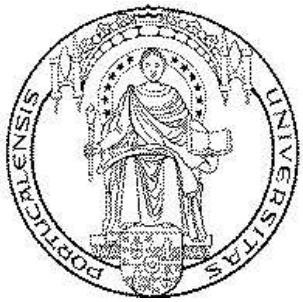
The Mob Core Language and Abstract Machine

Hervé Paulino Luís Lopes

Technical Report Series: DCC-2005-05



Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

---

Universidade do Porto

Rua do Campo Alegre, 823 4150-180 Porto, Portugal

Tel: +351+226078830 – Fax: +351+226003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

The Mob Core Language and Abstract Machine

Hervé Paulino * Luís Lopes †

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 a series of two technical reports we present a scripting language for programming mobile agents called Mob. In this first report we describe the language's syntax and semantics. 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

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 [6], Mole [13] 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 [9], 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 [14], X-Klaim [1], Nomadic Pict [17],

*DI - FCT, Universidade Nova de Lisboa, e-mail : herve@di.fct.unl.pt
†DCC - FC, Universidade do Porto, e-mail: lblopes@ncc.up.pt

Acute [10] and Alice [12]. 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 semantics of a subset of the Mob core language is *sound* relative to distributed form of the TyCO process calculus (TYped Concurrent Objects) [14, 11]. This subset contains all the primitives of Mob except the migration primitive (**go**). The encoding of the Mob abstract machine into TyCO also provides the specification for the Mob compiler. High-level constructs are obtained as derived constructs from the core language, thus preserving the semantics;
- compiled Mob programs are executed using the MIL (Multi-threaded Intermediate Language [15]) virtual machine, which is itself 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 The Mob Language and Semantics

Before describing the language and its semantics, we give a short overview of our main design goals. 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 the object-oriented programming paradigm.

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 and requires services to/from that network. Agents are handled in Mob programs much like objects in Object-Oriented languages, introducing constructors **agent**, to define an agent abstraction, and **new** 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 data-structures associated. In Mob, objects are first class entities and are created with the constructor **new**.

The *service* is the main abstraction of the language. Agents exist to implement services and do useful work by using services provided by other agents. There is absolutely 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 service. The types inferred by the MOB compiler are matched with those assumed for the service in the name service. If the agent implements a non-existing 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 remote method invocation.

To access a service, a programmer is required to get a binding for an agent that provides it. The binding is obtained dynamically by asking the network name service 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 [16]). The movement of an agent involves moving an entire virtual machine and its state to the target host in the network. The execution resumes on arrival at the target host, without external intervention.

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

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.

2.1 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 of the core language.

2.2 The Abstract Syntax

Reserved Words

The language defines a set reserved words that may not be used as identifiers:

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

Constant Identifiers

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

Identifiers

Identifiers are defined by the regular expression: $[a-zA-Z]$ $([a-zA-Z] \mid [0-9] \mid \_)^*$. They are divided in the following categories:

x, y	$\in$	Var	variables
X, Y	$\in$	ClassId	class identifiers
A, B	$\in$	AgentId	agent identifiers
s	$\in$	ServiceId	service identifiers
m	$\in$	MethodId	method labels

A sequence of elements of a given syntactic category α is written as $\tilde{\alpha}$.

Grammar

The syntax of a Mob program resorts to the following phrase classes:

ι	$\in$	Instruction	program instruction
V	$\in$	AssignValue	assignable value
M	$\in$	Methods	method implementation
e	$\in$	Expression	basic expression
c	$\in$	Constant	constant value
v	$\in$	LangValue	variable or constant
o	$\in$	Target	object or agent
bop	$\in$	BinOp	binary operator
uop	$\in$	UnOp	unary operator

A Mob program is syntactically a sequence of instructions (P) defined by the grammar in figure 1:

2.2.1 Syntactic Restrictions

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

- definitions of classes, agents and services, can only appear at top-level;
- an agent class 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 an **if** or a **while** instruction;
- the **go** and **exit** instructions can only appear inside the body of an agent definition's method;
- the method identifiers m_i in $\{m_1(\tilde{x}_1)\ \{\ P_1\ \} \ldots m_n(\tilde{x}_n)\ \{\ P_n\ \}\}$ are pairwise distinct;
- the parameters $\tilde{x}$ in an agent (**agent** $A(\tilde{x})$ **implements** $\tilde{s}$ **requires** $\tilde{s}$ M), a class (**class** $X(\tilde{x})$ M) or a method ($m(\tilde{x})\{P\}$) definitions are pairwise distinct;

P	::=	$\iota; P$	Sequential Composition
	|	ϵ	Empty Sequence
ι	::=	**agent** $A(\tilde{x})$ **implements** $\tilde{s}$ **requires** $\tilde{s}$ M	Agent Definition
	|	**service** s $\{\tilde{m}\}$	Service Definition
	|	**class** $X(\tilde{x})$ M	Class Definition
	|	**go** (v)	Agent Movement
	|	**return** (v)	Method Return
	|	**join** (x)	Thread Synchronization
	|	**lock** (x)	Lock Resource
	|	**unlock** (x)	Unlock Resource
	|	**if** (v) { P } **else** { P }	Conditional Execution
	|	**while** (v) { P }	Iterator
	|	**break**	Break
	|	**exit** ()	Terminate Agent Execution
	|	$o = V \quad \mid \quad o.x = v$	Assignment
V	::=	**new** A $(\tilde{v})$	New Agent
	|	**new** X $(\tilde{v})$	New Object
	|	**fork** { P }	New Thread
	|	**bind** $(s$ $v)$	Agent Discovery
	|	**host** ()	Current Host
	|	**exec** $(\tilde{v})$	External Call
	|	$o.m$ $(\tilde{v})$	Method Call
	|	e	Expressions
M	::=	$\{m_1(\tilde{x}_1)$ { P_1 } $\ldots m_n(\tilde{x}_n)$ { P_n } $\}$	Methods
e	::=	$v \quad \mid \quad e\ bop\ e \quad \mid \quad uop\ e$	Basic Expressions
v	::=	$o \quad \mid \quad o.x \quad \mid \quad c$	Language Values
o	::=	x | **self**	Target
c	::=	$bool$ | int | str | **null**	Constants
bop	::=	$+ \quad \mid \quad - \quad \mid \quad * \quad \mid \quad / \quad \mid \quad \% \quad \mid \quad == \quad \mid \quad !=$	
	|	$< \quad \mid \quad > \quad \mid \quad <= \quad \mid \quad >= \quad \mid \quad \&\& \quad \mid \quad \|\|$	Binary Operators
uop	::=	$! \quad \mid \quad -$	Unary Operators

Figure 1: Syntax of the Mob programming language.

