Survey of Theories for Mobile Agents*

Giovanna Di Marzo Serugendo Murhimanya Muhugusa
Christian Tschudin†
Jürgen Harms

Centre Universitaire d'Informatique, University of Geneva
24, rue Général Dufour, CH-1211 Genève 4
Phone: +41 22 705 76 43, Fax: +41 22 705 77 80
e-mail: dimarzocui.unige.ch
url: http://www.unige.ch/ dimarzo/home.html

Cahier du CUI no 106

November 15, 1996

*This work is supported by the Swiss National Fund for Scientific Research (FNSRS) grant 20-40631.94

†Institute fur Informatik, University of Zurich

Abstract

This paper investigates existing formalisms related to one or more features of the messenger paradigm: mobile processes, distributed processes, agents, communication through shared memory. A summary of several paradigms and their corresponding algebra is provided. For some of them a way of applying them to the special case of messengers is provided. An analysis and comparison of the different paradigms wrt the mobility is also provided.

Contents

1 Introduction

Messengers are mobile threads of execution coordinating their work through a shared memory and messenger queues, without central control. The salient features of messengers are: mobility, creation of new messengers at run-time, interaction through a shared memory, no central control, composition of messengers into families, these families are made of dynamically (at run-time) changing sets of messengers, collaboration between messengers.

Due to these features messengers can be considered under several different points of view. They can be seen as collaborative agents, as distributed processes similar to actors, as mobile processes, or as coordinated processes. Our aim is to give a semantics to messengers. For this reason and considering the different way messengers can be seen, we have been conducted to consider several different paradigms related to the above mentioned fields: algebra of actors, mobile petri nets, formalisms for agents, algebra for generative communication.

After a brief presentation of the messenger paradigm. This paper summarizes, in an informal way, some paradigms and their corresponding formalisms related to one or more features of messengers. For some of them an application to the messenger paradigm is proposed. The paper ends with a comparison of the presented paradigms.

1.1 The Messenger Paradigm

Messengers are mobile threads of execution useful for structuring distributed algorithms. A messenger execution takes place in a *messenger platform*. Several messenger platforms are connected by unreliable channels through which messengers are sent as simple data packets.

Inside a given platform, messengers are *sequential processes* executing *concurrently and in parallel*.

Messengers *coordinate their executions* by the means of (1) a shared *dictionary*, and (2) *messenger queues*. The dictionary is a shared data structure accessible to all messengers. Messengers can insert new data, change or remove old data in/from the dictionary. Messengers are able to insert themselves in queues, their execution is then stopped as long as they do not arrive at the head of the queue. Once at the head of the queue, two cases may occur, either the queue has been previously stopped, then the messenger at the head of the queue continues to wait until another messenger restarts the queue, or the queue is idle and the messenger at the head of the queue exits the queue and continues its execution where it has stopped when entering the queue.

Messengers can *create at run-time* (1) new data, (2) new messengers in the platform where they are executing, and they can *move* themselves or *send other messengers to other platforms*.

Platforms are connected by unreliable channels through which messengers are sent as simple data packets. Messengers are exchanged between two platforms as simple messages using a common external representation. Arriving messengers are turned into threads of execution by the corresponding platforms. Messengers only can be sent between platforms, if a data has to be sent to another platform, it is encapsulated in a messenger, which will retrieve the data from its code once it has arrived in the platform.

To summarize, messengers are anonymous and autonomous sequential processes, executing in parallel without central control and able to move between platforms and to coordinate their work by the means of a shared data structure and messenger queues.

2 Mobility in Process Algebra

Messengers are mobile threads of execution. For this reason, we can consider them as mobile processes. Formalisms focusing on the mobility part of processes are given by the family of process algebra related to π-calculus.

2.1 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" [14]. The philosophy behind π-calculus is heavily based on *names*. The basic entity is a name (with no structure), with names we built more complex entities called *processes*.

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

$$P ::= \sum_{i \in I} \alpha_i.P_i \quad | \quad P_1|P_2 \quad | \quad \nu x P \quad | \quad !P$$
$$\alpha ::= x(y) \quad | \quad \overline{x}y$$

  Where x, y stand for names, and . is the prefixing operator, + is the sum operator, | is the parallel operator, ν is the restriction operator, ! is the replication operator (means $P|P|\ldots$). Prefixes α 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.

  Example: Process $\overline{x}y|x(u).\overline{u}v$ will behave like $\overline{y}v$ after the exchange of message has taken place. 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 same channel x, in order to use it for sending name v.

- The *operational semantics* is roughly sketched. There are three labels for the transitions: the silent step τ, the input action $x\langle y\rangle$ and the output action $(\nu y)\overline{x}\langle y\rangle$. We present here only the transitions related to input, output, and interaction between two processes by channel-passing.

  Input 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.

  Output action: $x(y).P \xrightarrow{\overline{x}\langle z\rangle} P\{z/y\}$ 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: $\frac{P \xrightarrow{(\nu y')\overline{x}\langle y\rangle} P' \qquad Q \xrightarrow{\overline{x}\langle y\rangle} Q'}{P|Q \xrightarrow{\tau} \nu y'(P'|Q')} y' \cap fn(Q) = \emptyset$ 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, with a restriction on the names of Q'.

How is realized mobility ? The particularity of the π-calculus is to allow names of channels to be passed as parameters. Transition $x(y).P \xrightarrow{\overline{x}\langle z\rangle} P\{z/y\}$ 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 now 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.

