An Overview of Mobile Object-Z

Kenji Taguchi[1] and Jin Song Dong[2]

[1] Department of Information Science, Uppsala University,
Kenji.Taguchi@dis.uu.se
[2] Computer Science Department, National University of Singapore,
dongis@comp.nus.edu.sg

Abstract. Mobile Object-Z (MobiOZ) is an extended notation of Object-Z with mobile and communication primitives required for mobile agent applications. In this paper, we will give an overview of the MobiOZ language features and present its semantic foundation. We also demonstrate expressiveness of the notation through a number of examples.

1 Introduction

Recently, the notion of the *mobile agent* has been proposed in order to capture the new form of computation in communication networks. A mobile agent is a computing entity which can move around different hosts on the network, carrying its state and procedures. To support this new computation paradigm, many programming languages and systems have been developed, i.e., Telescript [16], Obliq [2] and Aglets [9]. However these languages and systems are not suitable to be applied at the specification and design level for mobile agents development because they are too involved with implementation details. Much higher level formalisms such as the π-calculus [11], Mobile Ambients [4] and, Dπ [13] have been proposed. Although these formalisms are elegant and can capture the semantics of mobile agent behaviours, they generally cannot scale up for modelling complex mobile agent systems which often have complex data and state properties.

In this paper, we will give an overview of an extension of Object-Z [7,14] for mobile agent applications (called MobiOZ), and present its semantic foundation based on labelled transition systems. MobiOZ includes mobile primitives inspired by Telescript [16], and communication primitives in process algebras. The design principle of the language has its source in the integrations of state-based and behavioural formalisms in Timed-CSP and Object-Z (TCOZ) [10], and CCS and Z [15].

The notion of locality is a key concept in mobile agent languages and systems. That is, agents can only directly communicate with each other at a common location. Our key idea in designing MobiOZ is to add a *locality* dimension to the Object-Z language with various mobile primitives (i.e. *go*, *send*, *here* etc) for capturing agent mobility. In an analogous fashion, many real-time modelling languages add a *time* dimension to the un-timed languages with explicit timing operators (i.e. *wait*, *timeout*, *deadline* etc) for capturing timing properties directly. With the support of the new MobiOZ primitives, many complex behaviours of mobile agents can be effectively modelled.

MobiOZ is based on the Agent-Place model advocated in Telescript, which has two essential entities, *agents* and *places*. The main difference in the roles of these entities

C. George and H. Miao (Eds.): ICFEM 2002, LNCS 2495, pp. 144–155, 2002.

is that agents can move around the network, while places cannot. Agents are to bring information and exchange it with other agents at places. An agent needs to move to some place to meet and interact with other agents. Places are stationary entities which host mobile agents and can be regarded as physical hosts or logical entities (domains in the communication network).

The agent-place model has an intuitive metaphor of distributed computation and simplifies the specification of mobile applications such as electronic commerce. The specification technique in MobiOZ is to specify a mobile agent as an Object-Z class together with its behaviour as process formulae, which include a special process for an entry point of execution. Process formulae are semantically interpreted at a place, which is a holder of processes with an explicit address.

This paper is organised as follows: the next section illustrates the basic syntax of MobiOZ and Section 3 presents its operational semantics. Some example specifications are given in Section 4. Section 5 discusses the language features of MobiOZ and finally section 6 compares related work and concludes this paper.

2 Syntax

In this section the syntax of MobiOZ is explained. We assume that the readers are familiar with Object-Z [7,14].

A basic type *Address* is preserved, which designates the unique address in the network.

$$[Address]$$

The communication model adopted by MobiOZ is based on the following doctrine [3], which is slightly different from the one adopted in Telescript[1].

- Local communication is synchronous
- Remote communication is asynchronous

All communication is written in a manner similar to that found in process algebras, and polyadic communication in which a channel may have more than one parameter, is allowed. The most important difference between channels in process algebras and MobiOZ is that channels in process algebras are constants, whereas channels in MobiOZ are declared as state variables in a class which may be assigned concrete channel names later. Let com_1, com_2 be state variables declared as channels of the same type in different classes, then an input channel com_1 and an output channel $\overline{com_2}$ can communicate each other, provided they refer to the same channel name.

