if } (v) \text{ \{ } P' \text{ \} } : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, P' :: Q, r) | T, W)
\end{align*}
\]

The **while** instruction requires three rules. Rule **[PushCont]** simply pushes the continuation of the instruction to the stack. This is required to allow the use of the **break** instruction to branch out of the loop (see rule **[Break]**).

\[
\text{[PushCont]}
\begin{align*}
\text{a}(h, C, H, (t, (B, \text{while } (v) \text{ \{ } P' \text{ \} } : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, P) :: Q, r) | T, W)
\end{align*}
\]

Rule **[WhileTrue]** executes the body of the **while** instruction composed with the instruction again, performing the loop. The process eventually stops when the value \(v\) evaluates to **false**. The execution then continues with the continuation popped from the stack.

\[
\text{[WhileTrue]}
\begin{align*}
\text{eval}(H, t, B, v) &= \text{true} \\
\text{a}(h, C, H, (t, (B, \text{while } (v) \text{ \{ } P \text{ \} } : P) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, P : \text{while } (v) \text{ \{ } P \text{ \} }) :: Q, r) | T, W)
\end{align*}
\]

Rule **[WhileFalse]** emulates the end of the loop resorting to the **break** instruction.

\[
\text{[WhileFalse]}
\begin{align*}
\text{eval}(H, t, B, v) &= \text{false} \\
\text{a}(h, C, H, (t, (B, \text{while } (v) \text{ \{ } P \text{ \} }) :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, (t, (B, \text{break} :: Q, r) | T, W)
\end{align*}
\]

The **break** instruction branches out of the loop. It pops the current code-block from the stack and begins the execution of the continuation (the new top of the stack). The environment of the continuation is updated with the modifications performed during the execution of the loop.

\[
\text{[Break]}
\begin{align*}
\text{a}(h, C, H, (t, (B, \text{break} : P) :: (B', P') :: Q, r) | T, W) &\rightarrow \text{a}(h, C, H, ((B' + B) | \text{dom}(B'), P') :: Q, r) | T, W)
\end{align*}
\]
Execute External Services

The `exec` instruction allows the interaction with external services. This interaction is defined by an interface of seven possible actions.

- **init**: opens a session with a service, and returns a session identifier;
- **read**: reads a given number of bytes from a session;
- **readLine**: reads a line from a session;
- **write**: writes the given data to a session;
- **action**: posts an action to be performed by the service associated to the session;
- **isAlive**: checks if the session is still active;
- **close**: closes the session.

The syntax of the `exec` instruction requires the existence of three arguments, of which the first is the string that determines which action is to be performed. The second argument is an integer value that, when the action is `init`, corresponds to the identifier of the service to be requested, and otherwise corresponds to the session identifier. The third argument is a string used to pass values to the action. The operation is performed by an internal built-in function `exec` that executes synchronously. Asynchronous calls can be performed by encapsulating the `exec` instruction in a new thread.

The protocol to interact with an external service is initiated by the `init` action, that receives as argument an integer value that identifies the service, and returns the session identifier. Once the session is opened, a series of `read`, `readLine`, `write`, `action`, and `isAlive` actions may be performed. To terminate the session, the `close` action must be used. Below is an example of a session with a FTP server. We assume that the FTP service identifier is 4.

```
x = exec("init" 4 "ftp.adomain");
x' = exec("action" x "GET afile");
x' = exec("read" x "4096");
y = x' != "";
while (y) {
  x' = exec("read" x "4096");
y = x' != "";
}
x' = exec("close" x "")
```

The example begins by opening a FTP session with a server located at `ftp.adomain`. The correspondent session identifier is placed on `x`. Next, it posts the GET `afile` action to fetch file `afile`, and reads its contents in chunks of 4096 bytes. Once the file is read, it closes the session.

Assignment of Expressions

The result of the computation of an expression may be assigned to a variable. The assignment involves adding a new entry in the environment `(B)` of the thread.

```
x = exec("init" 4 "ftp.adomain");
x' = exec("action" x "GET afile");
x' = exec("read" x "4096");
y = x' != "";
while (y) {
  x' = exec("read" x "4096");
y = x' != "";
}
x' = exec("close" x "")
```

The example begins by opening a FTP session with a server located at `ftp.adomain`. The correspondent session identifier is placed on `x`. Next, it posts the GET `afile` action to fetch file `afile`, and reads its contents in chunks of 4096 bytes. Once the file is read, it closes the session.
Handling Attributes

Assigning a value to an attribute of an object involves modifying the object’s closure. Thus, an inspection to the status of both the object and the attribute is required. If the access to both is granted to the current thread, the binding of the given attribute in the object’s closure is modified.

\[ \text{AttrAssignmentLocked} \]

\[
\begin{align*}
\text{eval}(H, t, B, v) &= \alpha \quad B(\text{self}) = \tau' \\
H(\tau') &= (t', (\text{bool}, B', X)) \\
\text{tryAccess}(H, \tau', t) &= \text{true} \\
(\text{eval}(H, t, B', x) = \tau'' \land \text{tryAccess}(H, \tau'', t) = \text{true}) \\
\lor \text{eval}(H, t, B, x) = c
\end{align*}
\]

\[ \text{AttrAssignmentLockedInAttr} \]

\[
\begin{align*}
\text{B(self)} &= \tau' \\
\text{tryAccess}(H, \tau', t) &= \text{false}
\end{align*}
\]

\[ \text{AttrAssignmentLockedInAttr} \]

\[
\begin{align*}
\text{B(self)} &= \tau' \\
H(\tau') &= (t', (\text{bool}, B', X)) \\
\text{tryAccess}(H, \tau', t) &= \text{true} \\
(\text{eval}(H, t, B', x) = \tau'' \land \text{tryAccess}(H, \tau'', t) = \text{false})
\end{align*}
\]

If not, the current thread may suspend on the reference holding the object, or on the one holding the actual attribute. Rule [AttrAssignmentLocked] covers the first case, where the thread cannot access the object.

