A Survey of Theories for Mobile Agents[1)]

Giovanna Di Marzo Serugendo

Centre Universitaire d'Informatique (CUI)

University of Geneva

24, rue Général-Dufour, CH-1211 Geneva 4

Giovanna.Dimarzo@cui.unige.ch

Murhimanya Muhugusa

Microcell Labs Inc.

1250, Blvd. René-Lévesque Ouest, Suite 400

Montréal (Québec), H3B 4W8 Canada

Christian F. Tschudin

International Computer Science Institute

1947 Center Street

Berkeley, CA 94704, USA

1) This work is supported by the Swiss National Science Foundation (FNRS) grant 20-47162.96.

Abstract

This paper presents a comparative survey of formalisms related to mobile agents. It describes the π-calculus and its extensions, the Ambient calculus, Petri nets, Actors, and the family of generative communication languages. Each of these formalisms defines a mathematical framework that can be used to reason about mobile code; they vary greatly in their expressiveness, in the mechanisms they provide to specify mobile code based applications and in their practical usefulness for the validation and the verification of such applications.

In this paper we show how these formalisms can be used to represent the mobility and communication aspects of two mobile code environments: Obliq and Messengers. We compare and classify the different formalisms with respect to mobility and discuss some shortcomings and desirable extensions. We also point to other emerging concepts in formalisms for mobile code systems.

Keywords: Mobile agents, process algebra, Petri nets, generative communication.

1 INTRODUCTION

Mobile agents, mobile computations and mobile code in general are becoming more and more popular. We have a feeling that these paradigms will enable a wide range of exciting new distributed applications. But beyond the fascination and basic engineering challenges, they are also the source of serious security concerns ("what happens with malicious agents, or inside malicious hosts?"), validation concerns ("has the application the correct functionality?") and verification concerns ("is the application implemented in a correct way?"). In order to obtain a conclusive answer to these questions, it is necessary to introduce formal methods providing a mathematical framework useful for specifying and verifying these applications.

This paper focuses on agent mobility and on the way agents interact with each other. It considers formalisms that either explicitly address code and process mobility or may be used for that purpose, even though they do not offer explicit means for that. These formalisms are either process algebra or Petri nets, and they all have features of varying levels of expressiveness that may be used to formalize mobility.

After a brief presentation of two mobile code environments, this paper summarizes several formalisms addressing the problem of mobility: the π-calculus and its extensions, the Ambient calculus, Petri nets, Actors, and the family of generative communication languages. For each of them, a possible (not necessarily the best) application to the two mobile agent environments is proposed. The paper ends with a comparison of the presented formalisms and gives some suggestions on the way to use them. This paper does not address the variety of agent types used in the AI community (intelligent agents, believable agents, autonomous agents, distributed artificial intelligence, etc.) and a possible formalization thereof.

2 MOBILE COMPUTATION (AGENT) ENVIRONMENTS

Making the computations mobile means that applications have to explicitly decide where in the net a computation should take place. *Locality* is made explicit in the hope that the network can be used more efficiently or because we realize that building a system that tries to completely hide the network (and thus locality) is impossible or inappropriate [Waldo *et al.* 1994]. Applications should be able to react in their specific way to various (network) *failures* instead of delegating this task to some general purpose distributed, but transparent, execution platform.

The term *mobile agent* captures best the view that computations can decide for themselves to move or to stay. There have been various systems proposed to implement mobile agents. The common element is the mobility of code between the nodes. Each node becomes an agent execution environment (i.e., virtual machine or interpreter for portability reasons) that understands a common set of agent instructions or a common programming language. But there are different ways of organizing code mobility and interprocess communication: Java [Lea 1997] realizes code mobility by enabling Web browsers to download bytecodes, objects communicate via method invocation, synchronization between threads is obtained by the means of locks associated to each object; Aglets [Lange *et al.* 1997] are Java objects that can execute on one host and suddenly halt execution, dispatch to a remote host, and restarts an execution there. When an aglet moves, it takes along its program code as well as state information; Obliq [Cardelli 1995] objects can be copied to remote sites and references can be adjusted accordingly, they communicate via local or remote method calls; Messengers [Tschudin 1993] are computations that can start new threads in remote hosts, they communicate via local shared memory only.

To give the flavor of different formalizations of a mobile agent environment, we chose the Obliq and Messenger environments as test cases. Obliq can be characterized as a language approach for extending the object-oriented framework to a distributed environment. Obliq's communication style is determined by the requirement that objects should remain callable although they reside on different hosts. The messenger approach is based on another communication model and imposes all communications to be local and indirect. This divergence, among others, helps to work out the differences between the formalisms, and how they could be used to model the notions of mobility, communication and synchronization. Before looking at the formalisms, we briefly introduce the Obliq and the Messenger systems. A more thorough description of these and other mobile agent environments, as well as a comparison of them can be found in [Di Marzo *et al.* 1996].

2.1 Obliq

Obliq is an object-oriented interpreted language. The Obliq environment consists of *sites* that are address spaces containing *locations* [Cardelli 1995]. Locations contain *values* and can be embedded. A value can be a location, an array, an object, etc. Two different kinds of entities can be moved from a site to another: procedures and objects references. First, the environment allows to move procedure code together with all the bindings (location, site) they need for retrieving values that are in their original site. More precisely, a procedure code is moved and executed in a site that is not its original site. The values that this code needs or creates are invoked or overridden at the original site. This way of retrieving/updating values at the original site is called *distributed lexical scoping*. Second, the reference to an object is allowed to migrate, but not the object itself. However, it is possible to create a *clone* of an object and to put it in a remote site. In addition to cloning, it is possible to perform *aliasing* of the original object to the cloned object. This results in every method invocation on the original object being redirected to the cloned object. The clone has the same state as the object itself and it accesses the values stored in the original site by distributed lexical scoping. Interaction and data exchange between objects occur through method calls.

2.2 Messengers

The messenger concept strives at reducing mobile agents to their essence, namely the sending of code instructions in order to realize communication [Tschudin 1993]. Messengers are anonymous and independent mobile threads of execution inside a *messenger execution platform*. They can send arbitrary code sequences (messenger packets) to neighboring platforms which are interconnected by unreliable channels. Each received messenger packet becomes a new *sequential process* (thread) executing the packet's code in parallel with other messenger threads. Messengers coordinate their execution *indirectly* by the means of (1) a shared *dictionary* and (2) *queues*. The dictionary is a strictly local shared data structure accessible to all messengers inside the same platform. Messengers can insert new data, change or remove old data into/from the dictionary. Messengers are able to insert themselves into queues – their execution is then stopped as long as they do not arrive at the head of the queue. Queues, which are dynamically created, have a name and an associated state (stop/go). They can, for example, be used to serialize the access to the shared dictionary. A heavily used data type in the implementation of messengers are "keys" [Tschudin 1997]. Keys are bit strings used to name queues or e.g., to identify session related data in the dictionary. These names are usually created by selecting randomly a new key which avoids complex cross-platform coordination of the name space.

3 π-CALCULUS AND ITS EXTENSIONS

The π-calculus and its extensions are process algebra that focus on process mobility. Processes communicate using channels. The channels define the configuration of the system. Processes send a channel name in the monadic π-calculus, tuples of channel names in the polyadic π-calculus, and tuples of processes and channel names in the higher-order π-calculus.

3.1 Monadic π-calculus

"The π-calculus is a way of describing and analyzing systems consisting of agents which interacts among each other, and whose configuration or neighborhood is continually changing [Milner 1993]." The philosophy behind π-calculus is heavily based on channel *names*. The basic entity is a (channel) name (with no structure) with which more complex entities called *processes* are built.

- A monadic π-calculus process is given by the following *syntax*:

$$P ::= \sum_{i \in I} \alpha_i.P_i \quad | \quad P_1|P_2 \quad | \quad P_1 + P_2 \quad | \quad \nu x P \quad | \quad !P$$

$$\alpha ::= x(y) \quad | \quad \overline{x}y$$

Where x, y stand for names, . is the prefixing operator, $+$ is the sum operator, $|$ is the parallel operator, ν is the restriction operator, $\nu x P$ makes the name x local to P: only P can use x, i.e., x is used nowhere except in P, ! is the replication operator (means $P|P|\ldots$). Prefixes α are atomic actions, they are of two forms: (1) input prefix $x(y)$ which means that the name y is received over channel x, (2) output prefix $\overline{x}y$ which means that the name y is sent over channel x. Names refer to channels. Given a π-calculus P: the bound names of P, noted $bn(P)$ are names restricted by ν or names y appearing in input prefixes $x(y)$; the free names of P, noted $fn(P)$ are the names appearing in P but not bound.