2.2 Polyadic π-calculus

In the monadic π-calculus names of channels without structure are allowed to be passed as messages between processes. In the polyadic π-calculus, the notion of type has been added. Instead of having names only, we can have tuples of names, as well as sorts, data structure and functions. The second difference comes from the notion 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*:

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

  Where x, y stand for names, $\tilde{x}, \tilde{y}$ stand for tuples (finite or infinite) of names, and . is the prefixing operator, + is the sum operator, | is the parallel operator, ν is the restriction operator, [] is the matching operator and D a constant. Constants are defined by equations of the form $D \stackrel{def}{=} (\tilde{x})P$, with $\tilde{x}$ the list of free names of P (formal parameters). 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 a tuple $\tilde{z}$ to be transmitted along channel x, once $\tilde{z}$ has been transmitted, the process continues as P, with $\tilde{y}$ instantiated with $\tilde{z}$. $\overline{x}\langle \tilde{y} \rangle.P$ is an output-prefixed process sending the tuple $\tilde{y}$ along channel x, once $\tilde{y}$ has been sent, the process behaves like P. $[x = y]P$ is a matching used to test the equality of x and y. $\nu x P$ makes the name x local to P, only P can use x, x is used nowhere except in P. + 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 infinite length. | is the parallel operator (interleaving). Constants are useful for defining infinite processes and recursion. $D \stackrel{def}{=} (\tilde{x})P$ 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 whose parameters can be channels.

- *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 allowed for the definition of constants as well as for the definition of processes. A process alway interact with its environment by sending or receiving messages along channels (output or input prefixes). Local computation of the process which do 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 systems. There are three labels for the transitions: the silent step τ, the input action $x\langle\tilde{y}\rangle$ and the output action $(\nu\tilde{y})\overline{x}\langle\tilde{y}\rangle$. We present here only the transitions related to input, output, interaction between two processes by message passing, and the constants.

  Input action: $\overline{x}\langle\tilde{y}\rangle.P \xrightarrow{\overline{x}\langle\tilde{y}\rangle} P$ means that after having sent message $\tilde{y}$ (a tuple of channel names) over channel x process $\overline{x}\langle\tilde{y}\rangle.P$ behaves like P.

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

  Interaction between processes: $\frac{P \xrightarrow{(\nu\tilde{y'})\overline{x}\langle\tilde{y}\rangle} P' \qquad Q \xrightarrow{\overline{x}\langle\tilde{y}\rangle} Q'}{P|Q \xrightarrow{\tau} \nu\tilde{y'}(P'|Q')} \tilde{y'} \cap fn(Q) = \emptyset$, 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, with a restriction on the names of 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 above polyadic π-calculus by allowing only tuples of length 1. Monadic π-calculus supports only one sort, while polyadic π-calculus supports several sorts.

2.3 Higher-order π-calculus

In the higher-order π-calculus, Sangiorgi goes a step further into 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.

In his thesis [16], Sangiorgi presents *first-order paradigm* and *higher-order paradigm* for mobility in process algebra. The notion of mobility in process algebra is realized with an exchange of 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 now take place through this port. It is the reference-passing mechanism we have seen in the π-calculus. The higher-order paradigm

allows processes (parameterized or not) to be passes as values in a communication. After a process has been transmitted, it can become to execute. It is a process-passing mechanism. Sangiorgi proves that the expressiveness of higher-order and first-order π-calculus is the same.

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

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

  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, in higher-order π-calculus, (formal or actual) parameters can be channels and/or processes, while in first-order π-calculus only channels can be passed as parameters. An example of process-passing: let the following two processes run in parallel: $\overline{x}\langle P \rangle.Q|x(X).X$, once the interaction between them has taken place, we have the following resulting process $Q|P$. 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.

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

- *Operational semantics* in terms of labeled transition system is given. 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 $(\nu \tilde{y})\overline{x}\langle \tilde{K} \rangle$. As for the first-order, we present here only the transitions related to input, output, message passing and constants.

  Input action: $\overline{x}\langle \tilde{K} \rangle.P \stackrel{\overline{x}\langle \tilde{K} \rangle}{\longrightarrow} P$ means that after having sent message $\tilde{K}$ (channel or process) over channel x process $\overline{x}\langle \tilde{K} \rangle.P$ behaves like P.

  Output action: $x(\tilde{U}).P \stackrel{\overline{x}\langle \tilde{K} \rangle}{\longrightarrow} P\{\tilde{K}/\tilde{U}\}$ means that if message $\tilde{K}$ (channel or process) is sent over channel x, then the process $x(\tilde{U}).P$, waiting for a message on x, receives the message instantiated to $\tilde{K}$ and behaves like P, where all occurrences of $\tilde{U}$ are replaced by $\tilde{K}$. $\langle . \rangle$ stands for real parameters, while $(.)$ stands for formal parameters.

  Interaction between two processes: $\dfrac{P \stackrel{(\nu \tilde{y'})\overline{x}\langle \tilde{K} \rangle}{\longrightarrow} P' \qquad Q \stackrel{\overline{x}\langle \tilde{K} \rangle}{\longrightarrow} Q'}{P|Q \stackrel{\tau}{\longrightarrow} \nu \tilde{y'}(P'|Q')} \tilde{y'} \cap fn(Q) = \emptyset$,
  i.e. 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, with a restriction on the names of Q'.

  Constants stand for functions with formal parameters (channels or processes) $\tilde{U}$ and actual parameters $\tilde{K}$ are treated with the transition:

$\frac{P\{\tilde{K}/\tilde{U}\} \xrightarrow{\mu} P'}{D\langle \tilde{K} \rangle \xrightarrow{\mu} P'} D \stackrel{def}{=} (\tilde{U})P$, i.e. the function with actual parameters $D\langle \tilde{K} \rangle$ behaves like P where the actual parameters have been substituted to the formal ones.