There is no access restriction on the reading of attributes. The rule simply retrieves the value of the attribute and binds the given variable to it. To ensure that correctness of the information to be read, the programmer must protect the access with a lock to the object.

\[ \text{ReadAttr} \]

\[
\begin{align*}
\text{B(o)} &= \tau' \\
H(\tau') &= (\_ (\_ B', X)) \\
B'(y) &= \alpha
\end{align*}
\]

Terminate an Agent

Finally, the [Exit] rule terminates the execution of an agent. This is required because agents are daemons and their execution must be explicitly terminated by the \texttt{exit} instruction. All the references to the agent in the network must be removed.

\[ \text{Exit} \]

\[
\begin{align*}
\text{B(self)} &= \alpha \\
H(\alpha) &= (\_ (\_ X)) \\
C(X) &= (\_ \_ \_ \_ S_1 \cdots S_n) \\
\text{SNS}(S_1) &= (\alpha_1, \{r@a_1, \ldots, r@a_n\}) \ldots \\
\text{SNS}(S_n) &= (\alpha_n, \{r@a_1, \ldots, r@a_n\}) \\
\text{A(ANS | dom(ANS) - (a), SNS + \{S_1 : (\alpha_1, \{r@a_1, \ldots, r@a_n\})}, \ldots, \{S_n : (\alpha_n, \{r@a_1, \ldots, r@a_n\})}))}
\end{align*}
\]

5 The Type System

In this section we present a type inference system for MOB that is very much inspired in the type-system developed by Vasco Vasconcelos for the TyCO calculus [17]. Types are ranged over by \(\alpha\), and are distinguished between types for primitive constants, ranged over by \(\rho\), types for classes and agents, types for objects, instances of agents and services, and a denumerable set of variables for types, ranged over by \(t\). A type for a class (or agent) is defined by a tuple of
two elements. The first holds the types for class (or agent) attributes, and the second, of the form $\beta$, defines the type for the interface of the class (or agent). $\beta$ types are records of the form 

$$\{ \tilde{m}_1 : (\tilde{\alpha}_1 \mapsto \alpha_1), \ldots, \tilde{m}_n : (\tilde{\alpha}_n \mapsto \alpha_n) \}$$

where $\tilde{m}_i$ denotes the identifier of a method; $\tilde{\alpha}_i$, the types of its parameters, and; $\alpha_i$, its return type.

$\alpha ::= \rho$  
Type of a primitive constant  
| $(\tilde{\alpha}, \beta)$  
Type of a class or agent  
| $\beta$  
Type of an object, an agent instance, or a service  
| $t$  
Type variable  
| $\mu t. \alpha$  
Type relational tree

$\beta ::= \{ \tilde{m}_1 : (\tilde{\alpha}_1 \mapsto \alpha_1), \ldots, \tilde{m}_n : (\tilde{\alpha}_n \mapsto \alpha_n) \}$  
Record type

$\rho ::= \text{int} | \text{string} | \text{bool} | \text{thread}$  
Primitive types

As in TyCO, types are interpreted as rational (regular infinite) trees. A type denoted by $\mu t. \alpha$ with $(\alpha \neq t)$ represents the rational solution for the equation $\alpha = t$. An interpretation of recursive types as infinite trees induces an equivalence relation on types: $\alpha \approx \alpha'$, if the tree solution for $\alpha = t$ and $\alpha' = t$ is the same.

Expressions

**Typings for expressions** are type assertions of the form $e : \alpha$, for an expression $e$, and its type $\alpha$. Expressions are formed by variables, constants, and operations over both of these. The type assignment is built from the types of constants and built-in operations and of types assigned to variables. The type of constants or a of built-in operation is given by the $\text{typeOf}$ built-in function. The later are represented as an application of the types of the arguments into the type of the operation. For example $\text{typeOf}(\text{false}) = \text{bool}$, and $\text{typeOf}(<) = \text{int} \mapsto \text{int} \mapsto \text{bool}$. For $\bar{v} = v_1 \ldots v_n$, a sequence of pairwise distinct values, and $\bar{\alpha} = \alpha_1 \ldots \alpha_n$, a sequence of types, we denote $v_1 : \alpha_1, \ldots, v_n : \alpha_n$, a sequence of type assignments as: $\bar{v} : \bar{\alpha}$.

$$[\text{CONST}] \quad \Gamma \vdash c : \text{typeOf}(c)$$

$$[\text{NULL}] \quad \Gamma \vdash \text{null} : t \quad t \text{ fresh } \wedge$$

$$t \neq \rho \in \{\text{int}, \text{bool}, \text{string}\}$$

$$[\text{GROUP}] \quad \Gamma \vdash e : \alpha \vdash (e) : \alpha$$

$$[\text{SEQN}] \quad \Gamma \vdash v_1 : \alpha_1 \ldots \Gamma \vdash v_n : \alpha_n$$

$$\Gamma \vdash \bar{v} : \bar{\alpha}$$

$$[\text{VAR}] \quad \Gamma \vdash x : \alpha \vdash x : \alpha$$

$$[\text{UNOP}] \quad \text{typeOf}(uop) = \rho_1 \rightarrow \rho_2 \quad \Gamma \vdash e : \rho_1$$

$$\Gamma \vdash uop \ e : \rho_2$$

$$[\text{BinOp}] \quad \text{typeOf}(bop) = \rho_1 \rho_2 \rightarrow \rho_3 \quad \Gamma \vdash e_1 : \rho_1 \quad \Gamma \vdash e_2 : \rho_2$$

$$\Gamma \vdash e_1 \ bop \ e_2 : \rho_3$$

21
Instructions

Type assignments for agents and services are, correspondingly, type assertions of the form $X : (\alpha, \beta)$ or $S : \beta$. Typings, denoted by $\Gamma$, is a map defined as:

$$\Gamma : \text{(Class} \cup \text{Service} \cup \text{Var}) \rightarrow \text{Type}$$

that contains the type assignments for classes, agents, services, and variables. For $\gamma$ ranging over the elements of $\text{dom}(\Gamma)$, we have that $\Gamma \setminus \gamma$ denotes the typing obtained by $\Gamma$ with its domain reduced from the elements in $\gamma$, and $\Gamma(\gamma)$ as the typing assigned to $\gamma$ if $\gamma \in \text{dom}(\Gamma)$. We also denote as $x_{\text{ret}}$, as the built-in identifier that holds the return type of a method $m$.