Remote communication is achieved by associating the destination address together with the channel name in the form of $addr :: \overline{com}$. Again address may be denoted by a state variable. Hence the channel name and the address used in the remote communication can be changed by a π-calculus [11] style of communication.

[1] In Telescript, only local communication was implemented. However the model [16] supports *connection* which is a remote communication between agents residing at different places.

The syntax of the channel is outlined below:

$x, x_1, \ldots$	variable name	e	$::= x \mid V \mid l$
$V, V_1, \ldots$	constant name	$expr$	$::= e \mid f(e, \ldots, e)$
f	function name	$input\_action$	$::= com(x, \ldots, x)$
com	input channel name	$output\_action$	$::= \overline{com}\langle expr, \ldots, expr\rangle \mid$
$\overline{com}$	output channel name		$l :: \overline{com}\langle expr, \ldots, expr\rangle$
$l, m, l_1, \ldots$	address name		

A channel is declared in the state schema of an agent with its types by using the keyword **chan**, which is the pre-defined type for channels. Both input and output channels are declared in the same manner, but when they are used in process formulae, output channels must have a line over the symbol. In order to support the polyadic channel, the notation is strengthened by the use of parameterised types. $com : \mathbf{chan}[T]$ declares a channel com with type T.

A mobile agent in MobiOZ has the following primitives:

$go(l)$	moves to the address
$send(< l, \ldots, l >)$	sends multiple copies to the addresses
$here(x)$	obtains the current address
$kill()$	kills itself

The primitive $go(l)$ will move the subsequent process to the designated address and start its execution at that address. The primitive $send(\langle l_1, \ldots, l_n\rangle)$ takes a sequence of addresses and creates multiple instances with unique new identities. The subsequent processes will start to execute at those addresses. It has a different semantics from the one in Telescript and will be discussed later in section 5. The primitive $here(x)$ returns the current address of the place, i.e., the variable x is instantiated by the current address. Finally, The primitive $kill()$ simply kills itself, i.e., its state and execution thread.

These primitives may be used in process formulae, and their semantic interpretation will be given in section 3.

Process formulae are defined at two different levels (agent and place). We will only show the syntax for agent level in this section:

$Op(x_1, \ldots, x_n)$	Parametric operation schema name
;	sequential operator
$\Box$	choice operator
$P(x_1, \ldots, x_n)$	process constant
0	null process
G	state guard

$$
\begin{aligned}
Guard\_Expr ::= {} & G \bullet Op(x_1, \ldots, x_n) \mid \\
& G \bullet input\_action \rightarrow Op(x_1, \ldots, x_n) \mid \\
& G \bullet output\_action \rightarrow Op(x_1, \ldots, x_n)
\end{aligned}
$$

$$
\begin{array}{lll}
\mathit{Mobile\_Op} ::= & \mathit{go}(l) \mid & \mathit{Proc} ::= 0 \mid \\
& \mathit{send}(\langle l, \ldots, l \rangle) \mid & \quad P(x_1, \ldots, x_n) \mid \\
& \mathit{kill}() \mid & \quad \mathit{Guard\_Expr} \mid \\
& \mathit{here}(x) & \quad \mathit{Mobile\_Op} \mid \\
& & \quad \mathit{Proc} \; ; \; \mathit{Proc} \mid \\
& & \quad \mathit{Proc} \; \Box \; \mathit{Proc}
\end{array}
$$

The syntax of operation schema in Object-Z is modified to have substituent parameters, which excludes input/output annotation '?'/'!' of variables (as various forms of channels indicate the input and output). The semantics of operation schemas are remined as atomic actions.

The defining equation which associates a process constant $P(x_1, \ldots, x_n)$ with a process expression *Proc* is denoted by $P(x_1, \ldots, x_n) \mathrel{\widehat{=}} \mathit{Proc}$.

As the syntax indicates, inter-class concurrency is avoided in MobiOZ.