- *Equivalences*: Bisimulation identifies processes with the same external behavior, i.e. the same labels in the labeled transition systems, the same interaction with the environment. Higher-order bisimulation identifies higher-order processes, i.e. processes allowing other processes as parameters, if their interactions with the environment are the same and if their internal processes are bisimilar. Barbed bisimulation is introduced in order to consider equivalent two processes like: $P|Q$ or $Q|P$, ... Barbed bisimulation is weaker than higher-order bisimulation. Barbed bisimulation gives visibility of the channel at which an action occurs.

2.4 Application to Mobile Agents/Messengers

Polyadic π-calculus is very attractive for formalizing messengers because it contains both the notions of mobility and that of channels. However, the fact that only channel names can be passed in channels, complexifies the formalization of messengers. HOπ offers, in addition, the possibility to send processes along channels. For this reason, we prefer to show how HOπ can be used for formalizing messenger.

Translation of messengers into HOπ can be done in the following manner:

- A *messenger channel* becomes a π-calculus process with a dedicated π-calculus channel. Its behavior consists in waiting on 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 (potential) data. The dictionary waits permanently and in parallel on all these π-calculus channels connected to data. 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 the 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 get out of the queue it gets out of the π-calculus channel and appears in the set of processes of the system.

- The *creation* of a new messenger in a messenger channel consists in sending the π-calculus process corresponding to the messenger along the π-calculus channel dedicated to the π-calculus process corresponding to the messenger channel, which will effectively realize the creation.

- A *messenger* is a π-calculus process. Its interactions with the dictionary, the queues and the messenger channels occur through dedicated π-calculus channels.

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

For the special case of polyadic π-calculus, we can also sketch a way of applying it to the messenger paradigm. Starting from the application of HOπ explained above, we change: (1) instead of having π-calculus processes for messenger channels, we have a π-calculus channel for each messenger channel, (2) each π-calculus process corresponding to a messenger has a dedicated π-calculus channel. This channel is used for knowing the name of the platform where the messenger is executing. If the messenger moves, the name of the new platform is sent along this channel, if a messenger is created, it is provided with the name of the first platform where it will execute. This name is important to let the messenger access the dictionary and the queues of the platform where it is executing.

3 Petri Nets

Messengers are concurrently executing processes. Petri nets and their extensions are concerned with the formalization of concurrent and parallel executing processes.

3.1 Mobile Petri nets

In [3], Asperti proposes both (1) *mobile petri nets* and (2) *dynamic petri nets*. Mobile petri nets handle a mobility à la π-calculus, i.e. the mobile petri net expresses the changing configuration of communication channels between processes. Dynamic petri nets are petri nets having the possibility to create a new subnet when a transition is fired.

We will sketch how they realize mobile and dynamic petri nets by avoiding the formal details.

Mobile petri nets

Mobile petri nets are a variation of place/transition nets with colored tokens allowing *names of places to appear as tokens*.

Example: ready$(PRINTER, TYPE)$, 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 $PRINTER$. In the mobile petri net we have that the file $FILE$ has *moved* from the place job to the place $PRINTER$. The binding of variables at the firing time, will precise which file will be printed on which printer. The enabling and firing of transitions, depends on a substitution function for the variables.

This way of managing mobility is very similar to that of π-calculus. 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.

Dynamic petri nets

Dynamic petri nets are mobile petri nets, where the postcondition of a transition can 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 transitions removes the unified tokens provided by the substitution function and pre-conditions of the transitions and adds, to the net places, transitions and pre- post-conditions given by the subnets appearing in the post-conditions of the transitions and the substitution for variables.

3.2 Communicative and Cooperative Nets

Sibertin-Blanc in [17] introduces both *communicative nets* and *cooperative nets*. A system is specified with components interacting with each other. Components are defined with petri nets, an interaction layer is provided to let them communicate or cooperate. Components (the petri nets) are allowed to appear and disappear dynamically, i.e. at run-time; and components are able to interact with each other in a dynamical way, i.e. links between components can change during the execution time. Moreover, the formalism provides the possibility 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 wanted dynamicity, the author introduces an algorithm, which produces a *synthetic net* starting from the component nets. The synthetic net is static but has a behavior equivalent to the whole system.

In the sequel, we will give some informal details on both communicative and cooperative nets. The overall structure is object-based, in the sense that components (communicative or cooperative nets) are instances of predefined object types.

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, that allows tokens to be 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 data it intends to send to another communicative net. Interaction between two communicative nets occur by message passing. Transitions are of three types: (1) *data function call*: a "classical" transition which consumes and produces 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 create.

The 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, e.g. 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 calls as transitions is produced.

Cooperative nets

Cooperative nets are similar to communicative nets. The most difference comes from the way cooperative nets interact. Instead of using message passing, as it is the case with communicative nets, cooperative nets use a client/server protocol. Differences occurring in the structure of communicative and cooperative nets are the following: (1) accept-places are split-ted 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, and *service request* and *service retrieve*. The first two types of transitions are similar to those defined in the communicative nets. The service request and service retrieve transitions are used to invoke the server for a service or to provide the client with the result respectively. As for communicative nets, transitions are dotted with variables for object names, thus the binding of the interacting nets is dynamically decided.

Similarly to communicative nets, a system of objects is now a set of cooperative nets with their marking, and an algorithm is provided to obtain a synthetic net equivalent to the whole system.