Type assignments for definitions, sequences of instructions and methods are thus denoted, respectively, by $\Gamma \vdash \delta$, $\Gamma \vdash P$ and $\Gamma \vdash M$. We begin by presenting the rules for service definitions and service requirement. Services are not removed from the set of typings until they are checked against the types defined for them in the network. Thus, when the local inference is done, the name resolver (R) is contacted to validate the local typings for the services.

\[
\text{[SERVICE]} \quad \frac{\Gamma \vdash \delta \quad \Gamma \vdash P}{\Gamma \vdash \text{service } S \{m_1 \ldots m_n\} \delta P} \quad (\Gamma(S) \approx \{m_1 : (\tilde{\alpha}_1 \mapsto \alpha_1), \ldots, m_n : (\tilde{\alpha}_n \mapsto \alpha_n)\})
\]

\[
\text{[REQUIRES]} \quad \frac{\Gamma \vdash \delta \quad \Gamma \vdash P}{\Gamma \vdash \text{requires } S \delta P} \quad (\Gamma(S_1) \approx \beta_1 \quad \ldots \quad \Gamma(S_n) \approx \beta_n)
\]

\[
\text{[SERVICECHECK]} \quad \frac{\{S_1 : \beta_1, \ldots, S_n : \beta_n\} \vdash \delta P \quad R = (\text{ANS}, SNS)}{\delta \vdash S P}
\]

Regarding classes and agents we define rules to type collections of methods, class and agent definitions. To allow mutual recursion between class and agent definitions we define two rules for both. One applied in the general case, and one other only applied when the definition is the last in the sequence. The later closes the system for definitions, removing them from the set of bindings by using a $defs$ function that, given an set $\Gamma$, returns a sequence of all the elements from its domain that belong to Class.

\[
\text{[METHODCOLLECTION]} \quad \frac{\Gamma \vdash x_1 : \tilde{\alpha}_1 \quad \Gamma \vdash \text{xret}_1 : \alpha_1 \quad \Gamma \vdash P_1 \quad \ldots \quad \Gamma \vdash x_n : \tilde{\alpha}_n \quad \Gamma \vdash \text{xret}_n : \alpha_n \quad \Gamma \vdash P_n}{\Gamma \setminus x_1 \ x_{\text{ret}}_1 \ \ldots \ x_n \ x_{\text{ret}}_n \vdash S \{P_1\} \ldots m_n \{P_n\} : \{m_1 : \tilde{\alpha}_1 \mapsto \alpha_1, \ldots, m_n : \tilde{\alpha}_n \mapsto \alpha_n\}}
\]

\[
\text{[CLASS]} \quad \frac{\Gamma \vdash \text{self} : \beta \quad \text{self}.x_1 : \alpha_1 \quad \ldots \quad \text{self}.x_n : \alpha_n \vdash M : \beta \quad \Gamma \vdash \delta \quad \delta \neq \epsilon}{\Gamma \vdash \text{class } X(\bar{x}) \ M \delta P}
\]

\[
\text{[CLASSISLASTDEF]} \quad \frac{\Gamma \vdash \text{self} : \beta \quad \text{self}.x_1 : \alpha_1 \quad \ldots \quad \text{self}.x_n : \alpha_n \vdash M : \beta}{\Gamma \setminus \text{defs}(\Gamma) \vdash \text{class } X(\bar{x}) \ M \ P}
\]

\[
\text{[AGENT]} \quad \frac{\Gamma \vdash \text{agent } X(\bar{x}) \ \text{provides } S \ \text{requires } S' \ M \delta P}{(\Gamma(X) \approx (\tilde{\alpha}, \beta), \forall S \in \tilde{S} : \Gamma(S) \approx \beta' \quad \implies \forall m \in \text{dom}(\beta') : m \in \text{dom}(\beta) \land \beta'(m) \approx \beta(m))}
\]

22
We now define the rules to type sequences of Mob instructions (P):

- **[Fork]** \( \Gamma \vdash x : \text{thread} \quad \Gamma \vdash P' \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = \text{fork}(P'); P \]

- **[Join]** \( \Gamma \vdash x : \text{thread} \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{join}(x); P \]

- **[Wait]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{wait}(x); P \]

- **[Notify]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{notify}(x); P \]

- **[Lock]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{lock}(x); P \]

- **[Unlock]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{unlock}(x); P \]

- **[Host]** \( \Gamma \vdash x : \text{string} \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = \text{host}(); P \]

- **[Go]** \( \Gamma \vdash v : \text{string} \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{go}(v); P \]

- **[Expr]** \( \Gamma \vdash x : \alpha \quad \Gamma \vdash e : \alpha \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = e; P \]

- **[If]** \( \Gamma \vdash v : \text{bool} \quad \Gamma \vdash P' \quad \Gamma \vdash P'' \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{if}(v) \{ P' \} \text{ else } \{ P'' \}; P \]

- **[While]** \( \Gamma \vdash v : \text{bool} \quad \Gamma \vdash P' \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{while}(v) \{ P' \}; P \]

- **[Break]** \( \Gamma \vdash P \)
  \[ \Gamma \vdash \text{break}; P \]

- **[Exit]** \( \Gamma \vdash P \)
  \[ \Gamma \vdash \text{exit}; P \]

- **[Return]** \( \Gamma \vdash v : \alpha \quad \Gamma \vdash x_{\text{ret}} : \alpha \quad \Gamma \vdash P \)
  \[ \Gamma \vdash \text{return } v; P \]

- **[Bind]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash v : \text{string} \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = \text{bind}(S,v); P \]

- **[BindAny]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = \text{bind}(S); P \]

- **[New]** \( \Gamma \vdash x : \beta \quad \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = \text{new } X(\psi); P \]

- **[MethodInv]** \( \Gamma \vdash P \)
  \[ \Gamma \setminus x \vdash x = o.m(\psi); P \]

\( S \vdash \text{agent } X(S) \vdash \text{provides } S \equiv P \)

\( \forall S \ni S(\Gamma) \equiv \beta' \implies \forall m \ni dom(\beta') : m \ni dom(\beta) \land \beta'(m) \equiv \beta(m) \)
Next, we present two simple programming examples in Mob and execute one of them in the MobAM.