An agent is a class which represents a computing entity that can move around the network. An agent is specified as an Object-Z class with process definitions. An agent class is given in the following form:

MozClassAgent

OZ_Definition

Process_Definition

where *OZ_Definition* is an Object-Z class definition which includes axiomatic definitions, state schema, initial schema, operation schemas, but excludes Object-Z operators: parallel composition, nondeterministic choice and sequential composition. Those operators are replaced by process formulae previously defined. *Process_Definition* is a set of defining equations.

We will not introduce a basic type for the unique identity of agents, since the reference semantics of Object-Z implicitly supports the unique identity of objects of a class which can also assure the unique identity of agents in MobiOZ.

Each agent with process definitions must have a particular process constant *Beh*, which designates an entry point of its execution.

3 Operational Semantics

In this section, we will present the operational semantics of MobiOZ based on labelled transition systems. In order to interpret mobile primitives in process formulae and subsequent state changes, we need to incorporate an explicit notion of locations and states with identity in the semantics.

Address denotes the set of all addresses, *State*, the set of all states, and *ID* the set of all implicit identities of Object-Z classes.

We will use the following symbols for this purpose:

$\sigma, \sigma', \ldots$ semantic function for state
$a, b, \ldots$ class identity

We first lift up the syntax of process formulae by assigning an address, a state and its identity.

$$[\![\cdot]\!] \ : \ \mathit{Proc} \longrightarrow \mathit{Proc} \times \mathit{Address} \times \mathit{State} \times \mathit{ID}$$

The state and location of each agent may be different from each other so that each semantic function which models the state of an agent is subscripted by its implicit identity and a process formula is associated with an address. We will use the notation $l[\![P]\!]_{\sigma_a}$ to denote the lifted process formula P with its state σ_a with an object identity a at an address l, where l is the address of a place where the process runs.

We will now define process formulae with explicit location and state with identity as below:

$$\mathcal{P}, \mathcal{Q}, \mathcal{P}', \ldots \ ::= \ 0 \ \mid \ l[\![P]\!]_{\sigma_a} \ \mid \ \mathcal{P} \mid\mid \mathcal{P}$$

where $||$ is the parallel composition of explicitly located process formulae.

Mobile agents residing at the same address can be easily distinguished by this notation. For instance, $l[\![P]\!]_{\sigma_a} \mid\mid l[\![Q]\!]_{\sigma_b}$, where a and b stand for agent identifiers, σ_a and σ_b for their agent states and P and Q are process formulae owned by different agents. Each mobile agent class has its unique entry point of execution that is designated by the process constant *Beh* so that the execution of each mobile agent starts at some address $l[\![\mathit{Beh}]\!]_{\sigma_a}$.

The semantics of Object-Z classes is often given using a state transition model in which operations make an old (before) state evolve to a new (after) state. We use the convention for semantics functions in which σ stands for an old state and σ' for a new state, and the state evolution caused by an operation *Op* will be denoted by the following notation:

$$\sigma, \sigma' \models \mathit{Op}(V_1, \ldots, V_n)$$

which reads that an operation $\mathit{Op}(V_1, \ldots, V_n)$ is valid under σ, σ'. A guard expression G is evaluated under a single state so that

$$\sigma \models G$$

In case of parametric operation $\mathit{Op}(x_1, \ldots, x_n)$ which takes n arguments, we assume that all parameters $x_1, \ldots, x_n$ appear in *Op*, and the result of all values taken in $\mathit{Op}(V_1, \ldots, V_n)$ is the replacement of all variables by corresponding values.

We will only deal with monadic communication in the semantics for sake of brevity.

Given the following actions:

$$\begin{aligned} \alpha, \beta ::= \tau \ \mid \ \mathit{go}(l) \ \mid \ \mathit{send}(\langle l_1, \ldots, l_n \rangle) \ \mid \ \mathit{here}(l) \ \mid \ \mathit{kill}() \ \mid \mathit{com}(V) \ \mid \\ \overline{\mathit{com}}\langle V \rangle \ \mid l :: \overline{\mathit{com}}\langle V \rangle \ \mid \ \mathit{Op}(V_1, \ldots, V_n) \end{aligned}$$

we can now define the labelled transition semantics $< \mathcal{P}, \{\xrightarrow{\alpha} \mid \alpha \in A\} >$, where A is the set of all actions defined above.