- a variable x is bound in P with an assignment ($x = V; P$); is bound in M in an agent definition (**agent** $A(\tilde{x})$ **implements** $\tilde{s}$ **requires** $\tilde{s}$ M) or class definition (**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 A is bound in M and in P with a statement **agent** $A(\tilde{x})$ **implements** $\tilde{s}$ **requires** $\tilde{s}$ $M; P$;
- a class identifier X is bound in M and in P with a statement **class** $X(\tilde{x})$ $M; P$;
- a service identifier s is bound in P with a statement **service** s $\{\tilde{m}\}; P$;

- 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.

2.3 Semantics

We provide the semantics for a Mob network in the form of an abstract 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, agents and threads we first introduce some notation, syntactic categories and auxiliary functions:

Notation

In the sequel we use the following notation:

- $\emptyset$: the empty set;
- ϵ : the empty sequence;
- X^* : the set $\{\} \cup X \cup X^2 \cup X^3 \cup \ldots$;
- 2^X : the set of subsets of X;
- $\alpha \in \alpha_1, \ldots, \alpha_n$: we abusively say that an element of a given category α belongs to a sequence of elements of that same category $\alpha_1, \ldots, \alpha_n$ if $\alpha \in \{\alpha_1, \ldots, \alpha_n\}$.
- $f + g$: the finite map f *modified* by g with domain $\mathrm{dom}(f) \cup \mathrm{dom}(g)$ and values

$$(f + g)(x) = \text{ if } x \in \mathrm{dom}(g) \text{ then } g(x) \text{ else } f(x).$$

- a stack of elements of a given category α is defined as: $\mathit{Stack}(\alpha) ::= \alpha :: \mathit{Stack}(\alpha) \mid \epsilon$.
- a pool of concurrently executing units defined by some given category α is defined as $\mathit{Pool}(\alpha) = \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$, where the commutative operator $\mid$ represents concurrent execution.

Syntactic Categories

The syntactic categories involved in the abstract machine are:

a, b	$\in$	AgentName	agent name
h	$\in$	Host	host identifier
c	$\in$	Constant	constant value
x, y	$\in$	Var	variable
X, Y	$\in$	ClassId	class identifier
A, B	$\in$	AgentId	agent identifier
D	$\in$	DefinitionId = AgentId $\cup$ ClassId	agent or class identifier
m	$\in$	MethodId	method label
s	$\in$	ServiceId	service identifier
ι	$\in$	Instruction	MOB instruction
P	$\in$	Instruction*	sequence of MOB instructions
C	$\in$	Code	code for the class and agent definitions
M	$\in$	Method	methods of a definition
H	$\in$	Heap	address space for the agent
r	$\in$	HeapRef	heap reference
K	$\in$	Closure	closure
u	$\in$	Value	constant value or a heap reference
T	$\in$	RunningThread	flow of execution
B	$\in$	Environment	environment of a thread
Q	$\in$	Stack	stack of interrupted blocks of code
S	$\in$	SuspendedThread	threads suspended on a heap reference
$\mathcal{A}$	$\in$	Agent	agent
t	$\in$	Type	type of a service
$\mathcal{R}$	$\in$	NameService	name service
Ans	$\in$	AgentNameService	agent name service
Sns	$\in$	ServiceNameService	service name service
$\mathcal{N}$	$\in$	Network	network

The Data Structures

The abstract machine has two layers: agents and threads. A network is described as a set of agents running concurrently plus a name service for agents. 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. Agents interact with the network by spawning new agents or moving between hosts. They interact with other agents by invoking methods.

- an *Agent Name* is an element of the set AgentName, ranged over by a, b, and represents a unique, network-wide, identifier for an agent;
- a *Host* is an element of the set Host $\subset$ String, ranged over by h, and represents a unique, network-wide, host identifier;
- an *Instruction* is an element of the set Instruction, ranged over by ι, and represents a MOB instruction. A sequence of instructions is denoted by P;
- *Method* represents a set of methods and is a map of the form Method = MethodId $\mapsto$ Var* $\times$ Instruction*, ranged over by M, and represents the methods in a class or agent;
- the *Code for the Agent* is a map defined as Code = DefinitionId $\mapsto$ Var* $\times$ Method $\times$ ServiceId*, ranged over by C and represents the code of all the classes required for the execution of an agent;
- a *Heap Reference* is an element of the set 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

identifier 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 $\bullet \in \text{HeapRef}$ represents a null heap reference;

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

- a *Value* is an element of the set Value $=$ Constant $\cup$ HeapRef, ranged over by u;

- an *Environment* is a map defined as Environment $=$ Var $\mapsto$ Value, ranged over by B, that represents a map from identifiers in the code to 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 Closure $=$ Environment $\times$ DefinitionId, ranged over by K, and represents the closure for an instance of a class or an agent located in an address space.

- a *Heap* is a map defined as Heap $=$ HeapRef $\mapsto$ HeapRef $\times$ (Closure $\cup$ 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. Hence the inclusion of a heap reference associated with each value indicating the thread who holds the lock to the contents;

- a *Stack* is an element of the set Stack $=$ *Stack*(HeapRef $\times$ Environment $\times$ Instruction*), ranged over by Q, and represents the stack of interrupted blocks of code, that will resume their execution as soon as the current sequence of instructions terminates. It is used as a call stack for local method invocations (within an agent) and to implement control-flow instructions such as **break**, **if** and **while**. We refer to elements of the stack as *code-blocks*;

- a *Running Thread* is an element of the set RunningThread $=$ HeapRef $\times$ Environment $\times$ Instruction* $\times$ *Stack*, ranged over by T, and represents a flow of execution. In the definition of a thread (r, B, P, Q), the reference r is a location in the heap to which the thread is bound. It is used to synchronize multiple threads and to indicate where the result of a remote method invocation must be placed;

- a *Suspended Thread* is a map defined as SuspendedThread $=$ HeapRef $\mapsto$ RunningThread, ranged over by S, and represents threads suspended on heap references;

- an *Agent* is an element of the set Agent $=$ AgentName $\times$ Host $\times$ Code $\times$ Heap $\times$ Pool(RunningThread) $\times$ SuspendedThread, ranged over by $\mathcal{A}$, and represents a multi-threaded autonomous computation. We write an agent (a, h, C, H, T, S) as $a(h, C, H, T, S)$ thus exposing the agent's name;

- a *Service Type* is a element of the set Type, ranged over by t.