3.3 CO-OPN

Biberstein in his Ph.D [4] defines CO-OPN/2 (Concurrent Object-Oriented Petri Nets), which is an object-oriented formalism based on algebraic petri nets for the specification of concurrent system. Basically, we have several algebraic petri nets able to interact with each other by the means of *synchronizations*. The considered algebraic petri nets allow two types of events: (a) internal transitions, spontaneously firable, simply called transitions, (b) external transitions, firable only if called by other transitions, called methods. Both transitions and methods can *call* one or more methods, i.e. require to synchronize with one or more methods. A method which is called by another event is *abstracted*, i.e. the external behavior of the calling event is equal to its behavior together with the behavior of the called method.

The overall structure is embedded in an object-oriented formalism enabling inheritance, subtyping, polymorphism and dynamicity (a special transition *create* is available).

CO-OPN/2 is the extension of CO-OPN which was object-based and not object-oriented.

4 Actors and Actors related formalisms

Messenger are distributed concurrently executing processes and communicating asynchronously. It is interesting to study the actor model, which is an early model for distributed processes based on an asynchronous message passing.

4.1 Actor Model

Actors, in the *actor model* of [1], 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 a 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 communicate in messages leading in dynamic communications. In the universe of actors all is an actor, e.g. a message is an actor.

4.2 Algebra of Actors

We will just mention here that a formalism for the actor model, based on call-by-value λ-calculus, has been developed by Agha et al. in [2].

We will present in the sequel an algebra of actors which is an application of the actor model to agents and communication between agents.

Static Actor Algebra

Dalmonte, in [11] proposes a *Static Actor Algebra* including many basic communication and synchronization primitives and shows how a speech act based language can be translated into this algebra.

The static actor algebra captures the following notions of actors: asynchronous message passing, agent identity, implicit receive (the semantics assumes that an agent starts waiting after it makes a request). Dynamicity is not supported here, i.e. there is a fixed set of actors, it is not possible to create new actors at run-time.

- Taste of syntax: An actor algebra is a triple (A, K, ρ) where $A \subseteq A_{actors}$, a set of actor names; K is a set of behaviors and ρ is a mapping from actor names to their initial behavior.

  Intuitively, the behavior is a function which maps a state of actor and an incoming message to a program. This program is the program 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 here explains the result of the *become* in the actor model. We do not give here a taste of

the syntax as it is very close to that of the below dynamic actor algebra, except for the dynamic part.

- Application to speech act performatives.
  This formalism addresses more particularly multi-agent systems composed of several agents communicating by the means of knowledge-level coordination primitives. Agents communicate by exchange of messages which are speech act performatives of the three following 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 by the means of 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.

  Example: the performative $assert(a, b, M)$ is translated in the algebra as: $send([a, b, (assert, \phi(M))])$. Due to the implicit receive, performatives requiring answers are translated by sending the request message and by changing the behavior of the requesting agent to enable him to wait for the answer.

Dynamic Actor Algebra

The above static algebra of actors is extended to a *Dynamic Algebra of Actors* in [12]. 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. Thus, agents are able to create dynamically (at run time) new agents. The basic actions that agents can perform are: *send, become, create* for respectively sending a message to an agent, changing its own behavior and creating a new agent.

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

$$A ::= {}^{a}(P)_{s}^{C} \quad | \quad {}^{a}C_{s} \,||\, [a, v] \quad | \quad A||A \quad | \quad A[f]$$

$$P ::= \text{become}(C, e).P \quad | \quad \text{send}(e_1, e_2).P \quad | \quad \text{create}(b, C, e).P | \sum_{i=1,\ldots,n} e_i : P \quad | \quad \surd$$

  An actor term is either a busy actor P with state s and name a and behavior C, or an idle actor (empty state) with state s, name a and behavior C (a behavior maps a message and a state to a program $C(x, y) \stackrel{def}{=} P$) running in parallel with an actor a waiting for a message v (implicit receive), or two actor terms in parallel or a renaming over actor names. A program is a sequence of instructions $\text{become}(C, e)$ for changing the behavior and state of the actor, $\text{send}(e_1, e_2)$ for sending message e_2 to actor e_1 and $\text{create}(b, C, e)$ 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.

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

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

  ${}^{a}(\mathrm{become}(C', e).P)^{C}_{s} \xrightarrow{\mathrm{become}(a)} {}^{a}(P)_{s} \,||\, {}^{a}C'_{[[e]]}$, 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.

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

  ${}^{a}C_{s} \,||\, [a, v] \xrightarrow{\mathrm{receive}(a,v)} {}^{a}(P[v/x, s/y])^{C}_{s}$, 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 but in a different order.

5 Coordination Languages and Generative Communication

Messengers interact with each other by the means of a shared dictionary, where each messenger can add, remove or change data. The family of coordination languages, also called generative communication languages, such as Linda and PoliS are concerned with this way of interaction.

5.1 Linda Paradigm

Linda [9] is a coordination language enabling to parallelize sequential programming languages. Sequential processes realize cooperation by accessing concurrently and in parallel to a shared multi-set of tuples. A tuple is a finite sequence of fields with a value and a type. The tuple space is a multi-set because it allows several copies of the same tuple to be in the space at the same moment.

Processes access the shared multi-set of tuples by using Linda language-independent 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` can be blocking operators, i.e. if the requested tuple is not present in the tuple space, the process performing this instruction has to wait until another process provides the tuple in the tuple space. Variables are allowed in the fields of tuples. Tuples are accessed by pattern matching. After it has been inserted a tuple is available for all processes. The order of insertion is completely independent from the order of reading or extraction.

Linda operators added to a sequential program enable sequential processes (1) to work in parallel and (2)to coordinate their work through the tuple space. Linda combined with a sequential programming language consists in extending the programming language with the above mentioned operators. Programming languages that have already been extended with Linda are: C, Scheme, Modula-2 and Eiffel.