The operational semantics of MobiOZ is given by derivation rules and congruence relations.

Some primitives take their arguments from state variables defined in a class in the derivation rules. This makes the following derivation rules different from ones in process algebras in which only parameters are passed to process formulae.

We will introduce the following symbol for addresses in order to simplify rules for mobile primitives:

$$addr, addr_1, \ldots ::= x \mid l$$

[Go Primitive]

$$\frac{\sigma_a(addr) = m}{l[\![go(addr)\ ;\ P]\!]_{\sigma_a} \xrightarrow{go(m)} m[\![P]\!]_{\sigma_a}}$$

If $addr$ in $go(addr)$ is a state variable defined in that agent with the id a, then it enables the subsequent process to move to the address it denotes by the semantic function σ_a.

Note that $\sigma_a(addr) = addr$, if $addr$ is a constant.

[Send Primitive]

$$\frac{\sigma_a(addr_1) = k_1, \ldots, \sigma_a(addr_n) = k_n}{l[\![send(\langle addr_1, \ldots, addr_n\rangle)\ ;\ P]\!]_{\sigma_a} \xrightarrow{send(\langle k_1,\ldots,k_n\rangle)} k_1[\![P]\!]_{\sigma_{b1}} \mid\mid \ldots \mid\mid k_n[\![P]\!]_{\sigma_{bn}}}$$

If $addr_1, \ldots, addr_n$ in $send(\langle addr_1, \ldots, addr_n\rangle)$ are state variables defined in that agent with its identity a, the subsequent process will migrate to the addresses denoted by the semantic function σ_a. The subsequent processes are interpreted by the same semantic function with newly created identities $b_1, \ldots, b_n$.

[Here Primitive]

$$\frac{\sigma'_a = \sigma_a \oplus \{x \mapsto l\}}{l[\![here(x)\ ;\ P]\!]_{\sigma_a} \xrightarrow{here(l)} l[\![P]\!]_{\sigma'_a}}$$

The *here* primitive obtains the current address. If the x in $here(x)$ is a state variable defined in a class with an identity a, it causes a state change.

$\oplus$ is the function overriding symbol in Z. Hence $\sigma'_a = \sigma_a \oplus \{x \mapsto l\}$ means that σ'_a is the same as σ_a except an assignment of value l to a variable x.

[Kill Primitive]

$$\frac{}{l[\![kill();\ P]\!]_{\sigma_a} \xrightarrow{kill()} l[\![0]\!]_{\varnothing}}$$

The *kill* makes the agent's state empty and all subsequent processes are ignored.

We will denote substitution of variables $x_1, \ldots, x_n$ by $V_1, \ldots, V_n$ in a process P by $P\{x_1/V_1, \ldots, x_n/V_n\}$.

[Object-Z Operation]

$$\frac{\sigma_a \vDash G,\ \ \sigma_a, \sigma'_a \vDash Op(V_1, \ldots, V_n)}{l[\![G \bullet Op(x_1, \ldots, x_n)\ ;\ P]\!]_{\sigma_a} \xrightarrow{Op(V_1,\ldots,V_n)} l[\![P\{x_1/V_1, \ldots, x_n/V_n\}]\!]_{\sigma'_a}}$$

[Input Prefix]

$$\frac{\sigma'_a = \sigma_a \oplus \{x \mapsto V\}, \sigma'_a \vDash G, \sigma'_a, \sigma''_a \vDash Op(V)}{l[\![G \bullet com(x) \to Op(x)\ ;\ P]\!]_{\sigma_a} \xrightarrow{com(V)} l[\![P\{x/V\}]\!]_{\sigma''_a}}$$

Input channel receives a value, which causes a state change as well as instantiating a value to the subsequent process.