Example: process $\overline{x}y|x(u).\overline{u}v$ will behave like $\overline{y}v$ after the channel name y has been sent. Indeed, the first process $\overline{x}y$ sends the channel name y along channel x, while the second one $x(u).\overline{u}v$ is waiting for the channel name u along the channel x, in order to use it for sending name v.

- The *operational semantics* is roughly sketched; it is inspired from the operational semantics of polyadic π-calculus given in [Sangiorgi 1993]. There are three labels for the transitions: the silent step τ, the input action $x\langle y\rangle$ and the output action $\overline{x}\langle y\rangle$. We present here only the transitions related to input, output, and interaction between two processes by channel-passing.

  Output action: $\overline{x}y.P \xrightarrow{\overline{x}\langle y\rangle} P$ means that after having sent message y (a channel name) over channel x, process $\overline{x}y.P$ behaves like P.

  Input action: $x(y).P \xrightarrow{x\langle z\rangle} P\{z/y\}$ means that if message z (a channel name) is sent over channel x, then the process $x(y).P$, waiting for a channel name on x, receives it and instantiates the channel name to z, it then behaves like P, where all occurrences of y are replaced by z. $\langle . \rangle$ stands for real parameters, while $(.)$ stands for formal parameters.

  Interaction between two processes: $\frac{P \xrightarrow{(\nu y)\overline{x}\langle y\rangle} P' \qquad Q \xrightarrow{x\langle y\rangle} Q'}{P|Q \xrightarrow{\tau} \nu y(P'|Q')} \quad y \cap fn(Q) = \emptyset$ means that if an output action causes P to become P', and the corresponding input action causes Q to become Q', then P and Q in parallel become P' and Q' in parallel, *and* the private (bound) name y emitted by P becomes a private (bound) name of $P'|Q'$.

The particularity of the π-calculus is to allow names of channels to be passed as parameters. If a process moves, its neighborhood changes and with it the channels it uses for communication.

Transition $x(y).P \xrightarrow{x\langle z\rangle} P\{z/y\}$ means that message z is sent along channel x. If we consider that z is not a simple value but a channel name, then the resulting process $P\{z/y\}$ is able to use this name as a channel for further communications. The actual value of the channel is instantiated during the execution of the process.

3.2 Polyadic π-calculus

The polyadic π-calculus extends monadic π-calculus with *typing* [Milner 1993]. Contrary to monadic π-calculus which allows only channel names to be output in channels, polyadic π-calculus allows tuples of names as well as sorts, data structures and functions to be output. Polyadic π-calculus introduces the notions of *abstraction* and *concretion*. Abstractions are used for the definition of processes with formal parameters, concretions are used for effectively employing processes with actual parameters.

- A polyadic π-calculus process is given by the following *syntax*:

$$P ::= \sum_{i \in I} \alpha_i.P_i \quad | \quad P_1|P_2 \quad | \quad P_1 + P_2 \quad | \quad \nu x P \quad | \quad [x = y]P \quad | \quad D\langle\tilde{x}\rangle$$

$$\alpha ::= x(\tilde{y}) \quad | \quad \overline{x}\langle\tilde{y}\rangle$$

Where x, y stand for names, $\tilde{x}, \tilde{y}$ stand for tuples (finite or infinite) of names, . is the prefixing operator, $+$ is the sum operator, $|$ is the parallel operator, ν is the restriction operator, [.] is the matching operator and D is a constant. Constants are defined by equations of the form $D \stackrel{def}{=} (\tilde{x})P$, with $\tilde{x}$ the (bound) list of formal parameters of P. Prefixes can be input ones ($x(\tilde{y})$) or output ones ($\overline{x}\langle\tilde{y}\rangle$).

The intuitive meaning of the syntax is the following: $x(\tilde{y}).P$ is an input-prefixed process waiting for some tuple $\tilde{z}$ to be transmitted along channel x. Once $\tilde{z}$ has been transmitted, the process behaves like P with $\tilde{y}$ instantiated with $\tilde{z}$. The notation $\overline{x}\langle\tilde{y}\rangle.P$ expresses an output-prefixed process sending the tuple $\tilde{y}$ along channel x. Once $\tilde{y}$ has been sent, the process behaves like P. $|$ is the parallel operator (interleaving). $+$ is the choice operator, $P + Q$ behaves non-deterministically like P or Q. A sum can be of length 0 (the inactive process), of finite or even infinite length. $\nu x P$ makes the name x local to P. $[x = y]P$ is a matching used to test the equality of x and y. Constants are useful for defining infinite processes and recursion. If $D \stackrel{def}{=} (\tilde{x})P$ has formal parameters $(\tilde{x})$, the process $D\langle\tilde{y}\rangle$ has real parameters $\tilde{y}$ of the same length than the formal parameters. Constants have to be seen as functions with parameters.

- *Abstractions and Concretions*: D and $(\tilde{x})P$ are called abstractions, whereas $\langle\tilde{x}\rangle P$ are called concretions. $a(\tilde{x}).P$ is noted $a.(\tilde{x})P$, and $\overline{a}\langle\tilde{x}\rangle.P$ is noted $\overline{a}.\langle\tilde{x}\rangle P$. Abstractions are used to define parameterized processes, combinators, and sorts. Concretions are the parameterized processes with formal parameters instantiated with actual ones.

- *Additional notation*: Local computation of the process which does not involve the environment is noted $\tau.P$. Replication (infinite) of a process P is noted $!P$ and stands for $P|P|P...$

- The *operational semantics* of the polyadic π-calculus is given in terms of a labeled transition system. It is very similar to that of monadic π-calculus. There are three labels for the transitions: the silent step τ, the input action $x\langle\tilde{y}\rangle$ and the output action $\overline{x}\langle\tilde{y}\rangle$. The transitions related to input, output are the same as those of monadic π-calculus, with single names y replaced by tuples of names $\tilde{y}$. We only mention the transitions for interaction between processes and for the constants.

  Interaction between processes: $\frac{P \xrightarrow{(\nu \tilde{y'})\overline{x}\langle\tilde{y}\rangle} P' \qquad Q \xrightarrow{\overline{x}\langle\tilde{y}\rangle} Q'}{P|Q \xrightarrow{\tau} \nu\tilde{y'}(P'|Q')}$ $\tilde{y'} \subseteq \tilde{y} - x, \tilde{y'} \cap fn(Q) = \emptyset$, i.e., an output action causes P to become P', the corresponding input action causes Q to become Q', then P and Q in parallel become P' and Q' in parallel, *and* emitted bound names become bound names of $P'|Q'$.

  Constants stand for functions with formal parameters $\tilde{x}$ and actual parameters $\tilde{y}$. They are treated with the transition: $\frac{P\{\tilde{y}/\tilde{x}\} \xrightarrow{\mu} P'}{D\langle\tilde{y}\rangle \xrightarrow{\mu} P'} D \stackrel{def}{=} (\tilde{x})P$, i.e., the function with actual parameters $D\langle\tilde{y}\rangle$ behaves like P where the actual parameters have been substituted to the formal ones.

The monadic π-calculus is obtained from the polyadic π-calculus by allowing only tuples of length 1. Monadic π-calculus supports only one sort while polyadic π-calculus supports several sorts.

3.3 Higher-Order π-calculus

The higher-order π-calculus (HOπ) [Sangiorgi 1993] goes a step further in the use of sorts and types. Monadic and polyadic π-calculus allows names and tuple of names to be transmitted respectively, while higher-order π-calculus allows functions of arbitrary any order to be transmitted.

Sangiorgi identifies a *first-order paradigm* and a *higher-order paradigm* for mobility in process algebra. The notion of mobility in process algebra is realized by sending messages that change the communication interface between components of the system. The first-order paradigm allows ports or names to be transmitted as messages. After the transmission of a port, the communication can take place through this port. This is the reference-passing mechanism we have seen in the π-calculus. The higher-order paradigm allows processes (parameterized or not) to be passed as values in a communication. After a process has been transmitted, it can begin its execution. This is a process-passing mechanism. Sangiorgi proves that the expressiveness of higher-order and first-order π-calculus are the same.

- A higher-order π-calculus process is given by the following *syntax*:

$$P ::= \sum_{i \in I} \alpha_i.P_i \quad | \quad P_1|P_2 \quad | \quad \nu x P \quad | \quad [x=y]P \quad | \quad D\langle\tilde{K}\rangle \quad | \quad X\langle\tilde{K}\rangle$$

$$\alpha ::= x(\tilde{U}) \quad | \quad \overline{x}\langle\tilde{K}\rangle$$

  Where X is an agent (process) variable, $\tilde{K}$ stands for any tuple of agent or (channel) name, and $\tilde{U}$ stands for any tuple of variable or (channel) name. The constants D are defined as $D \stackrel{def}{=} (\tilde{U})P$.