- an *Name Service*, $\mathcal{R}$, is composed by two maps, NameService $=$ AgentNameService$\times$ ServiceNameService. The first, defined as AgentNameService $=$ HeapRef $\mapsto$ Host, ranged over by *Ans*, represents a network-wide name service for locating agents. The second, defined as ServiceNameService $=$ ServiceId $\mapsto$ Type $\times$ 2^{HeapRef}, ranged over by *Sns*, represents a network-wide name service for obtaining the type of a service and the set of agents that implement it, denoted by I;

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

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 a reference r.

$$\mathsf{tryAccess} : \text{Heap} \times \text{HeapRef} \times \text{HeapRef} \mapsto \text{Bool}$$

$$\mathsf{tryAccess}(H, r', r) = \begin{cases} \textbf{true}, & \text{if } H(r') = (r, v) \text{ or } H(r') = (\bullet, v) \\ \textbf{false}, & \text{if } H(r') = (r'', v) \text{ and } r'' \neq r \end{cases}$$

Function tryLock tries to grant the lock for a value located at r' to a thread identified by r in a heap H. 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.

$$\mathsf{tryLock} : \text{Heap} \times \text{HeapRef} \times \text{HeapRef} \mapsto \text{Heap} \times \text{Bool}$$

$$\mathsf{tryLock}(H, r', r) = \begin{cases} (H + \{r' : (r, v)\}, \textbf{true}), & \text{if } \mathsf{tryAccess}(H, r', r) = \textbf{true} \\ (H, \textbf{false}), & \text{if } \mathsf{tryAccess}(H, r', r) = \textbf{false} \end{cases}$$

$$\mathsf{tryUnlock} : \text{Heap} \times \text{HeapRef} \times \text{HeapRef} \mapsto \text{Heap} \times \text{Bool}$$

$$\mathsf{tryUnlock}(H, r', r) = \begin{cases} (H + \{r' : (\bullet, v)\}, \textbf{true}), & \text{if } \mathsf{tryAccess}(H, r', r) = \textbf{true} \\ (H, \textbf{false}), & \text{if } \mathsf{tryAccess}(H, r', r) = \textbf{false} \end{cases}$$

Function code returns the code for a method m of an object (agent), given an the representation of the object's (agent's) class.

$$\mathsf{code} : (\text{Var}^* \times \text{Method} \times \text{ServiceId}^*) \times \text{MethodId} \mapsto \text{Var}^* \times \text{Instruction}^*$$

$$\mathsf{code}((\tilde{x}, M, \tilde{s}), m) = M(m)$$

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 r, and its environment B. The evaluation is fairly standard, however a few particularities require some closer attention: the **null** language constant is evaluated to $\bullet$ (a null reference in the machine); a variable may contain a heap reference whose value is another reference, which forces the resolution of the indirection; and finally, when the evaluation of an expression accesses a heap reference locked by some other thread, the function is recursively called until the value may be accessed.

$$\mathsf{evalSeq} : (\text{Heap} \times \text{HeapRef} \times \text{Environment} \times \text{Expression}^*) \mapsto \text{Value}^*$$

$$\mathsf{evalSeq}(H, r, B, v\ \tilde{v}) = \mathsf{eval}(H, r, B, v)\ \mathsf{evalSeq}(H, r, B, \tilde{v})$$
$$\mathsf{evalSeq}(H, r, B, \epsilon) = \epsilon$$

$$\mathsf{eval} : (\text{Heap} \times \text{HeapRef} \times \text{Environment} \times \text{Expression}) \mapsto \text{Value}$$

$$\mathsf{eval}(H, r, B, c) = \mathsf{val}(c) \qquad \text{where } \mathsf{val}(\textbf{null}) = \bullet$$

$$\mathsf{eval}(H, r, B, x) = val(c) \qquad \text{if } B(x) = c$$
$$\mathsf{eval}(H, r, B, x) = r' \qquad \text{if } B(x) = r',\ \mathsf{tryAccess}(H, r', r) = \textbf{true} \text{ and } H(r') = (\_, K)$$
$$\mathsf{eval}(H, r, B, x) = u \qquad \text{if } B(x) = r',\ \mathsf{tryAccess}(H, r', r) = \textbf{true} \text{ and } H(r') = (\_, u)$$
$$\mathsf{eval}(H, r, B, x) = \mathsf{eval}(H, r, B, x) \qquad \text{if } B(x) = r' \text{ and } \mathsf{tryAccess}(H, r', r) = \textbf{false}$$

$$\mathsf{eval}(H, r, B, x.y) = \mathsf{eval}(H, r, B', y)$$
$$\quad \text{if } B(x) = r',\ \mathsf{tryAccess}(H, r', r) = \textbf{true} \text{ and } H(r') = (\_, (B', \_))$$
$$\mathsf{eval}(H, r, B, x.y) = \mathsf{eval}(H, r, B, x.y)$$
$$\quad \text{if } B(x) = r' \text{ and } \mathsf{tryAccess}(H, r', r) = \textbf{false}$$

$$\mathsf{eval}(H, r, B, e\ bop\ e') = \mathsf{bop}(u, u')$$
$$\quad \text{if } \mathsf{eval}(H, r, B, e) = u \text{ and } \mathsf{eval}(H, r, B, e') = u'$$
$$\mathsf{eval}(H, r, B, uop\ e) = \mathsf{uop}(u) \qquad \text{if } \mathsf{eval}(H, r, B, e) = u$$

Function $\mathsf{copySeq}_{ab}$ 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 origin agent a, and the sequence of values to be copied $\tilde{u}$. The copy of each value is computed by function copy_{ab}.