[Output Prefix]

$$\frac{\sigma_a \models G, \sigma_a(expr) = V, \sigma_a, \sigma_a' \models Op}{l[\![G \bullet \overline{com}\langle expr\rangle \rightarrow Op\ ;\ \ P]\!]_{\sigma_a} \xrightarrow{\overline{com}\langle V\rangle} l[\![P]\!]_{\sigma_a'}}$$

Output channel may take some expression *expr* which will be evaluated under the semantic function σ_a.

[Asynchronous Output Prefix]

$$\frac{}{l[\![\overline{com}\langle V\rangle]\!]_{\varnothing} \xrightarrow{\overline{com}\langle V\rangle} l[\![0]\!]_{\varnothing}}$$

Asynchronous communication of MobiOZ is based on asynchronous π-calculus in which only a bare output channel without subsequent process is allowed. This bare output channel is only produced by the next rule which describes behaviour of asynchronous remote communication.

[Asynchronous Remote Communication Prefix]

$$\frac{\sigma_a \models G, \sigma_a(expr) = V, \sigma_a, \sigma_a' \models Op}{l[\![G \bullet k :: \overline{com}\langle expr\rangle \rightarrow Op\ ;\ \ P]\!]_{\sigma_a} \xrightarrow{k::\overline{com}\langle V\rangle} k[\![\overline{com}\langle V\rangle]\!]_{\varnothing} \,||\, l[\![P]\!]_{\sigma_a'}}$$

Once an agent sends a bare output channel to a designated address, the subsequent process remains at the same address. The bare output channel only carries a value without a state.

[Synchronisation]

$$\frac{l[\![P]\!]_{\sigma_a} \xrightarrow{com(V)} l[\![P']\!]_{\sigma_a'} \quad l[\![Q]\!]_{\sigma_b} \xrightarrow{\overline{com}\langle V\rangle} l[\![Q']\!]_{\sigma_b}}{l[\![P]\!]_{\sigma_a} \,||\, l[\![Q]\!]_{\sigma_b} \xrightarrow{\tau} l[\![P']\!]_{\sigma_a'} \,||\, l[\![Q']\!]_{\sigma_b}}$$

As the rule describes, two agents which communicate with each other must reside at the same place.

[Process Definition]

$$\frac{l[\![Proc\{x_1/V_1, \ldots, x_n/V_n\}]\!] \xrightarrow{\alpha} \mathcal{P}}{l[\![P(V_1, \ldots, V_n)]\!] \xrightarrow{\alpha} \mathcal{P}}\ P(x_1, \ldots, x_n) \mathrel{\widehat{=}} Proc$$

[Interleaving]

$$\frac{\mathcal{P}_1 \xrightarrow{\alpha} \mathcal{P}_1'}{\mathcal{P}_1 \,||\, \mathcal{P}_2 \xrightarrow{\alpha} \mathcal{P}_1' \,||\, \mathcal{P}_2}$$

[Congruence]

$$\frac{\mathcal{P} \equiv \mathcal{P}' \quad \mathcal{P}' \xrightarrow{\alpha} \mathcal{Q} \quad \mathcal{Q} \equiv \mathcal{Q}'}{\mathcal{P} \xrightarrow{\alpha} \mathcal{Q}'}$$

There are two kinds of congruence relations, one at the process level and another at the place level. For any $l \in Address$, any $\sigma \in States$ and any $a \in ID$,

$$l[\![0]\!]_{\varnothing} \equiv 0 \qquad \mathcal{P} \mid\mid 0 \equiv \mathcal{P}$$
$$l[\![P \Box Q]\!]_{\sigma_a} \equiv l[\![Q \Box P]\!]_{\sigma_a} \qquad \mathcal{P}_1 \mid\mid \mathcal{P}_2 \equiv \mathcal{P}_2 \mid\mid \mathcal{P}_1$$
$$l[\![(P \Box Q) \Box R]\!]_{\sigma_a} \equiv l[\![P \Box (Q \Box R)]\!]_{\sigma_a} \qquad (\mathcal{P}_1 \mid\mid \mathcal{P}_2) \mid\mid \mathcal{P}_3 \equiv \mathcal{P}_1 \mid\mid (\mathcal{P}_2 \mid\mid \mathcal{P}_3)$$
$$l[\![(0;\ P)]\!]_{\sigma_a} \equiv l[\![P]\!]_{\sigma_a}$$
$$l[\![(P;\ 0)]\!]_{\sigma_a} \equiv l[\![P]\!]_{\sigma_a}$$

It must be noted that $l[\![0]\!]_{\varnothing}$ and $l[\![0]\!]_{\sigma_a}$ have significantly different meanings, which will be discussed in the section 5.