Constants are to be seen as functions whose parameters can be processes or other functions.

The difference between first-order and higher-order π-calculus resides in the fact that (formal or actual) parameters can be channels and/or processes in the higher-order π-calculus, while in first-order π-calculus only channels can be passed as parameters.

Example: in $\overline{x}\langle P\rangle.Q|x(X).X$, once the interaction between the two processes has taken place, the resulting process is $Q|P$. Indeed, process $x(X).X$ was waiting for X to be sent along channel x, i.e., it was waiting for a process X defining its subsequent behavior.

Plain π-calculus is first-order (where only channel names can be used as parameters). In the second-order new types arise, namely functions which can have process types as types for parameters. We increase the order when we build functions whose parameters are of previous orders.

- The *operational semantics* is given in terms of a labeled transition system. The transition system is that of first-order π-calculus extended to enable processes to be used as parameters. There are three labels for the transitions: the silent step τ, the input action $x\langle\tilde{K}\rangle$ and the output action $\overline{x}\langle\tilde{K}\rangle$. The transitions related to input, output, interaction between two processes, and constants are the same than those of polyadic π-calculus with tuples of channels $\tilde{y}$ being replaced by tuples of channels or processes $\tilde{K}$.

- *Equivalences*: Bisimulation usually identifies processes with the same external behavior. Higher-order bisimulation identifies higher-order processes if their interactions with the environment are the same and if their internal processes are bisimilar.

The polyadic π-calculus is obtained from HOπ by allowing only tuples of names (no processes in the tuples).

3.4 Application to Mobile Agents

We will just show how HOπ can be used to model the mobile agent environments described above.

3.4.1 Obliq

A *site* becomes a π-calculus process with a dedicated π-calculus channel. The site waits for cloned objects or procedures to arrive on this channel. As soon as a cloned object or a procedure arrives on this channel, the site behaves like itself in parallel with the process corresponding to the cloned object or the procedure. A *location* becomes a π-calculus process maintaining a value with two dedicated π-calculus channels: one for reading the value, and one for updating the value.

A *procedure* becomes just a π-calculus process. An *object* becomes a π-calculus process which is made of several π-calculus processes in parallel (one for each method). A method is one of these sub-processes with a dedicated channel used by other processes to invoke that method.

The *cloning* of an object just causes the site to behave like itself in parallel with the π-calculus process of the cloned object. If a method m of an object a is aliased to a method m' of an object b, then the π-calculus process for method m changes its behavior in the following manner: each input action on the channel of method m is immediately followed by an output action towards the channel for m'.

In the case of distributed lexical scoping a *binding* becomes a π-calculus channel. A cloned object or a procedure is sent together with its binding. When it encounters a free identifier, it sends an output action through the binding (a π-calculus channel). The output value is a request for the cloned object's value or for the remote procedure to be executed on the free identifier. A process is waiting on that channel and actually serves these requests.

3.4.2 Messengers

A *messenger channel* becomes a π-calculus process with as many π-calculus channels as the number of messenger channels. Its behavior consists of waiting for a messenger (a π-calculus process) to arrive. Once the messenger has arrived in the channel, it becomes a new process in the platform.

The *dictionary* becomes a π-calculus process with as many dedicated π-calculus channels as data. The dictionary waits permanently for data on all these π-calculus channels. A write in the dictionary is an entry in the π-calculus channel, a read is an exit of the π-calculus channel. Another possibility is to consider each data as a process with a dedicated channel.

A *messenger queue* becomes a π-calculus process whose work is to manage processes entering and leaving the queue. A dedicated π-calculus channel is associated to each queue. Once a process enters the queue, it enters in the π-calculus channel and disappears from the set of processes of the system – once it gets out of the queue, it gets out of the π-calculus channel and appears in the set of active processes of the system. Keys that name queues are simple π-calculus names.

A *messenger* is a π-calculus process. It interacts with the dictionary and the queues through dedicated π-calculus channels. A messenger M_1 that sends messenger M_2 to platform P has to send the π-calculus process for M_2 along a π-calculus channel of platform P. The π-calculus process corresponding to the messenger platform P will actually create the π-calculus process for M_2.

A *messenger platform* becomes a π-calculus process whose behavior consists of the parallel execution of the following processes: π-calculus messenger processes, the π-calculus process for the dictionary and the π-calculus processes for messenger queues.

4 AMBIENT CALCULUS

"An *ambient* is a bounded place where computation happens. The interesting property is the existence of a boundary around an ambient. If we want to move computations easily we must be able to determine what should move; a boundary determines what is inside and what is outside an ambient [Cardelli and Gordon 1998]."

An ambient can be seen as a local computation environment containing all the necessary data, code and processes. A whole ambient can move together with its whole content (all the processes inside the ambient). The ambient calculus addresses also the problem of security (crossing of firewall).

- An ambient calculus process is given by the following *syntax*:

$$P, Q ::= (\nu n)P \quad | \quad 0 \quad | \quad P|Q \quad | \quad !P \quad | \quad M[P] \quad | \quad M.P \quad | \quad (x).P \quad | \quad \langle M \rangle$$

$$M ::= x \quad | \quad n \quad | \quad in\ n \quad | \quad out\ n \quad | \quad open\ n \quad | \quad \epsilon \quad | \quad M.M'$$

  Where n stands for names of ambients, ν is the restriction operator, 0 is the inactive process, $|$ is the parallel operator, ! is the replication operator, $M[P]$ are *ambients*, $M.P$ are processes executing an action regulated by capability M, (x) is an input action, $\langle M \rangle$ is an asynchronous output action.

  M is a *capability*. It is either a variable x, an ambient name n, *in* for enabling an ambient to enter another ambient, *out* for enabling an ambient to leave a surrounding ambient and to let them become sibling ambients, or *open* for opening up an ambient (for dissolving and revealing the ambient and its content). M can either be empty or a path of capabilities.

  The intuitive meaning of the syntax is the following: $(\nu n)P$ creates a unique name n within a scope P. 0 is the process that does nothing. $P|Q$ stands for the parallel execution of P and Q. $!P$ stands for the unbounded replication of process P. $n[P]$ denotes an ambient of name n, and process P is running inside n. P can be a complex process (maybe the parallel composition of some other processes). The ambient can move, the process P remains inside the ambient and continues running. P stands for all the processes running in the ambient n. Ambients can be nested, e.g., $n_1[n_2[P]]$ means that process P is running in ambient n_2, which in turn is a sub-ambient of ambient n_1. Several copies of an ambient (with the same name) can exist at the same time in a system. Process $M.P$ executes an action regulated by the capability M and then continues as the process P. (x) is an input action that causes process $(x).P$ to receive as input the name or capability x which is in its surrounding ambient. $\langle M \rangle$ is an *asynchronous* output action that causes M to be put in the surrounding ambient where it can be caught by some process as input.

- The *operational semantics* is sketched just for the capabilities and the communication primitives.

Entry Capability: $n[in\ m.P|Q]|m[R] \stackrel{in\ m}{\longrightarrow} m[n[P|Q]|R]$ means that the action *in* m causes the surrounding ambient n to move to a sibling ambient m. As a result the whole ambient n (with all the processes inside) moves to ambient m.

Exit Capability: $m[n[out\ m.P|Q]|R] \stackrel{out\ m}{\longrightarrow} n[P|Q]|m[R]$ means that ambient n leaves ambient m (its surrounding ambient). n and m become sibling ambients.

Open Capability: $open\ n.P|n[Q] \stackrel{open\ n}{\longrightarrow} P|Q$. Ambient n is removed, and the processes inside are revealed. The process that instructs the ambient to *open* is not in the ambient, but at the same level than the ambient.

Local anonymous communication: $(x).P \mid \langle M \rangle \rightarrow P\{x \leftarrow M\}$ means that the output action M is released in an ambient and that it is taken as input (in the same ambient) by a process. This process behaves after the input like P where x has been replaced by M. It is also possible to have communication between ambients (constructed on top of the communication inside an ambient).

Ambient Reduction: $\frac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$ reflects the fact that a process executes inside an ambient. If the process executes in a given manner, then it will execute in the same manner inside the ambient.

Capabilities instruct ambients: an ambient can move into a sibling ambient, or get out from a surrounding ambient. The ambient is instructed to move by one process inside the ambient. An ambient enters another ambient with all its processes inside (not only the process that issued the move). This is the *subjective move*. The exit capability is similar: a process inside the ambient instructs the ambient to exit its surrounding ambient: all the processes remain inside the exiting ambient, thus all the processes move.

On the contrary, *objective move* enables a process inside an ambient to move from one ambient to another. The process that performs the objective *in* move enters a new ambient, but its surrounding ambient remains at its place. The process that performs the objective *out* move exits from the current ambient, the other processes remain in that ambient.