6 Programming in Mob

We exemplify the syntax with two small examples. We assume that two classes were previously defined. These are Array and Map, and implement the usual operations with arrays and maps. FILEEXEC and IO are two integer constants that we also assume that were previously defined. Besides the Array and Map classes, these examples resort only to the base core Mob constructs, hence their verbosity.

The first example is that of a server and a client for a clock synchronising service (Time). The server in listing 1 provides a service that features a single method getTime() (lines 5 to 10). Note that the main method in line 4 may be empty since Mob agents run as daemons and some external action is required to terminate their execution. The program, not the launched agent, terminates with the exit instruction at line 13.

Listing 1: A time server agent

```plaintext
1 service Time { getTime }
3 agent TimeServer() provides Time {
5     main { }
7     getTime() { 
8         d = exec("init", FILEEXEC, "getTimeApplication"); // Open the session
9         x = exec("readLine", d, ""); // Read the output of the application
10        status = exec("close", d, ""); // Close the session
11        return (x); 
12    }
13 } 
14 x = new TimeServer(); // Create agent
15 exit; // Terminate program
```

The client (listing 2) requires the Time service in line 1 and, when run, takes an array of hosts and performs a cycle (lines 7 to 16) in which it moves to each of them in line 9, setting their clock according with central time from the TimeServer (lines 10 to 13). Lines 19 to 22 construct the array to be passed as argument to the instance of the agent created in line 23. The program terminates its execution in line 24.
Listing 2: A time client agent

agent TimeClient(hostList) requires Time {
  main() {
    timeServer = bind(Time); // Discover service
    iter = hostList.iterator(); // Build condition for while
    hasNext = iter.hasNext(); // For all hosts migrate and execute setTimeApplication
    while(!cond) {
      hostName = iter.next(); // Next host
      go(hostName); // Go to the next host
      time = timeServer.getTime(); // Get time from server
      command = "setTimeApplication " ^ time; // Build command to execute
      d = exec("init", FILEEXEC , command ); // Open session to execute the application
      status = exec("close", d, ""); // Close session
      cond = hasNext == true; // Build condition for while
    }
  }
  hosts = new Array(null, 0); // Construct array
  x = hosts.put("host1.net1");
  x = hosts.put("host2.net2");
  x = hosts.put("host3.net3");
  x = new TimeClient(hosts); // Create agent
  exit; // Terminate program
}

Another, slightly more complex application is a minimal Messenger service implemented in listing 3. The Messenger service, defined in line 1, provides three methods: a client may log in the system (logIn), log out from the system (logOut) or ask who is currently on-line (getLogged). Their implementation is done respectively in lines, 4 to 7, 8 to 11, and 12 to 14. The instance of the agent is created in line 17, with the map created in the line before. The programs terminates its execution in line 18.

Listing 3: A messenger server agent

service Messenger { logIn logOut getLogged }

agent MessengerServer(logged) provides Messenger {
  logIn(nickname, client) { // Log in the system
    x = logged.add(nickname, client);
    return (null);
  }

  logOut(nickname) { // Log out from the system
    x = logged.remove(nickname);
    return (null);
  }

  getLogged() { // Ask who is on-line
    return (logged);
  }

  logged = new Map(null, 0); // Create initial empty map
  x = new MessengerServer(logged); // Create agent
  exit; // Terminate program
}

The particular messenger client in this example (listing 4) first binds to the service and logs in the system (lines 7 to 9). Then, it initiates an input/output service (lines 10 and 11) and starts a loop (lines 12 to 35) in which the people currently on-line are listed (lines 13 to 23) and waits for the nickname in the input at line 24. The input triggers the creation of a new session between the client and the selected peer. The session is handled by a dedicated thread, which allows for several simultaneous conversations (lines 25 to 33). During the session, any input from the keyboard of the client is sent to the receptor (lines 29 to 33), until the "quit" keyword is typed to end the session.

A client provides the MessengerPeer service, defined in line 1, to provide the method that receives and prints remote messages (lines 38 to 41). A client is terminated by some event that invokes the close method (lines 43 to 47) that logs out from the system and halts the agent’s execution.
7 Executing an Example

We are now going to exemplify how a Mob script its executed by the MobAM. Our case studies will be the TimeServer and TimeClient examples. We will only provide a partial execution of both. We will focus on the creation of the agents and on their interaction, skipping some of the steps that are not directly related with these operations, and terminating our execution once the interaction is over.

Scripts are executed in the MobAM by encapsulating them in agents that execute their code. We assume that the Mob network target of our example is denoted by: A,R. In the sequel, we underline the components that have been altered by the application of a rule of the MobAM.