5.2 Polis Paradigm

PoliS (for polispace) [9] 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 execution threads, 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. Tuple operations are *Test, Consume, Loc_Eval, Out*, see below for the description of these operations. Tuples (program-tuple as well as normal tuple) can be sent or retrieved from another place by mentioning the name of this place. Communication among places and support for new places is provided by *meta Tuple Space*. The meta Tuple Space is responsible for routing the messages to the right place and for storing messages sent to places which are not yet creates, and for forwarding these messages to these places, once they have been created. Places and tuples can be dynamically created by agents.

Agents are completely independent from each other. An agent is able to perform actions on the tuples of any place by the means of a sequence of tuple operations located in the second part of the program-tuple. An agent performs four activities (operations) and in the following order: (1) *Test*, 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 ended 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 [9], 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.

5.3 Algebra for Generative Communication

In [10], Ciancarini proposes a process algebra like CCS, adapted to *generative communication*. Generative communication is an asynchronous interprocess communication based on a shared data structure. Communication is realized by inserting, reading or extracting elements in 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/in the tuple space. Processes are called agents. Agents interact with the tuple space by the means of 3

operations (already seen in Linda) `in,out,read` for respectively extracting a tuple, inserting a tuple or reading a tuple without consuming it.

Informally we present here the syntax of this algebra:

- *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 the extraction of reading of a message.

  Example: $\hat{a}.\tau.b$ is a process that (1) inserts message a in 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 the processes are: . prefix operator; + for choice operator; | for parallel operator; \ restriction operator; [] relabeling operator; rec recursion operator.

- *Agents* are given by expressions E of the form: $\underline{0}$ for the null agent, $\langle a \rangle$ for the message agent, $p.E$ an agent which begins with prefix p and continues as agent E, $E|E$ two agents in parallel, $E + E$ the choice between two agents , $E \backslash L$, where L is a set of messages, E works on the tuple space except the L part, $E[f]$, where f is a relabeling function, $\text{rec}\, x.E$, the agent is defined recursively.

- *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 $= \{a, \underline{a}, \overline{a}, \underline{\overline{a}} \in \text{Messages}\} \cup \{\tau\}$. 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 only the transitions related to the `in`, `out`, `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; $\langle a \rangle \xrightarrow{\overline{a}} \underline{0}$, message a is effectively removed, i.e. agent $\langle a \rangle$ disappears; $\langle a \rangle \xrightarrow{\underline{\overline{a}}} \langle a \rangle$, message a is just read without being removed, i.e. agent $\langle a \rangle$ remains;

- *Observational semantics* is introduced in order to let become equivalent under this semantics agents that were not equivalent under the operational semantics. The motivation behind the observational semantics lies in the fact that the order of insertion of messages in the tuple space has no importance. 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* more processes than in weak bisimulation become equivalent. The motivation here comes from the fact that "the choice between `out` operations does not depend on the environment". The following agents have to be equivalent: $\hat{a}.(\hat{b}.P + \hat{c}.P)$ and $\hat{a}.\hat{b}.P + \hat{a}.\hat{c}.P$. It is not the case if we consider weak bisimulation, but it is the case if we consider failure semantics.

5.4 Application to Mobile Agents/Messengers

Generative communication is very close to messengers. Indeed, messengers communicate asynchronously through a shared memory (the dictionary). They can insert new values in the dictionary, retrieve or delete previous inserted values. Differences between the two paradigms are that (1) messengers coordinate through the shared memory, but also through the concept of queues; (2) messengers can move from one platform to another; (3) messenger codes do not follow the four activities predefined in PoliS, messengers access the shared memory, move or insert in a queue, in any order and at any time.

Translation of messengers into PoliS-like processes can be done in the following manner:

- A *messenger platform* is a tuple space;
- The *dictionary* and the *queues* become normal tuples and the executing messengers present in the platform become a series of program-tuples.
- *Messenger code* is broken into smaller pieces, such that each piece of code is in conformance with the allowed activities of program-tuples (test, consume, local evaluation, out). A piece ends and another piece begins, each time, the sequence of messenger instructions does not follow the PoliS sequences of instructions of the program-tuples. Each piece of the messenger code is translated into a program-tuple. These program-tuples form a chain, each program-tuple produces, just before dying, a new program-tuple being the translation of the next piece of code of the original messenger, and so on, till the last one which dies without producing a new program-tuple.
- The *instructions* of the messengers are translated in the following manner:
  Submission of new messengers. It is translated into the creation and insertion of a new program-tuple into the desired tuple space.
  Insertion/Resume in/of a queue. Queues are used to serialize messenger executions, and to stop and resume messenger executions. To stick to the PoliS paradigm, a queue becomes a normal tuple of type queue of program-tuples. The program-tuple corresponding to the messenger inserting in the queue, extracts the queue from the shared memory, modifies it in order to inserts 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, and then dies.

6 Multi-agent Systems Theories

Messengers can be complex as well as really simple programs, i.e. we can put in a messenger code what we want. In particular we can include intelligence into a messenger program, leading to an intelligent agent. Besides the intelligence, simple messengers, without any intelligence, can be considered as collaborative agents because they interact with each other in order to realize common or independent tasks. However, messengers are not necessarily intelligent agents, even though reasoning can be implemented in their code. For this reason, it is interesting to investigate also formalisms developed for agents.