4 Examples

In this section, we will show a number of examples of how mobile agent applications can be specified in MobiOZ. The given types for those examples include:

$[Data, Name]$

4.1 Forwarding

Travelling is the basic functionality of mobile agents. In this first example, we present a simple scheme which describes how travelling agents is informed the next destination from a stationary agent at a place. This scheme is called *forwarding*.

Firstly a basic agent consists of three attributes *dest* (next location), *home* (original location) and *com* (communication channel). A forward agent is defined by inheriting the *BasicAgent* and will move to the next destination, once it receives an address *next_dest* from a channel *com*.

BasicAgent

$dest : Address$
$home : Address$
$com : \mathbf{chan}[Address]$

FAgent

BasicAgent

SetDest$(next\_dest : Address)$

$\Delta(dest)$

$dest' = next\_dest$

$Beh \mathrel{\widehat=} go(dest)\ ;\ com(next\_dest) \rightarrow$
$\quad SetDest(next\_dest)\ ;\ \ Beh$

4.2 Tracking

In this example, we will specify a simple scheme called *Registration* in [9]. This scheme consists of two agents, *Finder* which plays a role as a searcher and *Traveller* as a travelling agent, and a stationary agent *DBAgent* playing a role as a database, which keeps track of travelling agents. The travelling agent will report the current address to the database whenever it travels to some other place and the searcher will make a query to the database to find out the current location of the traveller.

Itinerary. A simple itinerary may be used by a travelling agent.

SimpleItinerary

$itinerary : \mathrm{seq}\, Address$

Del

$\Delta(itinerary)$

$itinerary' = tail\ itinerary$

SimpleTraveller

$BasicAgent, SimpleItinerary$

New_Dest

$\Delta(dest);\ \ Del$

$dest' = head\ itinerary$

$Beh \mathrel{\widehat=} here(home)\ ;\ \ Beh_1$
$Beh_1 \mathrel{\widehat=} go(dest)\ ;\ \ Do\_Task\ ;$
$([itinerary \neq \langle\,\rangle] \bullet New\_Dest;\ Beh_1$
$\Box\quad [itinerary = \langle\,\rangle] \bullet go(home))$
$Do\_Task \mathrel{\widehat=} 0$

In *SimpTraveller* class, an empty process definition *Do_Task* is introduced, which may be overridden by the specific subclasses of *SimpTraveller*.

Traveller

$SimpleTraveller[\ \textbf{redef}\ Beh_1\ ]$

$report : \textbf{chan}[Name, Address]$
$unreg : \textbf{chan}[Name]$
$db\_addr : Address$
$name : Name$

$Beh_1 \mathrel{\widehat=}$
$db\_addr :: \overline{report}\langle name, dest\rangle\ ;$
$go(dest)\ ;\ \ Do\_Task\ ;$
$([itinerary \neq \langle\,\rangle] \bullet New\_Dest;\ Beh_1$
$\Box$
$[itinerary = \langle\,\rangle] \bullet$
$db\_addr :: \overline{unreg}\langle name\rangle;\ go(home))$

Finder

$BasicAgent$

$traveller\_addr, db\_addr : Address$
$traveller\_name : Name$
$query : \textbf{chan}[Name, Address]$
$answer : \textbf{chan}[Address]$

Get_Address$(addr : Address)$

$\Delta(traveller\_addr)$

$traveller\_addr' = addr$

$Beh \mathrel{\widehat=}$
$db\_addr :: \overline{query}\langle traveller\_name, home\rangle;$
$answer(addr) \rightarrow Get\_Address(addr)$

The *Traveller* will be specified as a general subclass of the simple traveller. *Traveller* will use two remote communications to report the current address, and to notify that the traveller will no longer exist and the registry can be deleted. *db_addr* designates the address of the place where *DBAgent* will reside. *name* is the key of mobile agents on the database.