We begin by launching the TimeServer script from listing 1. We create a fresh key (a) for the agent running the script. The configuration of the network once we launch the agent is:

$$a(h, C, \text{launch}(\emptyset, x = \text{new TimeServer}(); \text{exit}(\text{null}), \emptyset) | A, R \rightarrow$$

where $C = \{\text{TimeServer} : \text{true}, \epsilon, \{\text{main} : (\epsilon, \epsilon), \text{getTime} : (\epsilon, P)\}, \emptyset, \text{Time}\}$ is the result of the application of function codeCollect to the program. Unfolding the launch macro we have:
Applying rule \([\text{NEWAGENT}]\) we create the new \(\text{TimeServer}\) agent. Since rule \([\text{NEWAGENT}]\) is somewhat extensive we present it again.

\[
\begin{align*}
\text{NEWAGENT} & \quad \text{evalSeq}(h, t, B, \overline{v}) = \overline{u} \quad \text{copySeq}_{ab}(C, H, \overline{u}) = (C', H', \overline{u}') \quad C(X) = (true, x, M, \ldots S_1 \ldots S_k) \\
\text{code}(C(X), \text{main}) = (c, P') & \quad B' = \{\text{self } r : x : \overline{d}' \} \quad b \in \text{AgentKey} \quad \text{and } r \odot b \in \text{HeapRef} \\
\text{fresh} & \quad SNS(S_1) = (\alpha_1, K_1) \\
\text{SNS}(S_k) = (\alpha_k, K_k) & \quad K_i = \{r \odot a_i | i \in \{1, \ldots, n\}\} \quad \text{and } K_k = \{r \odot a_i | i \in \{1, \ldots, n\}\}
\end{align*}
\]

\[
\begin{align*}
a(h, C, \{t : (t, \text{null})\}, (t, (\{x : r \odot b\}, \text{null}), \theta)) | A, R & \quad \rightarrow \\
b(h, C, \{r : (\{x : r \odot b\}, \text{null}), \theta) | A, (\text{ANS}, SNS) & \quad \rightarrow \\
b(h, C, \{r : (\{x : r \odot b\}, \text{null}), \theta) | A, (\text{ANS} + \{r \odot b : h\}, \text{SNS} + \{\text{Time} : (\alpha_1, r \odot b)\}) & \quad \rightarrow
\end{align*}
\]

We may now use the \([\text{EXIT}]\) rule to terminate the execution of the script (in agent \(a\)), and rule \([\text{AGENTGC}]\) to garbage collect the resulting \(0_a\) agent.

\[
\begin{align*}
0_a | b(h, C, \{r : (null, \{\text{self } r\}, \text{TimeServer})\}, \text{launch}((\text{self } r, \epsilon, \text{null}), \theta) | A, \quad & \quad \text{(ANS} + \{r \odot b : h\}, \text{SNS} + \{\text{Time} : (\alpha_1, r \odot b)\}) \equiv \\
b(h, C, \{r : (null, \{\text{self } r\}, \text{TimeServer})\}, \text{launch}(\text{self } r, \epsilon, \text{null}), \theta) | A, \quad & \quad \text{(ANS} + \{r \odot b : h\}, \text{SNS} + \{\text{Time} : (\alpha_1, r \odot b)\}) \quad \text{def}
\end{align*}
\]

From now on we will denote \((\text{ANS} + \{r \odot b : h\}, \text{SNS} + \{\text{Time} : (\alpha_1, r \odot b)\})\) as \((\text{ANS}', \text{SNS}') = R'.\) We proceed unfolding the \text{launch} macro.

\[
\begin{align*}
b(h, C, \{r : (null, \{\text{true}, \{\text{self } r\}, \text{TimeServer})\}, t' : (t', \text{null})), (t' , (\{\text{self } r\}, \epsilon, \text{null}), \theta) | A, R' & \quad \rightarrow
\end{align*}
\]

Since the code for \text{main} is empty, we may apply rule \([\text{END}]\) to terminate the thread.

\[
\begin{align*}
b(h, C, \{r : (null, \{\text{true}, \{\text{self } r\}, \text{TimeServer})\}, t' : (t', \text{null})), \text{notify}(t'), \theta) | A, R' & \quad \rightarrow
\end{align*}
\]

To notify the threads suspended on \(t'\) we apply rule \([\text{NOTIFY}]\). Note that there are no suspended threads, and thus the rule does not wake any thread.

\[
\begin{align*}
b(h, C, \{r : (null, \{\text{true}, \{\text{self } r\}, \text{TimeServer})\}, t' : (t', \text{null})), 0_t, \theta) | A, R' & \quad \rightarrow
\end{align*}
\]

We denote this network configuration as \(A', R'\), and launch the \text{TimeClient} from listing\(\![2]\) script onto it. Note that, in order to avoid using too many identifiers, we may repeat some of the ones used for references, since they are lexically bound to the agent that hosts them.

Let \(a'\) be the key for the agent executing the client script at host \(h'\), and \(P'\) be the code for the \text{main} method of the \text{TimeClient} agent. The initial state of the network with the spawning of the new agent is:
\[
\begin{align*}
& a'(h', C', \emptyset, \text{launch}(\emptyset, \text{hosts} = \text{new Array(null, 0); ...; exit(null), \emptyset}) | A', R' \quad \text{def}
\end{align*}
\]

where \( C' = C_0 + \{\text{TimeClient} : (\text{true}, \text{hostList}, \{\text{main} : (\text{C}, \text{P})\}, C_0, \epsilon)\} \) is the code repository for agent \( a' \) (result of the \text{codeCollect} function) that holds in \( C_0 \) the code for the \text{Array} class. Unfolding the \text{launch} macro we have:

\[
\begin{align*}
&a'(h', C', \{t : (t, \text{null})\}, (t, (\emptyset, \text{hosts} = \text{new Array(null, 0); ...; exit(null), \emptyset}) | A', R' \rightarrow
\end{align*}
\]

We now jump to line 23 and skip the creation of the \text{hosts} array. Assume that at the time of the creation of the agent, variables \text{hosts} is bound to a reference \( r \), that holds the entire array. We denote the heap of \( a' \) at this point as \( H \), and the set of bindings of the thread as \( B \) that contains \{\text{hosts} : \text{r}\}. In line 23 we find the creation of the agent:

\[
\begin{align*}
&a'(h', C', H, (t, (\emptyset, x = \text{new TimeClient(hosts); exit(null), \emptyset}) | A', R' \rightarrow
\end{align*}
\]

We now apply rule [NEWAGENT] to create the agent. For that we introduce a new agent key, \( b' \). The values given to the attributes of the class must be cloned. We thus perform the cloning of the reference bound to \text{hosts} and collect the code required by the methods of the \text{Array} class (of which \text{hosts} is an instance).