6.1 Wooldridge Temporal Logic

In his thesis [19], Wooldridge firstly defines, with mathematical sets, a multi-agent system as being the following tuple:

$$\langle Ag, \Delta_0, \rho, \iota, MR, AR \rangle$$

Where Ag is a set of agents identifiers, and the other symbols stands for mappings from Ag to an initial belief set (a set of formulae), a set of deduction rules, a belief revision function, a message interpretation function, a set of message rules, and a set of action rules respectively. The cycle of an agent is the following: (1) the agent interprets the messages it receives, (2) the agent uses the belief revision function to update its beliefs according to the previous action it took and the messages received, (3) the agent derives the belief set closure, (4) the agent derives the set of possible messages to send, chooses one and sends it, (5) the agent derives the set of actions to perform, chooses one and applies it. The intermediary notion used by Wooldridge is the *epistemic input*, which is either an action or the interpretation of a received message, both depending on the agent's set of beliefs.

Wooldridge proposes two execution models: a synchronous one and an interleaved one. In the synchronous execution model the next state of the system is reached when *all* the agents have performed a *move*. A *move* is a pair of action and message. In the interleaved model, the next state of the system is reached when *one* of the agents has performed a move, the others having not completed their move. The system state is a mapping from the set of all agent identifiers *Agid* to belief sets. After its move, an agent has performed a cognitive action and sent a message (a communicative action).

Upon these models, Wooldridge adds a linear time temporal logic enabling the reasoning about MAS and more specifically about beliefs, actions and messages. Example of operators of Wooldridge's *Agent Logic* (AL) are:

$(\text{Bel}\, i\, \phi)$	Agent i believes ϕ	$(\bigcirc \phi)$	Next ϕ
$(\text{Send}\, i\, j\, \phi)$	Agent i sent j message ϕ	$(\bigcirc \phi)$	Last ϕ
$(\text{Do}\, i\, \alpha)$	Agent i performs action α	$(\phi\, \mathcal{U}\, \psi)$	ϕ Until ψ
		$(\phi\, \mathcal{S}\, \psi)$	ϕ Since ψ

The operators of the AL language enables to define operators of linear time temporal logic as: $\Box$ always, $\blacksquare$ heretofore, $\Diamond$ sometimes, $\blacklozenge$ was, $\mathcal{W}$ unless, $\mathcal{Z}$ since, $\mathcal{B}$ before.

Given a model for AL of the multi-agent system, it is possible to prove temporal formulae in this model.

To summarize, Wooldridge formalizes beliefs, actions and message communications in a unified framework using the notion of epistemic inputs. A deduction system of beliefs is used, which derives new beliefs from previous ones and epistemic inputs. To reason about agents, Wooldridge uses a linear time temporal logic.

6.2 Singh's Logic

In [18], Singh presents a theoretical framework based on *possible worlds*, which is able to formalize features of agents like: intentions, know-how and communications. It is a modal logic of actions and time, which is based on a strict partial order of temporal *moments* denoting different world state. At each moment corresponds (1) a possible state of the world, given by atomic propositions holding at that moment, and (2) the intentions and know-how of agents. An agent can choose an action to perform according to its intentions and know-how. Intuitively, intentions are the properties that are true at a moment in all the possible paths that an agent can choose. A path is a sequence of actions and moments. For the know-how, agents which know how to realize a property, will choose the actions that will lead to the occurrence of that property in the future.

Communication between agents inside a multi-agent system is based on speech act performatives. Performatives are considered as actions and the set of actions is extended to incorporate the performatives.

7 Category Theory

Category theory is used since several years in computer science. The petri net community frequently uses category theory to give an abstract semantics to horizontal (fusion) and vertical (refinement) structuring of petri nets.

Here we firstly give some preliminary definitions and present a recent work on category theory applied to distributed processes.

Definitions of Categories and Functors

A *category* is a graph with O, the set of nodes, also called *objects*, and M, a set of arrows, also called *morphisms*. Domains and codomains of the arrows are given by two functions $dom : M \to O$ and $cod : M \to O$. An associative operation $\circ$ on the arrows is defined over all composable pair of arrows. Arrows $f : X \to Y$ and $g : Y \to Z$ are composable if the domain of g is the codomain of f, their composition $g \circ f : X \to Z$ is an arrow in M. A function $id : O \to M$ gives the identity morphism for each object, $dom(id(A)) = cod(id(A)) = A$ for each object A in O. $id(A)$ is also noted id_A and if $f : A \to B$, then $f \circ if_A = f$ and $id_B \circ f = f$.

A *functor* $f : \mathcal{C} \to \mathcal{D}$ from category $\mathcal{C}$ to category $\mathcal{D}$ is a pair of functions $F_O : \mathcal{C}_O \to \mathcal{D}_O$ and $F_M : \mathcal{C}_M \to \mathcal{D}_M$ such that (1) $F_M(f) : F_O(A) \to F_O(B)$ if $f : A \to B$, (2) $F_M(id_A) = id_{F_O(A)}$, for each A in O, (3) $F_M(g \circ f) = F_M(g) \circ F_M(f)$.

Category theory applied to transition systems and logic

Links between category theory and transition systems and category theory and logic are the following. For transition systems: the states of the transition system are the objects of the category, and the transitions between two states are the morphisms of the category. For logic, the formulae of the logic are the objects and the proof of formulae are the morphisms.

Symmetric Monoidal Categories for Parallel Distributed Processes

Cap [7] has demonstrated that a parallel non-deterministic distributed process can be modeled with a Symmetric Monoidal Category (SMC) according to a partial order semantics. Partial order theories define partial order relations on actions. Parallelism between two actions occur if there is no relation between them.