DBAgent

$db : Name \nrightarrow Address$
$report, query : \textbf{chan}\ [Name, Address]$
$unreg : \textbf{chan}\ [Name];\ \ answer : \textbf{chan}\ [Address]$

$$
\begin{array}{|l}
\hline
\textit{Register}(n : \textit{Name}, \textit{current\_addr} : \textit{Address}) \\
\Delta(db) \\
\hline
db' = db \oplus \{n \mapsto \textit{current\_addr}\} \\
\hline
\end{array}
$$

$$\textit{Unregister}(n : \textit{Name}) \mathrel{\widehat=} [\Delta(db) \mid db' = \{n\} \lhd db]$$

$$
\begin{array}{l}
\textit{Beh} \mathrel{\widehat=} (\textit{report}(n, \textit{current\_addr}) \rightarrow \textit{Register}(n, \textit{current\_addr}) \Box \\
\quad \textit{query}(n, \textit{home}) \rightarrow [n \in \operatorname{dom} db] \bullet \textit{home} :: \overline{\textit{answer}}\langle db(n) \rangle \Box \\
\quad \textit{unreg}(n) \rightarrow \textit{Unregister}(n)) \ ; \ \ \textit{Beh}
\end{array}
$$

Finder will use a channel *query* to make queries on the current address of the traveller, *answer* to receive the address.

Finally *DBAgent* will use a channel *query* to receive queries on the current address of the traveller, and *answer* to notify the address of the enquired mobile agent.

5 Language Features

MobiOZ can be best described as a *single-threaded* multi-agent with *strong mobility*. In this section, we discuss some of the language features of MobiOZ.

As was previously mentioned in the syntax section, MobiOZ does not allow inter-class concurrency as agents are single-threaded. In this sense, MobiOZ is a formal specification language for single-threaded multi-agent systems. However, from a system point of view, it is multi-threaded.

The weak and strong mobility notions are used to describe the implementations of agent systems [5] and greatly affect the syntax of mobile agent languages [1]. Mobile agent systems and languages with weak mobility need to start execution at a remote host at the initial point of execution. Java based implementations of mobile agent systems such as Aglets [9] suffer this problem. Any statements followed by mobile primitives such as $go(addr)$ will be ignored and re-start its execution at a specific entry point. In this sense, MobiOZ has a strong mobility.

MobiOZ supports asynchronous remote communication, but unlike the standard asynchronous communication which is characterised by unbounded FIFO queue of incoming messages, as our semantics shows that the asynchronous remote communication of MobiOZ behaves more non-deterministically.

It is easy to verify the following process formulae yields the same result by tracing transitions:

$$
\begin{array}{l}
\textit{here}(x) \ ; \ \ go(m) \ ; \ \ \overline{com}\langle V \rangle \ ; \ \ go(x) \ ; \ \ P \\
m :: \overline{com}\langle V \rangle \ ; \ \ P
\end{array}
$$

A natural question arises why MobiOZ provides only one of them, if those two formulae will yield the same result. The main difference between the two comes into play at implementation level, not at the specification level. Mobile agent computation require very strict security and also consideration of the network load, which does not

arise at the specification level. Remote communication is characterised as low security, but less network load. On the other hand messenger agents have high security and more network load.

Our aim was to design a wide spectrum formal specification language that can provide all necessary facilities for specifying of mobile agent applications. Hence we leave the choice to the specifiers which construct they would choose depending on their needs and requirements.

A mobile agent has its own thread of control and state so that the null process 0 which appears in a process formula does not necessarily mean that the agent's state is demolished (garbage collected) together with its process. In order to explicitly specify that its state as well as its process is abandoned, the *kill*() primitive is introduced in MobiOZ.