Communication primitives allow messages (names or capabilities) to be transmitted between two entities of the same ambients or between entities of two distinct ambients. Communication happens asynchronously and anonymously.

An ambient can be a *mobile process* which moves from one host (ambient) to another. An ambient can be a π-calculus channels which enables processes to communicate the names of other π-calculus channels (ambients). An ambient can be a firewall: a process that wants to cross a firewall has to know the password k (which is an ambient). The firewall itself is another ambient w. A pilot process enters the ambient password k, and with the *in* w capability enables the process to enter w provided the password k has been shown. The process that wants to cross the firewall enters it without knowing the firewall's ambient (it just knows the password). The pilot actually causes the process to cross the firewall. Locks are ambients with no process inside. A process that acquires the lock just *open* the lock: the lock disappears so no other process can acquire the lock. The former process releases the lock by creating the ambient lock again: $open\ n.n[\,].P \mid n[\,].open\ n.Q$.

4.1 Application to Mobile Agents

We will now show how the ambient calculus could be used to formalize the Obliq and Messengers environments.

4.1.1 Obliq

A *site* becomes an ambient, $s[l[P]|l'[O]|\ldots]$, with as many parallel sub-ambients as the number of locations. *Locations* become ambients with one process or one sub-ambient inside. *Procedures* become processes running inside a location ambient: $l[P]$. *Objects* become sub-ambients of a location ambient. The object has as many parallel processes as the number of methods: $l'[O[m1|m2|\ldots]]$.

The migration of a procedure P to site n consists of moving the location ambient $l[P]$ to n. After the move, the ambient $l[P]$ runs inside n and in parallel with the other processes running inside n. The cloning of an object O at a site n causes $l[O]$ to be added in parallel to all the other processes running in n. The aliasing of an object O to an object O' causes O to use a path of capabilities to actually performs methods of O'. Similarly, for the distributed lexical scoping: if a cloned object O needs an access to a value at the remote site it has to know the whole path $(M.M'...)$ necessary to retrieve the value in its original ambient.

4.1.2 Messengers

The *messenger platform* becomes an ambient with as many sub-ambients as necessary for the number of messengers running inside the platform, the number of data items in the dictionary, and the number of queues. A *messenger* becomes an ambient with one process inside. Keys are ambient names.

Each *data* of the dictionary is an ambient with one process inside that is able to receive as input a value (the value of the data) and is able to output this value if requested. A messenger that wants (1) to create the data, creates the corresponding ambient, (2) to override the data: enters the data ambient, takes as input the old value, outputs the new value, and leaves the data ambient, (3) to read the data: enters the data ambient, takes as input the value and outputs the same value, and leaves the data ambient. The process inside the data ambient performs the interaction with the messengers.

Queues are ambients with several sub-ambients (as many as the number of places in the queue) numbered from 1 to k. A messenger firstly enters the ambient queue, then it opens the first sub-ambient if it exists, i.e., there is currently no other messenger locking that sub-ambient, then it tries to open the next sub-ambient, and if it succeeds, then it creates the first sub-ambient, if not it waits. In that manner the messengers are shifted gradually to the head of the queue. The messenger at the head of the queue then waits for a lock on the entire queue. This lock is acquired or released by messengers controlling the queue.

A messenger M_1 that wants to submit a messenger M_2 to a remote platform creates an ambient for M_2 with a process inside whose first action is to move itself to the remote site ambient.

5 PETRI NETS

Petri nets are a formalism often used to model concurrent and parallel systems. The original place/transition Petri nets have been extended in different ways in order to capture high-level abstractions which are difficult or impossible to express with pure place/transition Petri nets. Early Petri nets extensions focused on adding "more structure" on tokens. Other extensions aimed at support for object-orientation and hierarchical Petri nets, and still more recent extensions are directed towards formalization of system dynamicity and process mobility. We briefly present in this section a number of Petri nets extensions, namely mobile Petri nets, dynamic Petri nets, communicative and cooperative Petri nets and CO-OPN, which may all be used to formalize process mobility.

5.1 Mobile and Dynamic Petri Nets

Mobile Petri nets and *dynamic Petri nets* are introduced in [Asperti and Busi 1996]. In mobile Petri nets, process mobility is expressed using variables and colored tokens in an otherwise static net. Dynamic Petri nets extend mobile Petri nets with mechanisms for modifying the *structure* of a Petri net, i.e., for creating new Petri nets when transitions are fired.

5.1.1 Mobile Petri Nets

As said above, *mobile Petri nets* are a variation of place/transition nets with colored tokens allowing *names of places* to appear as tokens. For instance, $\mathrm{ready}(PRINTER, TYPE), \mathrm{job}(FILE, TYPE) \triangleright PRINTER(FILE)$ is a transition, whose pre- and post-conditions are the left and right part of the $\triangleright$ symbol respectively. Capital names are variable names, they will be instantiated to actual names, only at the firing of the transition. We have two places involved in the pre-condition: ready and job, and one place in the post-condition $PRINTER$, a variable not known in advance. This transition means that if the spooler of name $PRINTER$ and of type $TYPE$ is ready then the job of name $FILE$ and of type $TYPE$ can be sent to the spooler $PRINTER$. The file $FILE$ has *moved* from the place job to the place $PRINTER$. The binding of variables at the firing time will determine which file will be printed on which printer. The enabling and firing of transitions depends on a substitution function for the variables.

Mobile Petri nets handle mobility à la π-calculus, i.e., the mobile Petri net expresses the changing configuration of communication channels between processes. In mobile Petri nets, names of places are allowed to appear as tokens inside places. In π-calculus, names of channels are sent along channels.

5.1.2 Dynamic Petri Nets

Dynamic Petri nets are mobile Petri nets extended with a mechanism for creating new subnets when a transition is fired. This is achieved by allowing the postcondition of a transition to be an entire net and not only a set of places with a post-condition. The enabling and firing of dynamic Petri nets depends on a substitution function for variables. The firing of a transition first removes the unified tokens determined by the substitution function and pre-conditions of the transition, and then adds new places and new transitions with pre- and post-conditions given by the subnets appearing in the post-conditions of the transition.

5.2 Communicative and Cooperative Nets

Communicative nets and *cooperative nets* specify a system as a set of components interacting with each other [Sibertin-Blanc 1994]. Components are Petri nets – an interaction layer is provided to let them communicate or cooperate. Components can be created or destroyed dynamically during the evolution of the system. The overall structure is object-based in the sense that components (communicative or cooperative nets) are instances of predefined object-types. In addition, components are able to interact with each other in a dynamic way, i.e., links between components can change during the system evolution. Moreover, the formalism allows to deduce properties of the whole system from the properties of the components and the way they are composed. To cope with the problem of predefined structure imposed by Petri nets and the desired dynamicity, the author provides an algorithm which produces a *synthetic net* from the component nets. The synthetic net is static but has a behavior that is equivalent to the whole system. Subsequently, we look in an informal way at some details of both communicative nets and cooperative nets.

5.2.1 Communicative Nets

An *object-type* defines the type of a component. It defines the data structure and the net structure of the component. The net structure is defined with a communicative net.

A *communicative net* is a Petri net where tokens are tuples of data types and/or object-types. Places are of two kinds: internal places, accessible only by the communicative net itself, or *accept-places* where the communicative net deposits the data it intends to send to another communicative net. Interaction between two communicative nets occurs by message passing. Transitions are of three types: (1) *data function* *call*: a "classical" transition consuming and producing tokens of tuples of data; (2) *message sending*: a transition of the form $v.mes(v_1, \ldots, v_n)$, where v is a variable for an object name, mes is an accept-place, and $v_1, \ldots, v_n$ are the variables corresponding to the message sent; (3) *object creation*: a transition of the form $v.create(v_1, \ldots, v_n)$, where v is a variable for an object name, and $v_1, \ldots, v_n$ are used for the initial marking of the net to be created.

Dynamic binding of nets is realized by the use of variables v in the transitions $v.mes(v_1, \ldots, v_n)$. The communicative net to which data has to be sent is not known in advance. For example, a token can be an object-type, and hence can determine at run-time the name of the communicative net to communicate with. Dynamic creation is obtained with the transition $v.create(v_1, \ldots, v_n)$.

A *system of objects* is a set of pairs of communicative nets and initial marking. Given a system of objects, an equivalent synthetic net consisting of only one object with only data function call transitions can be produced.