$$\mathsf{copySeq}_{ab} : \text{Code} \times \text{Heap} \times \text{Value}^* \mapsto \text{Code} \times \text{Heap} \times \text{Value}^*$$

$$\mathsf{copySeq}_{ab}(C, H, u\ \tilde{u}) = (C' + C'', H' + H'', u'\tilde{u}')$$
$$\quad \text{where } (C', H', u') = \mathsf{copy}_{ab}(C, H, u) \text{ and where } (C'', H'', \tilde{u}') = \mathsf{copySeq}_{ab}(C, H, \tilde{u})$$
$$\mathsf{copySeq}_{ab}(C, H, \epsilon) = (\emptyset, \emptyset, \epsilon)$$

$$\mathsf{copy} : \text{Code} \times \text{Heap} \times \text{Value} \mapsto \text{Code} \times \text{Heap} \times \text{Value}$$

$$\mathsf{copy}_{ab}(C, H, r@a) = (\{X : C(X)\} + C', H' + \{r'@b : (\bullet, (\{\tilde{x} : \tilde{u}'\}, X)\}, r'@b)$$
$$\quad \text{if } H(r@a) = (\_, (\{\tilde{x} : \tilde{u}\}, X)) \text{ where } (C', H', \tilde{u}') = \mathsf{copySeq}_{ab}(C, H, \tilde{u}), r'@b \text{ fresh}$$
$$\mathsf{copy}_{ab}(C, H, r@a) = (C', H' + \{r'@b : u'\}, r'@b)$$
$$\quad \text{if } H(r@a) = (\_, u) \text{ where } (C', H', u') = \mathsf{copy}_{ab}(C, H, u), r'@b \text{ fresh}$$
$$\mathsf{copy}_{ab}(C, H, r@c) = (\emptyset, \emptyset, r@c) \quad \text{where } c \neq a$$
$$\mathsf{copy}_{ab}(C, H, c) = (\emptyset, \emptyset, c)$$

The run function places a set of threads in concurrent execution.

$$\mathsf{run} : \text{RunningThread} \mapsto \text{Pool(RunningThread)}$$

$$\mathsf{run}(\{T_1, T_2, \ldots, T_n\}) = T_1 \mid \mathsf{run}(\{T_2, \ldots, T_n\})$$
$$\mathsf{run}(\{T\}) = T$$

The Initial and Final States

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

$\mathcal{N}$	::=	$\mathcal{A}, \mathcal{R}$	Network
$\mathcal{A}$	::=	$\mathcal{A} \mid \mathcal{A}$	Concurrent Agents
	$\mid$	$a(h, C, H, T, S)$	Running Agent
	$\mid$	$\mathbf{0}_{\mathcal{A}}$	Terminated Agent
T	::=	$T \mid T$	Concurrent Threads
	$\mid$	(r, B, P, Q)	Running Thread
	$\mid$	$\mathbf{0}_T$	Terminated Thread

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 is constant and denoted $\mathcal{H} \subset$ Host. 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 set of agents $\mathcal{A}$ running concurrently and distributed among the network nodes in the set $\mathcal{H}$:

$$\mathcal{A}, \mathcal{R}$$

We launch a program in the network by encapsulating its code (P) in an agent, that is placed in a host specified by the user. Thus, the initial state of execution will be:

$$a(h, \emptyset, \emptyset, \mathsf{launch}(\emptyset, P), \emptyset) \mid \mathcal{A}, \mathcal{R}$$

where a is a fresh agent identifier, h is the local host and $\mathsf{launch}(B, P)$ is a macro that creates a new thread with environment B and code P (see rule [ThreadLaunch] below).

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

Agents are daemons by default and must be explicitly terminated by the **exit** instruction, which produces the $\mathbf{0}_{\mathcal{A}}$ state. Thus, at the end of the program running in agent a, the configuration of the network will be of the form:

$$\mathbf{0}_{\mathcal{A}} \mid \mathcal{A}', \mathcal{R}'$$

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

$$\mathcal{A}', \mathcal{R}'$$

2.4 The Congruence Rules

Following Milner [7] we divide the computation rules in *structural congruence rules* and *reduction rules*. The structural congruence rules allows for the re-writing of the state of an agent into a semantically equivalent one, until a reduction rule may be applied. Thus, we define $\equiv$ has been the smallest congruence relation over a MOB agent, defined by the following rules:

[AGENTSWAP]

$$\mathcal{A} \mid \mathcal{A}' \;\equiv\; \mathcal{A}' \mid \mathcal{A}$$

[AGENTALPHA]

$$\frac{\mathcal{A} \;\equiv_\alpha\; \mathcal{A}'}{\mathcal{A} \;\equiv\; \mathcal{A}'}$$

[AGENTGC]

$$\mathbf{0}_{\mathcal{A}} \mid \mathcal{A} \;\equiv\; \mathcal{A}$$

The same holds for threads within an agent and thus we have rules below. Notice that no garbage collection rule is required, since the reduction rules collect all terminated threads.

[THREADSWAP]

$$a(h, C, H, T \mid T', S) \mid \mathcal{A} \;\equiv\; a(h, C, H, T' \mid T, S) \mid \mathcal{A}$$

[THREADALPHA]

$$\frac{T \;\equiv_\alpha\; T'}{a(h, C, H, T, S) \mid \mathcal{A} \;\equiv\; a(h, C, H, T', S) \mid \mathcal{A}}$$

[THREADTERM]

$$a(h, C, H, (r, B, \epsilon, \epsilon) \mid T, S) \mid \mathcal{A} \;\equiv\; a(h, C, H, \mathsf{notify}(r) \mid T, S) \mid \mathcal{A}$$

[THREADLAUNCH]

$$a(h, C, H, \mathsf{launch}(B, P) \mid T, S) \mid \mathcal{A} \;\equiv\; a(h, C, H + \{r : (r, \bullet)\}, (r, B, P, \epsilon) \mid T, S) \mid \mathcal{A} \qquad r \text{ fresh}$$

Rules [THREADTERM] and [THREADLAUNCH] define the notify and launch macros that are simply a mechanism to allow a more expressive writing of the reduction rules. In rule [THREADLAUNCH] r is a fresh identifier, the reference to the agent's thread that runs P. As it will become clear in rule [FORK], threads always have exclusive access to their heap reference.