A *symmetric monoidal category (SMC)* is a category $\mathcal{C}$ provided with a binary operation on the morphisms and objects (more precisely a bifunctor), $\odot : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. $\odot$ allows a neutral element N in O, $\odot$ is associative, commutative on the objects. It provides $\mathcal{C}$ with a monoidal structure.

A *monoidal-free category* has objects and morphisms generated by a set of atomic objects and atomic morphisms. Atomic objects and atomic morphisms generate the whole category.

A distributed system is made of several processes accessing concurrently a shared information. A process is modeled as a set of modification of the state information. A transition is a modification of the state of the information (not of the state of the processes). Information state is made of several smaller indivisible parts. Below we briefly present how processes, behaviors and observations are modeled by the means of SMC.

- *Processes*
  Objects in the category are local information states and morphisms are transitions between these states. An atomic object can be seen as the smallest unit (an indivisible part) of a local information state. A morphism $f : X \odot Y \to Z$ is a transition from the information state $X \odot Y$ to Z.
  The bifunctor $\odot$ on objects can be seen as the agglomeration of two states. $X \odot Y$ is an information state made of two parts X and Y.
  Parallel composition between transitions is given by the bifunctor on morphisms: $f \odot g : X \odot Z \to Y \odot U$ is the parallel occurrence of $f : X \to Y$ and $g : Z \to U$.
  Sequential composition is given by the operation $\circ$ of morphisms given by the SMC. $f \circ g : X \to Z$ is the sequential composition of $f : X \to Y$ and $g : Y \to Z$.
  A *process* is a safe (local state information appear only once), strict (), and monoidal-free (freely generated by the set of atomic states) SMC.

- *Behavior*
  A process can be non-deterministic, i.e. for a given information state there can be several transitions leading each to identical or different new information state. A behavior is *one* of these state changes.

A behavior is a strict, monoidal-free and safe SMC, which does not contain conflicts or co-conflicts (there is no situation with more than one possibility for executing state changes) and which is finitely causal (every atomic state has a finite number of causes and there is a finite number of initial causes; a state Y has a cause X if there exists a morphisms $t : X \to Y$).

- *Observation*
  Observations bind behaviors to processes. An observation of a process D is a pair (B, F), where B is a behavior and F is functor $F : B \to D$.

Higher-order processes are defined by the use of closed symmetric monoidal categories.

8 Codelets

Executing messengers can be viewed as "swarming" processes which execute separately. This view of messengers resembles the view of a chemical reaction, where molecules all react separately but all at the same time and in the same space. For this reason, the work of Hofstatder[13] considering piece of code working as chemical reactions, is connected to messengers.

In [13], Hofstadter explains several projects related to *analogies* of the form: "if *abc* becomes *abd*, how can we change *ijk*". These analogies are also called fluid concepts: "concepts with flexible boundaries, concepts that can stretch, that can adapt to circumstances".

The architecture used for realizing several applications using the notions of analogies and fluid concepts consists of (1) a *coderack*, which is a queue of *codelets* (small programs); (2) a *slipnet*, which is a set of permanent platonic concepts, (3) a *workspace*, which is an instance of slipnet concepts. Codelets, randomly chosen in the coderack, act on the workspace. They modify the workspace following analogies. The external behavior resembles chemical reactions, in the sense that each codelet contributes to the final result.

The applications realized in that manner are: Jumbo, which implements the joke of Jumble; Numbo for the Numble joke, Letter Spirit for calligraphic styles.

The translation of the codelets paradigm into the messenger paradigm is obvious: the coderack is the set of all executing messengers, the workspace is the dictionary. Codelets are messengers that do not need messenger queues.

9 Other Formalisms

Briefly, we will just mention some other formalisms related either to agents or distributed processes.

Subsumption

Brooks, in [5, 6] argues that intelligence is an *emergent* property of systems, and that there is no need of symbolic AI for an explicit representation and an explicit abstract reasoning of intelligent behavior.

Linear Logic

Linear logic has proven to successfully formalize distributed processes as well as agents. As messengers can be considered distributed processes, or agents, it is interesting to investigate linear logic based formalisms. Cap in [8] provides a calculus for distributed parallel processes which is a mixing of linear logic and algebraic specifications.

10 Conclusion

Messengers have conducted us to consider them under several points of view, each of them is connected to a well known domain of computer science. We found some similar, but different, paradigms, together with their formalisms.

These formalisms come from different domains and deserve different views of the messenger paradigm: multi-agent systems for messenger viewed as agents; generative communication for the communication by shared memory; π-calculus for mobility expressed by the exchange of channel names; Higher-order π-calculus for the exchange of processes; petri nets for concurrency; actors for asynchronous message passing.

This section briefly summarizes the different paradigms listed in this paper, and firstly extracts the specificity of the approach wrt the treatment of interaction between processes, secondly considers them under the point of view of mobility, and finally suggests some improvements to obtain a formalism well suited for the messenger paradigm.

Communication and Paradigms

In the **actor model** we have identified autonomous agents, which communicate asynchronously by message passing. Messages are sent personally to an actor at its mail address. Actors maintain received messages in a personal queue. Actors process a message: perform a local computation, send new messages and create new actors. After processing the message, they change their behavior by performing a *become*, in order to process the next message. Dynamicity, i.e. creation of new actors, at run time is allowed. Several actor spaces are possible. 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, by any agent. Program-tuples execute a well defined sequence of actions (test, consume, local evaluation, out), before dying or replacing their behavior by a new program-tuple, just like actors do when they perform a *become*. Dynamicity, i.e. creation of new program-tuples, at run time is allowed. Several tuple spaces are possible. Differences 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. Coordination and communication is realized through common shared memory.