5.2.2 Cooperative nets

Cooperative nets are similar to communicative nets. The main difference is the way cooperative nets interact. Instead of using message passing, as is the case with communicative nets, cooperative nets use a client/server protocol. Here are the differences in the structure of communicative and cooperative nets: (1) accept-places are split into two places: an *accept-place* for receiving parameters of requests and *result-places* to store the result of the processing of the request; (2) transitions are of four types: data function call, object creation, *service request* and *service retrieve*. The first two types of transitions are similar to those defined in communicative nets. The service request and service retrieve transitions are used respectively to invoke the server for a service and to provide the client with the result. As for communicative nets, transitions are enriched with variables for object names, thus the binding of the interacting nets is dynamically decided.

As is the case for communicative nets, a system of objects is a set of cooperative nets with their marking. An algorithm is provided to build a synthetic net equivalent to the whole system.

5.3 CO-OPN/2

CO-OPN/2 (Concurrent Object-Oriented Petri Nets) [Biberstein 1997] is an object-oriented formalism based on algebraic Petri nets for the specification of concurrent systems. A CO-OPN/2 system is a collection of independent but interacting objects (algebraic nets). An object is an encapsulated algebraic net in which the places compose the internal state and the transitions model the concurrent events of the object. A place consists of a multiset of algebraic values. Object identifiers are algebraic values, they can be stored in the places of algebraic nets. The transitions are divided into two groups: parameterized transitions (called methods) and internal transitions. The former correspond to the services provided to the outside, while the latter describe the internal behaviors of an object. Internal transitions are invisible to the exterior world and may be considered as spontaneous events. The interaction between objects (i.e., algebraic Petri nets) is synchronous, although asynchronous communications may be simulated. Thus, when an object requires a service, it asks to be synchronized with the method (parameterized transition) of the object providing the service. Objects can be created dynamically. The formal semantics of CO-OPN/2 is given in terms of concurrent transition systems expressing all the possible evolutions of objects' states. State changes are associated to a multiset of events that are simultaneously executable. The firing of an object's method causes (internal) transitions to be fired (spontaneously). The internal transitions are fired as long as their pre-condition is fulfilled. An object's method can be fired only if no transition is firable.

5.4 Application to Mobile Agents

Mobile Petri nets manage mobility in the same way than π-calculus: it is a mobility by reference passing. For this reason, we will not explain how to use these Petri nets for modeling mobility. We will just show the adequacy with π-calculus. The π-calculus process $c(x).x\langle y\rangle.P$ becomes the following mobile Petri net: $c(x) \triangleright x(y)$. Indeed the π-calculus process waits on channel c for the reference of channel x, in order to output y on channel x. The corresponding mobile Petri net has one transition and two places: c and x. The pre-condition of the transition removes x from place c and the post-condition inserts y into place x. Mobile Petri nets and π-calculus are equivalent in their way of specifying mobility, however they differ in the sense that Petri nets are well suited for modeling causality relations between events, while process algebra are well suited for composing processes.

We will now show how cooperative nets can be applied to mobile agents.

5.4.1 Obliq

A *site* becomes a component with an accept-place receiving object-types for procedures, objects, or values. A transition removes these object-types from the accept-place and creates for each of them a corresponding component. A *procedure* becomes a component: the Petri net realizes the procedure. An *object* is a component: a method has a dedicated accept-place (for activating the method) and a dedicated result-place (for delivering returned values). A *location* becomes the component's identity of the procedure component of the object component. In order to move a procedure to a site n or to clone an object, it is necessary to put in the accept-place of this site the object-type of the procedure or that of the cloned object. There it will be created and initialized by a transition in the site component.

An aliasing of an object a into an object b consists of forwarding any request received by a to b. A calling object performs a request to a, a searches in the right place for its alias, and instead of serving the request, it makes the request to b, and waits for the result. Once it receives the result, a puts the result into a result-place where the calling object can get it. Requests are realized by inserting some value into an accept-place, and results are obtained by removing these values from a result-place. Distributed lexical scoping is realized in a similar way to aliasing.

5.4.2 Messengers

The *messenger platform* becomes a component with as many accept-places as messenger channels. *Messengers* become components: the behavior defined by the Petri net is the behavior of the messenger. The *dictionary* is a triple (accept-place, result-place, transition) in the messenger platform component. A messenger that inserts a value into the dictionary inserts a pair $\langle name, value \rangle$ into the accept-place for the dictionary. The dictionary transition moves the pairs from the accept-place to the result-place. A messenger that removes a value from the dictionary removes the pair $\langle name, value \rangle$ from the result-place. For updating or consulting the pair $\langle name, value \rangle$ the messenger removes the pair from the result-place and inserts the (updated) pair into the accept-place. A *queue* is a component with an accept-place storing an algebraic value for a FIFO of component identities. A messenger enters a queue by inserting its component's identity into the accept-place of the queue. It then waits for the queue to let it go out: the queue informs the messenger by inserting a token into one of its accept-places. The queue can be stopped or resumed by interacting with other accept-places dedicated for that. The submit of messenger M_2 by messenger M_1 to a platform P is modeled by a transition *submit* in the M_1 component. This transition removes the identity of M_2 from a place of M_1 and inserts it into an accept-place of P. A transition in component P removes this identity and creates the component M_2 which is able to begin its execution in parallel with the other components.

6 ACTORS AND ACTORS RELATED FORMALISMS

The actor model is an early model for distributed processes based on an asynchronous message passing. Messages play a central role in the model. They are events which trigger actors' behaviors.

6.1 Actor Model

Actors, in the *actor model* of [Agha 1986], are independent concurrent processes which interact asynchronously with each other by message passing. A system of actors is a collection of actors executing concurrently and in parallel. Each actor has a mail address (an identity). A message sent to an actor is stored in its mail queue. The behavior of an actor is dependent of the messages received: in response to a message, an actor can (1) *send* messages to other actors, these messages will be put in the receivers' mail queues, (2) *create* dynamically new actors with specified behaviors, and finally (3) *become* a new actor which will process the next message. Mail addresses can be communicated in messages leading to dynamic communication configurations. In the universe of actors everything is an actor e.g., a message is an actor.

6.2 Algebra of Actors

It should be noted that a formalism for the actor model based on a call-by-value λ-calculus has been developed in [Agha *et al.* 1997]. Subsequently, we present an algebra of actors which is an application of the actor model to agents and communication between agents.

6.2.1 Static Actor Algebra

In [Dalmonte and Gaspari 1995], it is shown how a speech act based language can be translated into a *Static Actor Algebra* including many basic communication and synchronization primitives. The static actor algebra captures the following notions of actors: asynchronous message passing, agent identity, implicit receive (the semantics assumes that an agent waits after it made a request). Dynamicity is not supported, i.e., there is a fixed set of actors for which it is not possible to create new actors at run-time.

- We just give a flavor of the *syntax*. An actor algebra is a triple (A, K, ρ) where $A \subseteq A_{actors}$, A is a set of actor names; K is a set of behaviors and ρ is a mapping of actor names into their initial behavior.

  Intuitively, the behavior is a function which maps a state of actor and an incoming message into a program. This program contains the actions that the actor has to perform in response to the incoming message. In the (static) actor model, an actor waits for a message. Once it has received a message, it performs some local computation according to the received message, it then sends some messages and finally performs a *become*, i.e., changes its current behavior to another behavior. The behavior function explains the result of the *become* in the actor model. The syntax is very similar to the syntax of dynamic actor algebra given below, except for the dynamic part.

- Application to speech act performatives.
  This formalism addresses more particularly multi-agent systems composed of several agents communicating by means of knowledge-level coordination primitives. Agents communicate by exchange of messages which are speech act performatives of three types: *assertive* (assertion of a fact to be true), *directive* (command, request, suggestion), *declarative* (information about the agent's internal capability). Agents are *uniquely identified* (notion of agent identity), they exchange messages (performatives) in an *asynchronous* manner, and use an *implicit receive* (there are no receive performatives). The three types of performatives (assertive, directive and declarative) are expressed using the *send* and/or

become instructions available in the actor model. A performative between two agents is translated into the *send* of a message whose content corresponds to the performative.

6.2.2 Dynamic Actor Algebra

The above static algebra of actors is extended to a *Dynamic Algebra of Actors* in [Gaspari 1996]. It captures the basic interaction mechanisms of the actor model: asynchronous message passing between identified actors, with an implicit receive, and the creation of actors at run-time. Actors are able to create dynamically (at run time) new actors. The basic actions that actors can perform are: *send, become, create* for respectively sending a message to another actor, changing its own behavior and creating a new actor.