2.5 The Reduction Rules

Each Mob instruction requires at least one machine transition to be processed. The rules are written in one of the following forms, depending if preconditions are required or not:

[Rule Name]

P: (Preconditions)

C: (Current State)

N: (New State)

[Rule Name]

C: (Current State)

N: (New State)

The first rule allows reduction to occur under structural congruence:

[Cong]

P: $\mathcal{A} \equiv \mathcal{A}'$ — **C:** $\mathcal{A}, \mathcal{R}$ / **N:** $\mathcal{A}'', \mathcal{R}''$

C: $\mathcal{A}', \mathcal{R}$

N: $\mathcal{A}'', \mathcal{R}''$

The following rule widens the scope of a reduction in an agent to the whole network:

[AgentRed]

P: — **C:** $\mathcal{A}$ / **N:** $\mathcal{A}''$

C: $\mathcal{A} \mid \mathcal{A}', \mathcal{R}$

N: $\mathcal{A}'' \mid \mathcal{A}', \mathcal{R}''$

Next we provide the rules for the language constructs.

Classes, Services and Agents

A service specifies an interface implemented by some Mob agent. Service definitions are used to supply information to the type-system. Type-checking of a Mob program is performed at compile-time by matching the inferred types for services used or provided by the program with their definitions kept in the name service. 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 name service. If the service is introduced for the first time by the program (an interface for it does not yet exist in the name service) then the type inferred for the service will become the adopted interface for the service as registered in the name service. So, when the agent is created the Sns map is updated with the agent reference for all the services the agent implements.

[Service]

C: $a(h, C, H, (r, B, \textsf{service}\ s\ \{\tilde{m}\}\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P, Q) \mid T, S)$

Simple classes define abstract data-types and we call their instances *objects*. They add an entry to the code area of the agent containing a closure with slots for the attributes of the class, the code for its methods, and for an empty sequence of implemented services, since a regular object does not provide any services. The rule is as follows:

[Class]

C: $a(h, C, H, (r, B, \textsf{class}\ X(\tilde{x})\ \{m_1(\tilde{x}_1)\ \{\ P_1\ \} \ldots m_n(\tilde{x}_n)\ \{\ P_n\ \}\}\ ; P, Q) \mid T, S)$

N: $a(h, C + \{X : (\tilde{x}, \{m_1 : (\tilde{x}_1, P_1), \ldots, m_n : (\tilde{x}_n, P_n)\}, \epsilon)\}, H, (r, B, P, Q) \mid T, S)$

Some classes are special in that they represent full computations. We call their instances *agents*, and we use a different keyword to differentiate them. The definition of an agent is very much like that of a regular class. We create a new entry for a closure with the attributes and code for the methods and indicate which services are provided by the agent.

The **requires** keyword only supplies information to the type-system, indicating which required services must be checked against the definitions in the Sns. This is done at compile time. On the other hand, the **implements** keyword does not only supply information to the type-system, but also states which service entries must be updated whenever a new instance of the class is created. Therefore, the sequence of services implemented by the agent must be kept in order to perform this action in rule [NewAgent].

[Agent]

C: $a(h, C, H, (r, B, \textsf{agent}\ A(\tilde{x})\ \textsf{implements}\ \tilde{s}\ \textsf{requires}\ \tilde{s}'$
$\{m_1(\tilde{x}_1)\ \{\ P_1\ \} \ldots m_n(\tilde{x}_n)\ \{\ P_n\ \}\}\ ; P, Q) \mid T, S)$

N: $a(h, C + \{A : (\tilde{x}, \{m_1 : (\tilde{x}_1, P_1), \ldots, m_n : (\tilde{x}_n, P_n)\}, \tilde{s})\}, H, (r, B, P, Q) \mid T, S)$

Note that agents may not always implement services and thus $\tilde{s}$ may be an empty sequence.

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, and keeps a link for the code of the class.

[NewObject]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad C(X) = (\tilde{x}, M, \epsilon) \quad r'$ fresh

C: $a(h, C, H, (r, B, x = \textbf{new}\ X(\tilde{v})\ ; P, Q) \mid T, S)$

N: $a(h, C, H + \{r' : (\bullet, (\{\textbf{self} : r', \tilde{x} : \tilde{u}\}, X))\}, (r, B + \{x : r'\}, P, Q) \mid T, S)$

Agents in Mob are similar to objects, but they have an execution unit associated to them. This unit will begin by execute the agent's **main** method, a required method that defines the agent's initial behavior, an approach common to many programming languages.

The new agent runs concurrently with the other agents running at the same host. It is initiated with a heap containing its closure at r' (as in the [NewObject] rule) and the reference for the initial thread at r''. This thread executes the agent's **main** method with the agent's environment B'. The parent agent keeps a binding to the reference $r'@b$ of the created agent in x.

[NewAgent]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad \mathsf{copySeq}_{ab}(C, H, \tilde{u}) = (C', H', \tilde{u}') \quad C(A) = (\tilde{x}, M, (s_1\ \ldots\ s_n))$
$\mathsf{code}(C(A), \textbf{main}) = ()P' \quad B' = \{\textbf{self} : r', \tilde{x} : \tilde{u}'\}$
$Sns(s_1) = (t_1, I_1)\ \ldots\ Sns(s_n) = (t_n, I_n) \quad b, r'@b$ fresh

C: $a(h, C, H, (r, B, x = \textbf{new}\ A(\tilde{v})\ ; P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

N: $a(h, C, H, (r, B + \{x : r'@b\}, P, Q) \mid T, S) \mid$
$b(h, C, H' + \{r' : (\bullet, (B', A))\}, \mathsf{launch}(B', P'), \emptyset) \mid \mathcal{A},$
$(Ans + \{r'@b : h\}, Sns + \{s_1 : (t_1, I_1 + \{r'@b\}), \ldots, s_n : (t_n, I_n + \{r'@b\})\})$

Note that both maps of the name service are updated. The reference $r'@b$ holding the agent's closure will be the key in the Ans 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, will be updated with $r'@b$, e. g., $(s_1 : (t_1, I_1 + \{r''@b\}))$ for the implemented service s_1.

Multi-threaded Agents

The **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. This is achieved by granting the thread exclusive access to itself (see [Join] rules).