\[
\begin{align*}
\text{evalSeq}(H, t, B, \text{hosts}) &= r \\
\text{copySeq}_{\text{rb}}(C', H, r) &= (C_0, H', r') \\
\text{codeIn}(C', \text{main}((P')) &= \emptyset
\end{align*}
\]

\( H' \) and \( C_0 \) are respectively the heap and code closure for \( r' \). Thus, the heap of the new agent will be \( H' + \{r'' : (\text{null}, (\text{true}, \{\text{hostList} : r'\}, \text{TimeClient}))\} \), where \( r'' \) is the reference that holds the agent’s closure. Since \text{hosts} holds an array, the only code required by the reference associated to it in the new agent is the \text{Array} class, kept in the \( C_0 \) code repository. Thus, the code repository for the new agent will be \( C_0 + \{\text{TimeClient} : (\text{true}, \text{hostList}, \{\text{main} : (\epsilon, \text{P})\}, C_0, \epsilon)\} = C' \), which contains the code for \text{Array} and \text{TimeClient}. Note that the methods of class \text{TimeClient} do not require any extra code. The resulting state is:

\[
\begin{align*}
&a'(h', C', H, (t, (B + \{x : r'' @ b'\}, \text{exit}, \emptyset, \emptyset) | A', \{\text{ANS}' + \{r'' @ b' : h'\}, \text{SNS}'\} \rightarrow
\end{align*}
\]

We then terminate the execution the client script (agent \( a' \)) and garbage collect it to obtain:

\[
\begin{align*}
&b'(h', C', H' + \{r'' : (\text{null}, (\text{true}, \{\text{hostList} : r'\}, \text{TimeClient}))\}, \text{launch}((\text{self} : r''), P', \text{null}, \emptyset) | A', \{\text{ANS}' + \{r'' @ b' : h'\}, \text{SNS}'\} \quad \text{def}
\end{align*}
\]

Unfolding once again the \text{launch} macro we obtain the state of the client prior to its interaction with the \text{Time} service provider:

\[
\begin{align*}
&b'(h', C', H' + \{r'' : (\text{null}, (\text{true}, \{\text{hostList} : r'\}, \text{TimeClient}))\}, t' : (t', \text{null}), t'(\{\text{self} : r''\}, P', \text{null}, \emptyset) | A', \{\text{ANS}' + \{r'' @ b' : h'\}, \text{SNS}'\}
\end{align*}
\]

28
Remember that \( A' = b(h, C', \{ r' : (null, \{ true, \{ \self : r' \}, \emptyset, TimeServer \}), t' : (t', null) \}, O_r, \emptyset) \mid A \).
To perform the interaction between both agents we need to expose both their states from the network. To present a less extensive state we denote the heap of each agent as:

\[
H'' = \{ r' : (null, \{ true, O_r, TimeServer \}), t' : (t', null) \}
\]
\[
H''' = H' + \{ r'' : (null, \{ hostList : r', TimeClient \}), t' : (t', null) \}
\]
Thus, by using the congruence rules we can obtain the following configuration of the network with both agents exposed and associated:

\[
\begin{align*}
\text{Client} & \quad \text{Server} \\
(b(h', C', H''), (t', (\{ self : r'' \}, timeServer = bind(T), ...), null)), \emptyset) & \quad (b(h, C, H'', O_r, \emptyset)) \mid A, \\
\{ ANS' + \{ r'' @ b' : h' \}, SNS' \} & \rightarrow \\
\end{align*}
\]

We are now going to execute some of the code in \( P' \), the code for the \textbf{main} method of the client. We begin in line 3, where we apply rule \([\text{BIND}\text{ANY}]\). We know that \( \{ r @ b : h' \} \) is present in the resolver, thus \( r @ b \) is the result of the binding operation.

\[
(b(h', C, H''), (t', (\{ self : r'' \}, timeServer : r @ b), i = hostList.iterator(); ...), null)), \emptyset) \mid \]
\[
(b(h, C, H'', O_r, \emptyset)) \mid A, \{ ANS' + \{ r'' @ b' : h' \}, SNS' \} \equiv +^*
\]

Next we proceed with the execution of agent \( b' \). In order to apply rules over this agent, we have to disassociate it from agent \( b \). For that we use rule \([\text{AGENT}\text{-ASSOC}]\).

\[
(b(h', C, H''), (t', (\{ self : r'' \}, timeServer : r @ b), i = hostList.iterator(); ...), null)), \emptyset) \mid \]
\[
(b(h, C, H'', O_r, \emptyset)) \mid A, \{ ANS' + \{ r'' @ b' : h' \}, SNS' \} \equiv +^*
\]

Now we have a state from which we may execute agent \( b' \). In its code we skip the instructions until line 9, where we find a \textbf{go}. Consider that the current state is:

\[
(b(h', C, H''), (t', (B'. \textbf{go} \text{(hostName)}; ...), null)), \emptyset) \mid b(h, C, H'', O_r, \emptyset) \mid A, \{ ANS' + \{ r'' @ b' : h' \}, SNS' \} \rightarrow \\
\]

where we denote the updated bindings as \( B' \), assuming that \( B'(\text{hostName}) = h'' \). We now apply rule \([\text{GO}]\) to migrate the agent to \( h'' \):

\[
(b(h', C, H''), (t', (B'. \text{time} = \text{timeServer.getTime}(); ...), null)), \emptyset) \mid b(h, C, H'', O_r, \emptyset) \mid A, \\
\{ ANS' + \{ r'' @ b' : h' \}, SNS' \} \equiv
\]

Next, we have a method invocation on the \text{timeServer} agent. Since \( B'(\text{timeServer}) = r @ b \), and the agent running the thread is \( b' \), the invocation is remote. We have thus to apply rule \([\text{REMOTE}\text{INVOKE}]\). Neither the heap, nor the code repository of the target agent are modified, since the method as no parameters. Before applying it, we remember the rule for remote invocation.