- A flavor of the *syntax* of the algebra is sketched below. An actor term is a set of actors running in parallel. An actor term is defined by:

  $$A ::= {}^{a}(P)_{s}^{C} \quad | \quad {}^{a}C_{s} \,||\, [a, v] \quad | \quad A||A \quad | \quad A[f]$$
  $$P ::= \text{become}(C, e).P \quad | \quad \text{send}(e_1, e_2).P \quad | \quad \text{create}(b, C, e).P \quad | \quad \sum_{i=1,\ldots,n} e_i : P \quad | \quad \surd$$

  An actor term is one of the following: (1) a busy actor, with P the remaining program to execute, and with state s, name a and behavior C; (2) an idle actor (ready and waiting for messages) with state s, name a and behavior C (a behavior maps a message and a state into a program $C(x, y) \stackrel{def}{=} P$) running in parallel with an actor a waiting for a message v (implicit receive); (3) two actor terms running in parallel; (4) a renaming over actor names. A program is a sequence of instructions become(C, e) for changing the behavior and state of the actor, send(e_1, e_2) for sending message e_2 to actor e_1 and create(b, C, e) for creating an actor with actor name b, behavior C and initial state e. A program can also be a choice between guarded programs, the semantics is such that only one guarded condition e_i is true. A program can also be empty.

- The *operational semantics* is given in terms of a transition system. We will only reproduce the transitions which are interesting from the point of view of creation and communication among actors.

  ${}^{a}(\text{send}(e_1, e_2))_{s}^{C}.P \xrightarrow{\text{send}([[e_1]],[[e_2]])} {}^{a}(P)_{s}^{C} \,||\, ([[e_1]], [[e_2]])$, the sending of a message causes the actor term representing the message to be created.

  ${}^{a}(\text{become}(C', e).P)_{s}^{C} \xrightarrow{\text{become}(a)} {}^{a}(P)_{s} \,||\, {}^{a}C'_{[[e]]}$, an actor which changes its behavior causes a new actor with the same name to be created and the old actor has an empty behavior, i.e., it will die at the end of its computation.

  ${}^{a}(\text{create}(b, C', e).P)_{s}^{C} \xrightarrow{\text{create}(d)} {}^{a}(P[d/b])_{s}^{C} \,||\, {}^{d}C'_{[[e]]}$, a create causes a new idle actor to be created with the specified behavior and initial state.

${}^{a}C_{s} \mid\mid [a, v] \xrightarrow{\text{receive}(a,v)} {}^{a}(P[v/x, s/y])^{C}_{s}$, an idle actor waiting for a message becomes a busy actor with behavior $C(x, y) \stackrel{def}{=} P$ given by the message and the current state.

- *Equivalences*: Intuitively, the order in which messages are sent to actors is not relevant. For this reason, the author considers a semantics based on weak bisimulation, allowing to consider equivalent two actors which send the same set of messages in a different order.

6.3 Application to Mobile Agents

The dynamic actor algebra distinguishes between a program (idle actor) and its execution (busy actor). We use multiple actor spaces (several spaces with actors running inside) in order to model multiple Obliq sites or messenger platforms.

6.3.1 Obliq

A *site* becomes an actor which waits for locations. A *location* becomes an actor name. A *procedure* becomes an actor. An *object* becomes an actor made of several actors in parallel (as many as the number of methods). A procedure is moved to a site in the following way: its location name is sent to the site actor, which creates the corresponding actor.

A cloned object is sent to a site actor. The site actor creates the actor corresponding to the cloned object with create(b, C, e), where b is the identity of the object (its location), C is its code, and e its initial state. The aliasing of object a to object b causes a to become a new actor a' whose behavior consists of waiting for messages $[a', v]$ and to send (forward) these messages to b with send(b, v). The distributed lexical scoping is similar, but in the reverse order. It is b that forwards messages for free identifiers to the original object a.

6.3.2 Messengers

A *messenger platform* becomes an actor whose behavior is to wait for an actor name and a behavior. As soon as such a pair is received, it creates the actor and becomes itself again. Thus, a *messenger* is modeled by an actor. A messenger M_1 that sends a messenger M_2 to a platform sends the actor name and the behavior of M_2 to the actor platform. Each *data* item in the dictionary becomes an actor too. A messenger that wants to change some data into the dictionary sends a message $\langle update, value \rangle$ to the data actor. This actor modifies the data and becomes itself. A messenger actor that wants to read a value has to firstly send a request to the data actor and then waits for that actor to send the value.

Messenger *queues* are actors that wait for queuing requests. A messenger actor that enters a queue sends a message to the queue actor and then becomes an actor that waits for a message from the queue (it is blocked until the queue actor sends it a message).

7 COORDINATION LANGUAGES AND GENERATIVE COMMUNICATION

Coordination languages, also called generative communication languages, focus on the problem of coordination in a multi-process system. We present in this section two such languages: Linda and PoliS. They allow concurrent processes to coordinate their execution by using anonymous and asynchronous communication through shared memory.

7.1 The Linda Paradigm

Linda [Ciancarini 1994] is a coordination language that may be used to parallelize sequential programming languages. Sequential processes cooperate through concurrent access to a shared multiset of tuples. A tuple is a finite sequence of fields with a value and a type. The tuple space is a multiset because it can contain several copies of the same tuple.

Processes access the shared multiset of tuples using the Linda operators: `out(t)`, for inserting a new tuple `t` in the tuple space; `in(t)`, for extracting tuple `t` from the tuple space; `read(t)`, for reading the value of a tuple; and `eval(t)`, for inserting a tuple in the tuple space whose fields can be function evaluations. The `in` and `read` are blocking operators. Fields of a tuple can be variables. Tuples are accessed by pattern matching. After it has been inserted in a tuple space, a tuple is available for all processes. The order of insertion is completely independent from the order of reading or extraction.

Adding Linda operators to a sequential program enables the resulting sequential processes to work in parallel and to coordinate their work through the tuple space. Combining Linda with a sequential programming language means extending the programming language with the above mentioned operators. The programming languages C, Scheme, Modula-2 and Eiffel have been extended with Linda.

7.2 The PoliS Paradigm

PoliS (for polispace) [Ciancarini 1994] is a programming model which extends Linda with *multiple* tuple spaces. The distributed system is a collection of tuple spaces. Tuple spaces are also called *places* and each place has a name. Tuples are of two different kinds: program-tuple and "normal" tuples. Program-tuples are also called *agents* and are threads of execution, while normal tuples are data (a finite number of fields with value and type). A program-tuple is made of two parts: (1) the heading, which is a normal tuple, and (2) a sequence of tuple operations. The four tuple operations are *Test, Consume, Loc_Eval,* or *Out*; they are described below. A tuple can be sent or retrieved from another place by mentioning the name of the place. Communication among places and support for new places is provided by the *meta tuple space*.

The meta tuple space is responsible for routing the messages to the right place and for storing messages sent to a place which is not created yet. These messages will be delivered once the place is created. Places and tuples can be dynamically created by agents.

Agents are completely independent from each other. An agent performs actions on the tuples of any place by executing the sequence of tuple operations specified in the second part of the program-tuple. An agent performs four activities (operations) in the following order: (1) *Test* i.e., the agent tests the values of some tuples, this implies reading tuples and thus waiting for them if they are not in the mentioned place; (2) *Consume* i.e., the agent deletes some tuples; (3) *Loc_Eval* i.e., the agent starts some internal computation (local evaluation) that has no effect on the places; (4) *Out* i.e., the agent inserts new tuples in the places and/or creates new places. Once it has finished its activities the agent dies. An agent is activated in a place if the place contains both a program-tuple and a normal tuple matching the heading of the program-tuple.

ESP [Ciancarini 1994], for Extended Shared Prolog, is a logic programming language based on multiple tuple spaces. Tuples are Prolog terms. Roughly, we can say that the local evaluation part of the activities of an agent is a sequential Prolog program.

7.3 Algebra for Generative Communication

A process algebra similar to CCS that is adapted to *generative communication* is proposed in [Ciancarini *et al.* 1996]. Generative communication is an asynchronous interprocess communication based on a shared data structure. Communication is realized by inserting, reading or extracting elements into/from the shared data structure. Linda is the most representative language for generative communication.

The proposed algebra supports a single tuple space. Tuples in the tuple space are called messages, and each message is considered as an autonomous agent which removes, reads or inserts itself from/into the tuple space. Processes are called agents. Agents interact with the tuple space using the three operations (already seen in Linda) `in,out,` or `read` for respectively extracting a tuple, inserting a tuple or reading a tuple without consuming it.

- We informally present here the *syntax* of this algebra. *Agents* are given by:

$$E ::= \underline{0} \quad | \quad \langle a \rangle \quad | \quad p.E \quad | \quad E|E \quad | \quad E+E \quad | \quad E \backslash L \quad | \quad E[f] \quad | \quad x \quad | \quad \mathrm{rec}\, x.E$$

  Where *Messages* are denoted by: $a, b, \ldots$, and their corresponding agent $\langle a \rangle, \langle b \rangle$. They belong to the set Messages.