[Fork]

P: r' fresh

C: $a(h, C, H, (r, B, x = \textbf{fork}\ \{P'\}\ ; P, Q) \mid T, S)$

N: $a(h, C, H + \{r' : (r', \bullet)\}, (r, B + \{x : r'\}, P, Q) \mid (r', B, P', \epsilon) \mid T, S)$

A thread can suspend waiting for the completion of another thread using the instruction **join**. The instruction uses the thread's handle returned by a previous **fork** statement. Since a given thread has

a lock on its own heap reference, any other thread calling **join** will suspend until the first completes and frees the reference, rule [JoinSuspend]. If the thread on which the synchronization is performed terminates its execution before the **join** instruction is called, the reference is freed and the calling thread never suspends,rule [Join].

[Join]

P: $\mathsf{evalSeq}(H, r, B, x) = r' \quad H(r') = (\bullet, \bullet)$

C: $a(h, C, H, (r, B, \textbf{join}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P, Q) \mid T, S)$

[JoinSuspend]

P: $\mathsf{evalSeq}(H, r, B, x) = r' \quad H(r') = (r', \bullet)$

C: $a(h, C, H, (r, B, \textbf{join}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H, T, S + \{r' : (r, B, P, Q)\})$

If the code currently under execution terminates and the stack of previously interrupted code-blocks is empty, the thread has run out of code to execute and terminates, as in rule [End]. In the process it releases the lock associated with its heap reference and places a notify thread to wake up all the threads suspended upon it. The [Notify] rule wakes up every thread suspended on the given reference. $S(r)$ is the set of threads suspended on that reference r.

[End]

P: $H(r) = (r, \bullet)$

C: $a(h, C, H, (r, B, \epsilon, \epsilon) \mid T, S)$

N: $a(h, C, H + \{r : (\bullet, \bullet)\}, \mathsf{notify}(r) \mid T, S)$

[Notify]

P: $H(r) = (\bullet, \bullet)$

C: $a(h, C, H, \mathsf{notify}(r) \mid T, S)$

N: $a(h, C, H, T \mid \mathsf{run}(S(r)), S|_{dom(S)-\{r\}})$

Agent Movement and Discovery

An agent may move to another host, changing the topology of the distributed computation, rule [Go]. The original host will proceed without the moving 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 is discovered by following the binding for the **self** identifier. This can be done, since the **go** instruction can only appear inside the body of an agent definition's method.

[Go]

P: $\mathsf{evalSeq}(H, r, B, v) = h' \quad a \in dom(Ans) \quad B(\textbf{self}) = r'@a \quad h' \in \mathcal{H}$

C: $a(h, C, H, (r, B, \textbf{go}(v)\ ; P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

N: $a(h', C, H, (r, B, P, Q) \mid T, S) \mid \mathcal{A}, (Ans + \{r'@a : h'\}, Sns)$

An agent may invoke a method in another agent only if it has a binding to the reference holding the target agent's closure. Agent discovery in Mob is service-oriented, meaning that agents are discovered for the services they implement. The instruction **bind** consults the network name service and retrieves a heap reference, $r'@b$, associated with an agent that implements a service s and is presently running in host h, rule [Bind]. However, sometimes the host where the agent is running is irrelevant and is not taken is consideration when the reference is picked, rule [BindAny]. In both these rules, the picking criterion is left to the implementation.

[Bind]

P: $\mathsf{evalSeq}(H, r, B, v) = h \quad Sns(s) = (t, I) \quad \exists r'@b \in I :\ Ans(r'@b) = h \wedge b \neq a$

C: $a(h, C, H, (r, B, x = \textbf{bind}(s\ v)\ ; P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

N: $a(h, C, H, (r, B + \{x : r'@b\}, P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

[BindAny]

P: $Sns(s) = (t, I) \quad \exists r'@b \in I :\ b \neq a$

C: $a(h, C, H, (r, B, x = \textbf{bind}(s\ \textbf{null})\ ; P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

N: $a(h, C, H, (r, B + \{x : r'@b\}, P, Q) \mid T, S) \mid \mathcal{A}, (Ans, Sns)$

Current Host

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

[Host]

C: $a(h, C, H, (r, B, x = \textbf{host}()\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B + \{x : h\}, P, Q) \mid T, S)$

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 from an object in some other agent's address space is not possible, unless the agent's interface provides some means to access the object. Methods of the agent itself can of course be invoked from within the agent. Thus, rule [LocalInvoke] applies to both these scenarios. We qualify the reference with its location to ensure that these rules only apply to local references.

The method call pushes the continuation of the thread unto the stack. The resulting code-block is composed by the reference where the result of the method must be placed, and the continuation's environment and sequence of instructions. In this environment, variable x is bound to r'', the place where the returned value is to be placed.

The method's body is set for execution with the object's environment modified with the values assigned to the method's parameters.

Rule [LocalInvokeLocked] 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.

[LocalInvoke]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad B(y) = r'@a \quad \mathsf{tryAccess}(H, r'@a, r) = \textbf{true} \quad H(r'@a) = (B', X)$
$\mathsf{code}(C(X), m) = (\tilde{x})P' \quad r''$ fresh

C: $a(h, C, H, (r, B, x = y.m(\tilde{v}) \; ; P, Q) \mid T, S)$

N: $a(h, C, H + \{r'' : (\bullet, \bullet)\}, (r, B' + \{\tilde{x} : \tilde{u}\}, P', (r'', B + \{x : r''\}, P) :: Q) \mid T, S)$

[LocalInvokeLocked]

P: $B(y) = r'@a \quad \mathsf{tryAccess}(H, r'@a, r) = \textbf{false}$

C: $a(h, C, H, (r, B, x = y.m(\tilde{v}) \; ; P, Q) \mid T, S)$

N: $a(h, C, H, T, S + \{r' : (r, B, x = y.m(\tilde{v}) \; ; P, Q)\})$

The **return** instruction resumes the execution of the code-block pushed to the stack by the method call. However, this code-block may not be the one at the top of the stack. This may happen, for example, if the **return** instruction was executed from within the body of one or several **if** or **while** instructions, since the rules associated to these instructions also use the stack. Nonetheless, it is simple to obtain the desired code-block, since it is the topmost one waiting for a result. Therefore, all the code-blocks not waiting for a result (holding $\bullet$ in the first slot) are discarded, rule [LocalReturnDiscard]. Once the right code-block is found, it is placed under execution and the heap updated with the binding of the given reference to the returned value, rule [LocalReturn].