\[
\begin{align*}
\text{evalSeq}(H, t, B, \emptyset) = \emptyset & \quad B(o) = r'' @ b \quad H'' @ b = (true, B', X)) \\
\text{copySeq}(C, H, \emptyset) = (C', H', \emptyset) & \quad r'' @ a \in \text{HeapRef fresh} \\
(a(h, C, H, t, (B, x = o.m(v)); P :: Q, r)) \mid T, W) & \quad b(h', C', H' + H'', \text{launch}(\{ self : r', x : \emptyset \}, x = \text{self.m(x)}); \text{return}(x), r'' @ a) \mid T', W')) \mid A, R \rightarrow \\
(a(h, C, H + \{ r'' : (t, null) \}), T, W + \{ r'' : (t, (B + \{ x : r'' \}), P :: Q, r) \}) \\
(b(h', C + C', H' + H'', \text{launch}(\{ self : r', x : \emptyset \}, x = \text{self.m(x)}); \text{return}(x), r'' @ a) \mid T', W')) \mid A, R
\end{align*}
\]

Before applying the rule we have to associate the agents that will take part on the communication. Thus, applying rule \([\text{AGENT\text{-ASSOC}}]\) we have:
We may now apply rule [REMOTEINVOKE] to create the new thread in the target agent, and suspend the calling thread, waiting for the result.

\[
(b'(h'', C'', h'', \langle t', (B', \text{time} = \text{timeServer.getTime()}, \text{null}) \rangle, 0) | b(h, C, H, \langle 0, \emptyset \rangle)) | A, (\text{ANS}' + \{r''\langle b' : h'' \rangle, \text{SNS}'\} \rightarrow
\]

Now we proceed the execution in agent b to execute the method. To have a state on which we can apply rules over \(b\), we have to disassociate the agents and commute their position.

\[
b(h, C, H, \text{launch}(\langle \text{self} : r, \text{x} = \text{self.getTime()} : \text{return}(x).r''\langle b' \rangle : \langle 0, \emptyset \rangle \rangle) | A, (\text{ANS}' + \{r''\langle b' : h'' \rangle, \text{SNS}'\} \equiv \text{def}
\]

From now on we denote \((\text{ANS}' + \{r''\langle b' : h'' \rangle, \text{SNS}'\})\) as \(R''\). Unfolding \text{launch} we create a new thread associated to a new thread reference \(t''\).

\[
b(h, C, H, \langle t'' : \langle \text{null} \rangle \rangle, \langle t'', (\langle \text{self} : r, \text{x} = \text{self.getTime()} : \text{return}(x).r''\langle b' \rangle \rangle : \langle 0, \emptyset \rangle \rangle \rangle) | A, R''
\]

We now apply rule [LOCALINVOKE] to perform the local invocation in \(b\). We know that:

\[
H''(r) = (\text{null}, (\text{true}, \emptyset, \text{TimeServer}))
\]

\[
C(\text{TimeServer}) = (\text{true}, c, (\langle \text{main} : (c, \emptyset), \text{getTime} : (c, P) \rangle, \emptyset, \text{Time})
\]

Thus, \(P\) is the code to execute. We create a new reference \(r'\) to hold the result of the method, suspend the current thread on it, and create a new thread, associated to the same reference as the current \(t''\), to execute the method.

\[
b(h, C, H, \langle t'' : \langle \text{null} \rangle \rangle, \langle t'', (\langle 0, \emptyset, P, r' \rangle, \langle r' : \langle t'', (\langle x : r' \rangle, \text{return}(x).r''\langle b' \rangle \rangle \rangle \rangle) \rangle | A, R''
\]

The body of the method \(P\) executes until it reaches a return instruction, that returns the value for the time \((c)\). Consider that \(B''\) are the updated bindings of the thread executing \text{getTime}, and that \(B''(x) = c\).

\[
b(h, C, H, \langle t'' : \langle \text{null} \rangle \rangle, \langle t'', (\langle 0, \emptyset, P, r' \rangle, \langle r' : \langle t'', (\langle x : r' \rangle, \text{return}(x).r''\langle b' \rangle \rangle \rangle \rangle) \rangle | A, R''
\]

We will now apply rule [RETURN]. The value is placed on the correspondent reference.

\[
b(h, C, H, \langle t'' : \langle \text{null} \rangle \rangle, \langle t'', (\langle 0, \emptyset, c \rangle, \langle \text{notify} : (r' : \langle t'', (\langle x : r' \rangle, \text{return}(x).r''\langle b' \rangle \rangle) \rangle) \rangle | A, R''
\]

\[
b(h, C, H, \langle t'' : \langle \text{null} \rangle \rangle, \langle t'', (\langle 0, \emptyset, c \rangle, \langle \text{notify} : (r' : \langle t'', (\langle x : r' \rangle, \text{return}(x).r''\langle b' \rangle \rangle) \rangle) \rangle | A, R''
\]
Applying rule [NOTIFY]. Note that, from the definition of run in section \[4\] we have that:

\[
\text{run}((\{\cdot\}, (\{x: r'\}, \text{return}(x)), r''@b')) = \langle t'', (\{x: r'\}, \text{return}(x)), r''@b'\rangle
\]

The resulting state is thus:

\[
b(h, C, H' + \{t'': (t'', \text{null}), r': (\text{null}, c)\}, t'', (\{x: r'\}, \text{return}(x)), r''@b'), \emptyset) |
\]

\[
b'(h'', C', H'' + \{r'': (\text{null}, c)\}, 0_r, \{r'': \langle (t', (B + \{\text{time: } r''\}, \text{command = ...}), \text{null})\rangle\}) |
\]

\[
A. R'' \equiv \rightarrow
\]

To return the result back to the calling agent we have to apply rule [REMOTE\text{RETURN}]. Before we need to associate the agents, we skip over this part. The value to be placed in the heap of the calling agent b', the target of \text{return}, must be a clone of the value in the heap of b. The operation is performed by the \text{copy} function.