  Prefixes are $\{a, \underline{a}, \hat{a} | a \in \text{Messages}\} \cup \{\tau\}$. They are noted $p, q, \ldots$.

  a is a request for extracting the message a from the tuple space.

  $\underline{a}$ is a request for reading the message a without consuming it.

$\hat{a}$ is for inserting message a in the tuple space.

τ represents an internal computation which does not interact with extraction or reading of a message.

Example: $\hat{a}.\tau.b$ is a process that (1) inserts message a into the tuple space, (2) realizes some internal computation represented by τ, and (3) extracts message b from the tuple space. The process ends after the extraction of message b.

Operators on agents are: . for prefix operator; + for choice operator; | for parallel operator; \ for restriction operator with $L \subseteq$ Messages; [.] for relabeling operator with f a relabeling function; rec for recursion operator. x stands for an agent variable.

- The *operational semantics*, given by the means of a transition system, has the following characteristics: the states are the agents and the labels of the transitions are taken in the set Labels $= \{a, \underline{a}, \overline{a}, \underline{\overline{a}} \in$ Messages$\} \cup \{\tau\}$. a is for the request to extract a from the tuple space, $\underline{a}$ is the request for reading a without extraction, $\overline{a}$ for actually extracting a from the tuple space, and $\underline{\overline{a}}$ for actually reading a without consuming it. τ is both for internal computation and for the adjunction of a message to the tuple space.

  In order to illustrate how the tuple space is accessed by the agents, we give here the transitions related to the `in`, `out` and `read` operations: $a.P \xrightarrow{a} P$ is the request to extract a; $\underline{a}.P \xrightarrow{\underline{a}} P$ is the request to read a (without consuming it); $\hat{a}.P \xrightarrow{\tau} \langle a \rangle | P$ puts message a in the tuple space, i.e., adds agent $\langle a \rangle$ to the set of agents; $\langle a \rangle \xrightarrow{\overline{a}} \underline{0}$ means that message a is removed, i.e., agent $\langle a \rangle$ disappears; $\langle a \rangle \xrightarrow{\underline{\overline{a}}} \langle a \rangle$ means that message a is just read without being removed, i.e., agent $\langle a \rangle$ remains.

- An *observational semantics* is introduced to define a weaker equivalence than that resulting from the operational semantics. The motivation behind the observational semantics lies in the fact that the order of insertion of messages in the tuple space is not very meaningful. Under *weak bisimulation,* processes like $\hat{a}.\hat{b}.P$ and $\hat{b}.\hat{a}.P$ are equivalent because inserting first a and then b or vice-versa is equivalent. Under *failure semantics* an equivalence even weaker than weak bisimulation is defined. The motivation is that "the choice between `out` operations does not depend on the environment". The following agents: $\hat{a}.(\hat{b}.P + \hat{c}.P)$ and $\hat{a}.\hat{b}.P + \hat{a}.\hat{c}.P$ are equivalent under failure semantics but not under weak bisimulation.

7.4 Application to Mobile Agents

We use multiple tuple spaces in order to model the multiple Obliq sites or the multiple messenger platforms.

7.4.1 Obliq

A *site* becomes a tuple space with as many tuples as the number of locations. A *location* becomes a tuple in a site. A *procedure* becomes an agent (program-tuple). An *object* becomes an agent made of several agents in parallel (as many as the number of methods). A procedure that arrives at a site is inserted as an agent tuple in the tuple space. The cloning of an object causes to input the object as a message in the local tuple space and to become an agent. The aliasing of object a to object b causes a to change its behavior and to forward messages to b by inserting these messages in the tuple space where b is. The distributed lexical scoping works similarly but in the opposite way.

7.4.2 Messengers

A *messenger platform* becomes a tuple space. The *dictionary* and the *queues* become normal tuples in the messenger platform tuple space. The executing *messengers* present in the platform become a set of program-tuples.

A messenger M_1 that sends messenger M_2 to a platform creates and inserts a new program-tuple into the desired tuple space. A messenger that wants to enter a queue extracts the queue from the shared memory, modifies it by inserting a program-tuple at the end of the queue (the program-tuple representing the messenger code) and then puts the modified queue in the shared space. After that the program-tuple dies. A messenger resuming a queue becomes a program-tuple which extracts the queue from the shared memory, modifies it in order to extract the first program-tuple present in the queue, puts in the shared memory both the modified queue and the program-tuple.

8 DISCUSSION

The formalisms we have presented in this paper have distinguished flavors and offer different views of the mobile agent paradigm. This section compares the investigated formalisms from the point of view of mobility and cites related work.

8.1 Mobility and Formalisms

According to the degree of expressiveness of mobility we can classify the formalisms as follows:

- *Mobility à la π-calculus:* It is a mobility by reference passing. Processes do not move but the communication configuration changes. This way of managing mobility is that of π-calculus and of mobile Petri nets.

- *Mobility à la HOπ:* It is a "true" mobility where processes can really be sent through channels. Processes can move and change their configuration.

- *Mobility à la ambient calculus:* It is a more general kind of mobility as it allows mobility of processes, of channel names, and of a whole environment (a process with its surrounding context).

- *No explicit mobility support:* Mobility is introduced by using the dynamicity of the formalism. A process "moves" by creating a copy of itself at the new location and by ending its execution in the current location. This way of managing mobility is that of the presented extensions of Petri nets, dynamic algebra of actors, and generative communication between multiple tuple spaces.

8.2 Communication and Paradigms

Among the various paradigms we described in this paper, there seems to be no agreement on the "right" communication model to base concurrent systems on. Some use synchronous communication while others use asynchronous communication primitives. Similarly, a number of models use anonymous and indirect communication instead of "personalized" and direct communication.

In the Actor model we find autonomous agents which communicate asynchronously by message passing. Messages are sent directly to an actor's mail address – actors receive messages in a personal mail queue. Thus, coordination and communication is realized through an asynchronous and personal message passing. In the PoliS, Linda approaches we have autonomous agents which communicate asynchronously through a shared memory. Messages are available for all the agents. They can be modified, removed or inserted at any time and by any agent. The major difference between the actor model and the PoliS paradigm is that the communication medium is personal (queue) in the actor model, while it is commonly shared in the PoliS paradigm. In the Obliq environment we have objects communicating by synchronous method calls. Using mobile Obliq procedures and object aliasing, it is possible to redirect communication patterns – which are always direct (object references) – across sites. Messengers communicate asynchronously and anonymously through a shared dictionary.

While at the level of the formalism, the choice between synchronous and asynchronous primitives can be considered a matter of convenience, it makes a difference at the implementation level. Communication *between* remote locations is naturally buffered, therefore an asynchronous model seems adequate. *Inside* a location, however, synchronous communication is easier to implement because the runtime system has not to maintain buffers.

The messenger approach sidesteps somehow these classifications. First, there is no data exchange other than the local one because it is the purpose of mobility to replace inter-location communication. Second, local communication is indirect, and in this respect similar to Linda and PoliS, even though it has different communication primitives. Unlike the tuple space approach there are no blocking policies built into the primitives for accessing the shared memory: messengers have to implement their own synchronization and data management scheme using the thread queues. It is this property that often makes the mapping of messenger concepts into the formalisms a little bit clumsy.

8.3 Levels of Abstractions

Most formalisms we discuss provide an equivalence relation between processes by the means of *bisimulation*. However, in practice, it is still difficult to use them to reason about complex systems, to prove and deduce system properties, or to use them for validation or verification. For instance, it is difficult to give formal answers to questions like: "given a system of several agents, how can we be sure that the joint work of these agents produces the desired computation?" This is a great concern for all researchers working on "multi-agent" systems. Some multi-agent architectures have exhibited "emergent stable behavior". However, there do not exist formalisms which could catch such a "state". This also parallels our lack of knowledge about whether and how such stable behavior can be engineered.

We also point to another area where we would like to see some progress: it would be quite useful to have a new level of abstraction, that is, in-between single mobile agents and the system as a whole. By this we mean specifications that focus on *families* of processes. Thus, related questions would be: How can we specify a collection (family) of mobile agents working towards some common goal? How is the family's changing set of agents represented? How can the growth or decrease in size of a family be described? How can we specify the family's distribution over several platforms? What are properties that can be attributed to a mobile agent family? Based on such a "collection abstraction", new and useful properties could be studied that are currently difficult to formalize. For example, we could reason about the optimal usage of the computing resources of an agent family, the adaptiveness of the family to changing border conditions, or the survivability of the family from a security point of view.

8.4 Still Other Formalisms

The formalisms described in this paper represent a quite limited selection among many other altogether useful approaches to formally describe concurrent and mobile systems. Mobility is now a major focus and new formalisms stressing the notions of locality, failure and security continue to be proposed.