[LocalReturnDiscard]

C: $a(h, C, H, (r, B, \textbf{return}(v) \; ; P, (\bullet, \_, \_) :: Q) \mid T, S)$

N: $a(h, C, H, (r, B, \textbf{return}(v) \; ; P, Q) \mid T, S)$

[LocalReturn]

P: $\mathsf{eval}(H, r, B, v) = u \quad r' \neq \bullet$

C: $a(h, C, H, (r, B, \mathbf{return}(v)\ ; P, (r', B', P') :: Q) \mid T, S)$

N: $a(h, C, H + \{r' : (\bullet, u)\}, (r, B', P', Q) \mid T, S)$

Remote Method Invocation in an Agent

Agents interact through method invocation. Remote method calls always launch a new thread in the target agent to execute the corresponding code[1]. The thread carries a heap reference, $r''@a$, from the heap of the calling agent, where the result is to be placed on return. The environment for the new thread is composed of the bindings for the object plus the values for the method's parameters. The calling thread is suspended on the reference $r''@a$, waiting for the result.

[RemoteInvoke]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad B(y) = r'@b \quad \mathsf{tryAccess}(H', r'@b, r@a) = \mathbf{true}$
$\mathsf{copySeq}_{ab}(C, H, \tilde{u}) = (C'', H'', \tilde{u}') \quad H'(r'@b) = (B', A) \quad \mathsf{code}(C'(A), m) = (\tilde{x})P' \quad r''@a$ fresh

C: $a(h, C, H, (r, B, x = y.m(\tilde{v})\ ; P, Q) \mid\ T, S) \mid\ b(h', C', H', T', S') \mid \mathcal{A}, \mathcal{R}$

N: $a(h, C, H + \{r'' : (r'', \bullet)\}, T, S + \{r'' : (r, B + \{x : r''\}, P, Q)\}) \mid$
$b(h', C' + C'', H' + H'', (r''@a, B' + \{\tilde{x} : \tilde{u}'\}, P', \epsilon) \mid\ T', S') \mid \mathcal{A}, \mathcal{R}$

The values assigned to the method's parameters are passed by value, except agents that are passed by reference. Passing them by value would constitute a new form of migration. 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.

If the agent on which the method is to be invoked is locked, the rule is similar to the previous. The difference lies in that the thread to be launched in the target agent is placed in its set of suspended threads, instead of its pool of running threads. This requires for the agent to be freed before the thread starts its execution.

[RemoteInvokeLocked]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad B(y) = r'@b \quad \mathsf{tryAccess}(H', r'@b, r@a) = \mathbf{false}$
$\mathsf{copySeq}_{ab}(C, H, \tilde{u}) = (C'', H'', \tilde{u}') \quad H'(r'@b) = (B', A) \quad \mathsf{code}(C'(A), m) = (\tilde{x})P' \quad r''@a$ fresh

C: $a(h, C, H, (r, B, x = y.m(\tilde{v})\ ; P, Q) \mid\ T, S) \mid\ b(h', C', H', T', S') \mid \mathcal{A}, \mathcal{R}$

N: $a(h, C, H + \{r'' : (r'', \bullet)\}, T, S + \{r'' : (r, B + \{x : r''\}, P, Q)\}) \mid$
$b(h', C' + C'', H' + H'', T', S' + \{r' : (r''@a, B' + \{\tilde{x} : \tilde{u}'\}, P', \epsilon)\}) \mid \mathcal{A}, \mathcal{R}$

[1] 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.

The return value from a remote method call 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 notify thread is placed in the pool of threads of the calling agent, that will trigger the [Notify] rule and awake the calling thread.

Note that the stack of the thread is empty. This means that the method call was perform by some other thread. Only in this scenario, the result is sent to the calling agent. Any other scenarios where the stack is not empty are covered by the regular stack-handling rules.

[RemoteReturn]

P: $\mathsf{evalSeq}(H, r, B, v) = u \quad \mathsf{copySeq}_{ab}(C, H, u) = (C'', H'', u')$

C: $a(h, C, H, (r@b, B, \mathbf{return}(v)\ ; P, \epsilon) \mid T, S) \mid\ b(h', C', H', T', S') \mid \mathcal{A}, \mathcal{R}$

N: $a(h, C, H, T, S) \mid b(h', C' + C'', H' + H'' + \{r : (\bullet, u')\}, \mathsf{notify}(r) \mid T', S') \mid \mathcal{A}, \mathcal{R}$

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 **lock** instruction restricts the access to a reference to the current thread. The operation is only allowed if no other thread has exclusive access over the reference, rule [Lock]. If such precondition is not fulfilled the thread suspends on the reference, rule [LockFailed]. 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 that exclusive access operations can only be performed on references owned by the agent.

[Lock]

P: $\mathsf{evalSeq}(H, r, B, x) = r'@a \quad \mathsf{tryLock}(H, r'@a, r) = (H', \mathbf{true})$

C: $a(h, C, H, (r, B, \mathbf{lock}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H', (r, B, P, Q) \mid T, S)$

[LockFailed]

P: $\mathsf{evalSeq}(H, r, B, x) = r'@a \quad \mathsf{tryLock}(H, r'@a, r) = (H, \mathbf{false})$

C: $a(h, C, H, (r, B, \mathbf{lock}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H, T, S + \{r'@a : (r, B, \mathbf{lock}(v)\ ; P, Q)\})$

Instruction **unlock** returns the public access to a given heap reference and notifies every thread suspended on the reference, so that they may resume their execution (rule [Unlock]). If a thread tries to free an object without having exclusive access to it, the operation is ignored, rule [UnlockIgnore].

[Unlock]

P: $\mathsf{evalSeq}(H, r, B, x) = r'@a \quad \mathsf{tryUnlock}(H, r'@a, r) = (H', \textbf{true})$

C: $a(h, C, H, (r, B, \textbf{unlock}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H', \mathsf{notify}(r'@a) \mid (r, B, P, Q) \mid T, S)$

[UnlockIgnore]

P: $\mathsf{evalSeq}(H, r, B, x) = r'@a \quad \mathsf{tryUnlock}(H, r'@a, r) = (H, \textbf{false})$

C: $a(h, C, H, (r, B, \textbf{unlock}(x)\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P, Q) \mid T, S)$

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 pushes the continuation of the thread (P) to the stack. This enables a clean implementation of the **if** and the **break** instructions.