\[
eval[h'' + \{t'': (t'', \text{null}), r': (\text{null}, c)\}, t'', (\{x: r'\}, x) = c
\]

\[
\text{copy}_{wb}(h, H + \{t'': (t'', \text{null}), r': (\text{null}, c)\}, c) = (\emptyset, \emptyset, c)
\]

Thus, we obtain:

\[
b(h, C, H' + \{t'': (t'', \text{null}), r': (\text{null}, c)\}, 0_r, \emptyset) |
\]

\[
b'(h'', C', H'' + \{r'': (\text{null}, c)\}, \text{notify}(r''), \{r'': \langle (t', (B + \{\text{time: } r''\}, \text{command = ...}), \text{null})\rangle\}) |
\]

\[
A. R'' \equiv \rightarrow
\]

We may now apply rule [NOTIFY] to wake the calling thread. Remember that in order to apply the rule on agent b' we need to disassociate and commute the agents.

\[
b'(h'', C', H'' + \{r'': (\text{null}, c)\}, (t', (B + \{\text{time: } r''\}, \text{command = ...}), \text{null}), \emptyset) |
\]

\[
b(h, C, H' + \{t'': (t'', \text{null}), r': (\text{null}, c)\}, 0_r, \emptyset) \mid A. R'' \rightarrow
\]

Thus, the thread that performed the method invocation resumes its execution, with a binding for the value (c), through reference r''''. We conclude here the partial execution of our example. The server agent enters a state of idleness waiting for new requests, while the agent will execute the \text{setTimeApplication} binary program to set the local clock.

8 Conclusions

In this report we have presented the syntax and semantics for the core of a language for programming mobile agents, named MOB. A new report, focusing on the encoding of the semantics of this language into a process calculus is currently being prepared.

The MOB core-language compiler and run-time system are implemented. The first is an implementation of the encoding to be presented in the next report, and the second an extension to the distributed TyCO run-time to allow the execution of MOB computations.

The run-time is currently being extended with primitives for interaction with external services. This will allow MOB to act as a coordination language for mobile agents that interact with web services for: recognition/execution of programs in several high-level languages, building itineraries through external search engines, database transactions, and network communication through known protocols, such as SMTP, FTP, or HTTP.

Future plans also include an integrated tool for programming, debugging and monitoring agents.
References

[1] Lorenzo Bettini, Rocco de Nicola, and Rosario Pugliese. X-Klaim and Klava: Programming Mobile Code. In TOSCA 2001, volume 62. Elsevier Science, 2001.

[2] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In First International Symposium on Agent Systems and Applications (ASA’99)/Third International Symposium on Mobile Agents (MA’99), Palm Springs, CA, USA, October 3-6, pages 22–29. IEEE Computer Society, 1999.

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

[4] Robert S. Gray. 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), Baltimore, MD, USA, November 28 - December, 1995.

[5] Kohei Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In European Conference on Object-Oriented Programming (ECOOP’91), Geneva, Switzerland, July 15-19, volume 512 of Lecture Notes in Computer Science, pages 141–162. Springer-Verlag, 1991.

[6] Markus Horstmann and Mary Kirtland. DCOM Architecture. Microsoft, Redmond, WA, USA, 1997.

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

[8] Robin Milner, Joachim Parrow, and David Walker. A Calculus of Mobile Processes (parts I and II). Information and Computation, 100(1):1–77, 1992.

[9] Object Management Group. The Common Object Request Broker: Architecture and Specification. Object Management Group, Needham, MA, USA, 2001.

[10] Openwings. Openwings: A Service-Oriented Component Architecture for Self-Forming, Self-Healing, Network-Centric Systems (Rev 2.0). http://www.openwings.org, 2001.

[11] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In Proceedings of the First International Workshop on Mobile Agents (MA’97), Berlin, Germany, April 7-8, volume 1219 of Lecture Notes in Computer Science, pages 316–323. Springer-Verlag, 1997.

[12] David Platt. Introducing Microsoft .NET, Third Edition. Microsoft Press, Redmond, WA, USA, 2003.

[13] António Ravara, Ana Matos, Vasco Vasconcelos, and Luís Lopes. Lexically Scoped Distribution: What You See Is What You Get. In FGC: Foundations of Global Computing, volume 85(1) of Electronic Notes in Theoretical Computer Science. Elsevier Science, 2003.

[14] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute: High-level Programming Language Design for Distributed Computation. In ICFP ’05: Proceedings of The 10th ACM SIGPLAN International Conference on Functional Programming, Tallinn, Estonia, September 26-28, pages 15–26. ACM Press, 2005.

[15] Gert Smolka. Concurrent Constraint Programming Based on Functional Programming. In Chris Hankin, editor, Programming Languages and Systems, volume 1381 of Lecture Notes in Computer Science, pages 1–11. Springer-Verlag, 1998.
[16] Markus Straßer, Joachim Baumann, and Fritz Hohl. Mole - A Java Based Mobile Agent System. In M. Muehlhaeuser, editor, Special Issues in Object Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP 96), Linz, Austria, July 8-12, pages 301–308, Heidelberg, 1997. dpunkt.verlag.

[17] Vasco Vasconcelos. A Process Calculus Approach to Typed Concurrent Objects. PhD thesis, Keio University, 1994.

[18] Vasco Vasconcelos, Luís Lopes, and Fernando Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In Workshop on High Level Programming Languages (HLCL’98), Nice, France, September 12, volume 16(3) of Electronic Notes in Theoretical Computer Science, pages 19–34. Elsevier Science, 1998.

[19] Jim Waldo. The Jini Architecture for Network-Centric Computing. Communications of the ACM, 42(7):76–82, 1999.

[20] James E. White. Telescript Technology: Scenes from the Electronic Marketplace. General Magic White Paper, General Magic edition, 1995.

[21] Pawel T. Wojciechowski and Peter Sewell. Nomadic Pict: Language and Infrastructure Design for Mobile Agents. IEEE Concurrency, 8(2):42–52, 2000.