6 Related Work and Conclusion

There are a number of stateless basic formalisms such as mobile calculi which model several forms of mobility ranging from passing channel names in the π-calculus [11], to dynamic change of hierarchical structures in *Mobile Ambients* [4]. Indeed, our mobile and communication primitives, and their semantic interpretations are indebted to those works.

However we regard MobiOZ as a stateful high-level formal specification language in which mobile agents are stateful objects that carry their states and procedures while they are travelling around the network. Stateful mobile agents require more elaborate locking mechanisms for mobility than those stateless basic formalisms in order to avoid a circumstance in which some process migrates to other host while the rest of processes remain. This is why MobiOZ does not allow inter-object concurrency.

We have witnessed that the main trend of stateful high-level languages for mobile agents is based on Linda which includes LLinda [6] and LIME [12]. LLinda enhances the Linda model with distributed and multiple tuple spaces with access controls. LIME is equipped with reactive programming primitives, in addition to location sensitive access controls.

Another notable example is MobileUnity [8], which is an extension of Unity with mobile and reactive primitives, and their associated proof methods. Mobility is achieved by attaching a distinguished variable for location to each program and change of location is mimicked by assigning a new location to that variable.

Even they are powerful enough to simulate main mobile features of other formalism, e.g., LLinda could simulate the private name passing and the scope extrusion mechanism of the π-calculus [6], they are not readily applicable for the development of mobile agent systems due to the lack of corresponding mobile primitives and a different underlying model for mobile agents.

This conceptual mismatch between those formalisms and existing programming languages and systems makes them hard to be used to develop mobile agent applications.

In this paper, we have given an overview of an integrated formal specification language MobiOZ for mobile agent applications and presented its semantics which sepa-

rates agent's states with agent's identifiers and captures state changes while the agent is moving around the network.

There are many research issues that remain to be addressed, including enhancement of language features by more powerful communication mechanism such as broadcasting, and verification procedures based on the semantics presented in this paper. We are planning to develop the verification method based on a version of Hennessy-Milner logic based on labelled transition systems presented in this paper.

Acknowledgements

We would like to thank Gabriel Ciobanu and Hugh Anderson for many helpful comments. This work is partially supported by the Academic Research grant *Integrated Formal Methods* from National University of Singapore.

References

1. L. Bettini and R. De Nicola. Translating Strong Mobility into Weak Mobility. In *Proceedings of 5th International Conference on Mobile Agents (MA) 2001*. IEEE, 2001.
2. L. Cardelli. A Language with Distributed Scope. In *Conference Record of POPL'95*, pages 286–297. ACM Press, 1995.
3. L. Cardelli. Wide Area Computation. *ICALP'99*, pages 10–24. 1999.
4. L. Cardelli and A. Gordon. Mobile Ambients. *Foundations of Software Science and Computational Structures*, pages 140–155. Springer-Verlag, 1998.
5. G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. *Mobile Object Systems - Towards the Programmable Internet*, pages 93–111. 1997.
6. R. De Nicola, G. Ferrari, and R. Pugliese. Locality based Linda: programming with explicit localities. *TAPSOFT-FASE'97*, pages 712-726. Springer-Verlag, 1997.
7. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan, March 2000.
8. P. J. M. G.-C. Roman and J. Y. Plun. Mobile unity: reasoning and specification in mobile computing. *ACM Trans. Software Engineering and Methodology*, 6(3):250–282, 1997.
9. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1999.
10. B. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. ICSE'98, pages 95–104. IEEE, 1998.
11. R. Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, 1999.
12. G. Picco, A. Murphy, and G.-C. Roman. LIME:Linda Meets Mobility. *ICSE'99*, pages 368–377, IEEE, 1999.
13. J. Riely and M. Hennessy. A typed language for distributed mobile processes (extended abstract). *POPL'98*, pages 378–390, 1998.
14. G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.
15. K. Taguchi and K. Araki. The State-based CCS Semantics for Concurrent Z Specification. *ICFEM'97*, pages 283–292. IEEE, 1997.
16. J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*, pages 437–472. MIT Press, 1996.