[IfFalse]

P: $\mathsf{evalSeq}(H, r, B, v) = \textbf{false}$

C: $a(h, C, H, (r, B, \textbf{if}\ (v)\ \{P'\}\ \textbf{else}\ \{P''\}\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P'', (\bullet, B, P) :: Q) \mid T, S)$

[IfTrue]

P: $\mathsf{evalSeq}(H, r, B, v) = \textbf{true}$

C: $a(h, C, H, (r, B, \textbf{if}\ (v)\ \{P'\}\ \textbf{else}\ \{P''\}\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P', (\bullet, B, P) :: Q) \mid T, S)$

Likewise, the **while** instruction also requires two rules that depend on the condition's evaluation. Rule [WhileFalse] ignores the body of the instruction, proceeding with the continuation P. Rule [WhileTrue] pushes the continuation P to the stack, and executes the body of the while statement. The process eventually stops when the v evaluates to **false**. The execution then continues with the continuation P popped from the stack (see rule [Resume]).

[WhileFalse]

P: $\mathsf{evalSeq}(H, r, B, v) = \textbf{false}$

C: $a(h, C, H, (r, B, \textbf{while}\ (v)\ \{P'\}\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P, Q) \mid T, S)$

[WhileTrue]

P: $\mathsf{evalSeq}(H, r, B, v) = \textbf{true}$

C: $a(h, C, H, (r, B, \textbf{while}(v)\ \{P'\}; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B, P'\ ; \textbf{while}(v)\ \{P'\}, (\bullet, B, P) :: Q) \mid\ T, S)$

The **break** instruction terminates the execution of the current code-block.

[Break]

C: $a(h, C, H, (r, B, \textbf{break}\ ; P, (\bullet, B', P') :: Q) \mid T, S)$

N: $a(h, C, H, (r, B, \epsilon, (\bullet, B', P') :: Q) \mid\ T, S)$

When the body of an **if** or a **while** instruction terminates, or is explicitly terminated by a **break**, the continuation placed on the top of the stack must be set for execution, rule [Resume]. The environment of the continuation must be updated with the current environment, to inherit all the new bindings for the variables of its domain. The restriction to the domain of B' will reduce the scope of the variables created in the body of the **if** or **while** to that same body.

[Resume]

C: $a(h, C, H, (r, B, \epsilon, (\bullet, B', P) :: Q) \mid T, S)$

N: $a(h, C, H, (r, (B' + B)|_{dom(B')}, P, Q) \mid T, S)$

Execute External Commands

The **exec** instruction calls an external command handler exec that executes the command synchronously. It receives as input the evaluation of arguments to the **exec** instruction. Asynchronous calls can be performed by encapsulating the **exec** instruction in a newly created thread.

[Exec]

P: $\mathsf{evalSeq}(H, r, B, \tilde{v}) = \tilde{u} \quad \mathsf{exec}(\tilde{u}) = u$

C: $a(h, C, H, (r, B, x = \textbf{exec}(\tilde{v})\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B + \{x : u\}, P, Q) \mid T, S)$

Assignment of Expressions

The value associated to an expression may be assigned to a variable, after the expression has been computed. The assignment involves adding a new entry in the environment (B) of the thread.

[Assignment]

P: $\mathsf{evalSeq}(H, r, B, e) = u$

C: $a(h, C, H, (r, B, x = e\ ; P, Q) \mid T, S)$

N: $a(h, C, H, (r, B + \{x : u\}, P, Q) \mid T, S)$

Assigning a value to an attribute of an object involves modifying the object's closure. Thus, an inspection to the object's access status is required. If the access is granted to the current thread, the binding of the given object's attribute is modified, rule [AttrAssignment]. If not, the current thread suspends on the reference holding the object, rule [AttrAssignmentLocked].

[AttrAssignment]

P: $\mathsf{evalSeq}(H, r, B, e) = u \quad B(x) = r' \quad H(r') = (r'', (B', D)) \quad \mathsf{tryAccess}(H, r', r) = \mathbf{true}$

C: $a(h, C, H, (r, B, x.y = e\ ; P, Q) \mid T, S)$

N: $a(h, C, H + \{r' : (r'', (B' + \{y : u\}, D))\}, (r, B, P, Q) \mid T, S)$

[AttrAssignmentLocked]

P: $\mathsf{evalSeq}(H, r, B, e) = u \quad B(x) = r' \quad \mathsf{tryAccess}(H, r', r) = \mathbf{false}$

C: $a(h, C, H, (r, B, x.y = e\ ; P, Q) \mid T, S)$

N: $a(h, C, H, T, S + \{r' : (r, B, x.y = e\ ; P, Q)\})$

Terminate an Agent

Finally, this last rule terminates the execution of an agent.

[Exit]

C: $a(h, C, H, (r, B, \mathbf{exit}()\ ; P, Q) \mid T, S)$

N: $\mathbf{0}_{\mathcal{A}}$

3 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 the distributed version of the TyCO 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.

Acknowledgements This work is partially supported by FCT's project MIMO (contracts POSI/CHS/39789/2001).

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 *ASA/MA'99*, 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)*, 1995.

[5] Yasuaki Honda and Mario Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

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

[7] Robin Milner. *Communication and Concurrency*. Preentice-Hall, 1989.

[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] Holger Peine and Torsten Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In *Proc. of the First International Workshop on Mobile Agents MA'97*, volume 1219 of *LNCS*, pages 316–323. Springer-Verlag, 1997.

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

[11] António Ravara, Ana Matos, Vasco T. Vasconcelos, and Luís Lopes. Lexically Scoping Sistribution: What You See Is What You Get. In *FGC: Foundations of Global Computing*, volume 85(1) of *ENTCS*. Elsevier Science, July 2003.

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

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

[14] 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)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[15] Vasco Vasconcelos and Luís Lopes. A Multi-threaded Typed Assembly Language: Intermediate Language and Virtual Machine. Unpublished.

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

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