Plain CHOCS [Thomsen 1993] is an intermediary calculus between π-calculus and HOπ in the sense that it enables processes to communicate through fixed channels. Processes are allowed to send other processes, but not names, through the channels. The configuration of processes is static (names are not allowed to be passed in the channels), but processes can change their behavior dynamically. The calculus of [Amadio 1997] is based on an asynchronous π-calculus and introduces the notions of explicit distribution of processes to locations, failure of locations, detection of failure locations, and mobility of processes. The join-calculus [Fournet and Gonthier 1996] and the distributed join-calculus [Fournet *et al.* 1996] are extensions of the π-calculus which introduce the notion of named location and distributed failure. Locations form a tree of embedded locations, and locations can move from one location to another.

Regarding the problem of coordination among mobile agents we mention LLinda [De Nicola *et al.* 1997a; De Nicola *et al.* 1997b] which is a process calculi for *multiple* tuple spaces that also addresses security aspects. Security is also at the center of the spi-calculus [Abadi and Gordon 1997]: this π-calculus extension was designed for the description and analysis of cryptographic protocols.

An important aspect of all these formalisms is whether they can serve as a base for tools and for verification of properties. In [Montanari and Pistore 1997] the π-calculus is modeled in terms of history-dependent automata (with local names in the transitions) that may lead to simpler and automatic validation procedures. Mobile UNITY [Roman *et al.* 1997] is an extension of UNITY [Chandy and Misra 1988] that augments the program state with a location attribute. Recently, this formalization of physical mobility has been used for expressing various forms of code mobility [Picco *et al.* 1997a]; program properties have been expressed and formally verified [Picco *et al.* 1997b] using the UNITY proof logic.

9 CONCLUSIONS

In this paper we investigated formalisms for concurrent systems which we applied to the two mobile code environments Obliq and Messengers. Some of these formalisms stick well to the mobile agent paradigm, while others needed a little bit more of a roundabout way. Process algebra are more useful for highlighting the parallelism and choice aspect of processes. High-level Petri-nets (especially with object-oriented features) are more useful for specifying systems just before they are implemented. A central element for mobile agents are the communication primitives: a mismatch in communication styles between the formalism and the mobile code environment inevitably leads to more complicated descriptions.

For the following years we expect that new variants of these formalisms will be developed that capture mobile agent systems in a more natural way. The increased interest in coordination (that goes beyond simple data communication) is clearly a promising research area whose results can be extended to groups of mobile agents instead of a single agent. Resource control and security for mobile code are the two other nascent research fields that are critical for the success of mobile code systems. Formalisms document the progress in these fields and will continue to help clarifying the underlying issues.

References

Abadi, M. and A. D. Gordon (1997), "A Calculus for Cryptographic Protocols: The Spi Calculus," In *Fourth ACM Conference on Computer and Communications Security*, ACM Press, pp. 36–47.

Agha, G. A. (1986), *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, MA.

Agha, G. A., I. A. Mason, S. F. Smith, and C. L. Talcott (1997), "A Foundation for Actor Computation," *Journal of Functional Programming 7*, 1, 1–72.

Amadio, R. M. (1997), "An Asynchronous Model of Locality, Failure, and Process Mobility," In *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, D. Garlan and D. Le Métayer, Eds., volume 1282 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 374–391, full version as Rapport Interne, LIM Marseille, and Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

Asperti, A. and N. Busi (1996), "Mobile Petri Nets," Technical Report UBLCS-96-10, Laboratory for Computer Science, University of Bologna, Italy.

Biberstein, O. (1997), "CO-OPN/2 An Object-Oriented Formalism for Concurrent Processes," Ph.D. thesis, University of Geneva, Geneva, Switzerland.

Cardelli, L. (1995), "A Language with Distributed Scope," *Computing Systems 8*, 1, 27–59.

Cardelli, L. and A. D. Gordon (1998), "Mobile Ambients," In *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS)*, Lisbon, to appear.

Chandy, K. and J. Misra (1988), *Parallel Program Design*, Addison-Wesley, Reading, MA.

Ciancarini, P. (1994), "Distributed Programming with Logic Tuple Spaces," *New Generation Computing 12*, 3, 251–284.

Ciancarini, P., R. Gorrieri, and G. Zavattaro (1996), "Towards a Calculus for Generative Communication," In *Proceedings of 1st IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, E. Najm and J. Stefani, Eds., Chapman, Paris, France, pp. 289–306.

Dalmonte, A. and M. Gaspari (1995), "Modelling Interaction in Agent System," Technical Report UBLCS-95-7, Laboratory for Computer Science, University of Bologna, Italy.

De Nicola, R., G. Ferrari, and R. Pugliese (1997a), "Coordinating Mobile Agents via Blackboards and Access Rights," In *Proceedings of the 2nd International Conference on Coordination Languages and Models (COORDINATION'97)*, D. Garlan and D. Le Métayer, Eds., volume 1282 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 220–237.

De Nicola, R., G. Ferrari, and R. Pugliese (1997b), "Locality Based Linda: Programming with Explicit

Localities," In *Proceedings of Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 712–726.

Di Marzo, G., M. Muhugusa, and C. F. Tschudin (1996), "Agent Mobility," In *Bots and other Internet Beasties*, J. Williams, Ed., Sams.net, Indianapolis, IN, pp. 375–406.

Fournet, C. and G. Gonthier (1996), "The Reflexive CHAM and the Join-Calculus," In *Conference Record of POPL'96: The* 23^{rd} *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM Press, New York, NY, pp. 372–385.

Fournet, C., G. Gonthier, J. Levy, L. Maranget, and D. Remy (1996), "A Calculus of Mobile Agents," In *Proceedings of 7th International Conference on Concurrency Theory (CONCUR'96)*, U. Montanari and V. Sassone, Eds., volume 1119 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 406–421.

Gaspari, M. (1996), "Towards an Algebra of Actors," Technical Report UBLCS-96-9, Laboratory for Computer Science, University of Bologna, Italy.

Lange, D. B., M. Oshima, G. Karjoth, and K. Kosaka (1997), "Aglets: Programming Mobile Agents in Java," In *1st International Conference on Worldwide Computing and Its Applications (WWCA'97)*, T. Masuda, Y. Masunaga, and M. Tsukamoto, Eds., volume 1274 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 253–266.

Lea, D. (1997), *Concurrent Programming in Java*, The Java Series, Addison-Wesley, Reading, MA.

Milner, R. (1993), "The Polyadic π-calculus: a Tutorial," In *Logic and Algebra of Specification*, Hamer, Brauer, and Schwichtenberg, Eds., Springer-Verlag, Berlin, Germany, pp. 1–49.

Montanari, U. and M. Pistore (1997), "History-Dependent Automata," Technical report, Dipartimento di Informatica, Universita di Pisa, Italy, draft available from ftp://ftp.di.unipi.it/pub/Papers/pistore/ HDautomata.ps.gz.

Picco, G. P., G.-C. Roman, and P. J. McCann (1997a), "Expressing Code Mobility in Mobile UNITY," In *Proceedings of the 6th European Software Engineering Conference held jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'97)*, M. Jazayeri and H. Schauer, Eds., volume 1301 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 500–518.

Picco, G. P., G.-C. Roman, and P. J. McCann (1997b), "Reasoning About Code Mobility in Mobile UNITY," Technical Report WUCS-97-43, Washington University in St. Louis, Saint-Louis, MO.

Roman, G.-C., P. J. McCann, and J. Plunn (1997), "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM Transactions on Software Engineering and Methodology 6*, 3, 250–282.

Sangiorgi, D. (1993), "Expressing Mobility in Process Algebra," Ph.D. thesis, University of Edinburgh, Edinburgh, United Kingdom.

Sibertin-Blanc, C. (1994), "Cooperative Nets," In *Proceedings of Application and Theory of Petri Nets*,

R. Valette, Ed., volume 815 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 471–490.

Thomsen, B. (1993), "Plain CHOCS. A Second Generation Calculus for Higher Order Processes," *Acta Informatica 30*, 1, 1–59.

Tschudin, C. F. (1993), "On the Structuring of Computer Communications," Ph.D. thesis, University of Geneva, Geneva, Switzerland, Thèse No 2632.

Tschudin, C. F. (1997), "The Messenger Environment M0 – A Condensed Description," In *Mobile Object Systems: Towards the Programmable Internet (MOS'96)*, J. Vitek and C. Tschudin, Eds., volume 1222 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 149–156.

Waldo, J., G. Wyant, A. Wollrath, and S. Kendall (1994), "A Note on Distributed Computing," Technical Report SML 94-29, Sun, reprinted in volume 1222 of *LNCS*, Springer-Verlag, Berlin, Germany, pp. 49–64.