In the **messenger paradigm**, we have anonymous and autonomous messengers, communicating asynchronously through a shared memory and coordinating their executions by the means of messenger queues. Dynamicity, i.e.

creation of new messengers, and multiple messenger platform are allowed. Messenger can move, but we think that this is also possible in the PoliS approach: creation of a new program-tuple in another tuple space and death of the current program-tuple. Two main differences are to be noted wrt the Polis approach: messengers use messenger queues and shared memory to coordinate their executions, and they are not obliged to perform a well defined sequence of operations. Coordination is realized through shared memory and messenger queues.

In **analogies**, codelets are piece of code which is considered as chemical reactions all executing autonomously and in parallel. The result is given by the contribution of each codelet. Communication occur indirectly through modifications to the workspace.

At our sense, the general communication mechanism between processes should be: (1) asynchronous, because we can built synchronous communication upon asynchronous one, (2) via shared memory, because, we can built message passing upon shared memory, and (3) anonymous, because via a shared memory process can send informations to a specified process (a specified data concerns a specified process).

We are convinced that this type of communication, not only can realize other types of communication, but also is a basis for new types of distributed algorithms.

Mobility and Formalisms

We can classify formalisms according to their degree of expressiveness of mobility.

No mobility or mobility provided by the dynamicity. Indeed, if a formalism allows dynamicity, i.e. the creation at run-time of new processes between several different spaces, then 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 dynamic algebra of actors, some extensions of petri nets, generative communication between multiple tuple spaces.

Mobility à la π-calculus. It is a mobility by reference passing. Processes do not move, but the configuration of communication changes. This way of managing mobility is that of π-calculus and of mobile petri nets [3].

Mobility à la HOπ. It is a "true" mobility where processes can really be sent through channels. We find this type of mobility in the HOπ, and in the messenger paradigm.

Research Topics

Firstly, we can think to extend the above formalisms to the messenger paradigm or mobile processes.

It is possible to adapt the calculus for generative communication to the messenger paradigm: (1) extension to multiple tuple spaces, (2) addition of the notion of queues.

Mobile petri nets of [3], use a mobility à la π-calculus, where the name of the ports are transmitted, i.e. the name of the place is a token. These mobile petri nets can be adapted to stick to HOπ. Tokens are no longer channels, but processes. We could have the following schema: places are allowed to be tokens, and nets are post-conditions of transitions. We could envisage the case,

where nets are allowed to be tokens (in the pre- and post-conditions), mobility is there realized immediately: the net represents a process and the net is a token, the token moves thus the process moves. Thus, we would have several levels of granularity inside the same formalism. Two cases for dynamicity could be envisaged: (1) as in dynamic nets, nets could also be post-conditions of transitions in order to realize the dynamicity; (2) tokens for nets in the post-conditions imply the net to be changed to a new net with more subnets.

Mix of higher-order π-calculus (HOπ) and algebra for generative communication: the idea is to use the mobile feature of HOπ (process passing) and the processes coordination through shared memory of generative communication, in order to obtain a formalism exactly suited to messengers.

Secondly, we can extend the formalisms from the level of processes to the level of families of processes. Questions to answer are: how is it possible to specify a collection of messengers or agents working together, how to represent the changing set of agents participating in the family, the growth or decrease of a family (new agents appear, old agents die), the distribution of the family over several platforms, what are the properties of the family, etc.

References

[1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] Gul Agha, Ian Mason, Scott Smith, and Carolyn Talcott. A Foundation for Actor Computation. In *DIMACS'94*, 1994.

[3] Andrea Asperti and Nadi Busi. Mobile Petri Nets. Technical Report UBLCS-96-10, University of Bologna, 1996.

[4] Olivier Biberstein. *CO-OPN/2 An Object-Oriented Formalism for Concurrent Processes*. PhD thesis, University of Geneva, 1997.

[5] R. A. Brooks. Intelligence without reason. In *IJCAI'91*, pages 569–595, 1991.

[6] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[7] C. H. Cap. An Operational Interpretation for Symmetric Monoidal Categories. Submitted to a conference.

[8] C. H. Cap. A Calculus of Distributed and Parallel Processes - An Approach Using Linear Logic and Algebraic Specification. Technical report, Department of Computer Science, University of Zürich, 1994.

[9] P. Ciancarini. Distributed Programming with Logic Tuple Spaces. Technical Report UBLCS-93-7, University of Bologna, 1993.

[10] P. Ciancarini, R. Gorrieri, and G. Zavattaro. Generative Communication in Process Algebra. Technical Report UBLCS-95-16, University of Bologna, 1995.

[11] Angela Dalmonte and Mauro Gaspari. Modelling Interaction in Agent System. Technical Report UBLCS-95-7, Laboratory for Computer Science, University of Bologna, February 1995.

[12] Mauro Gaspari. Towards an Algebra of Actors. Technical Report UBLCS-96-9, Laboratory for Computer Science, University of Bologna, April 1996.

[13] D. Hofstadter and the Fluid Analogies Research Group. *Fluid Concepts & Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought*. Basic Books, 1995.

[14] Robin Milner. The polyadic π-calculus: a tutorial.

[15] D. Sangiorgi. *Expressing Mobility in Process Algebra*. PhD thesis, University of Edinburgh, 1993.

[16] C. Sibertin-Blanc. Cooperative Nets. In Robert Valette, editor, *Application and Theory of Petri Nets 1994: proceedings*, volume 815 of *LNCS*, pages 471–490. Springer-Verlag, 1994.

[17] M. P. Singh. *A Theoretical Framework for Intentions, Know-How, and Communications*. Number 799 in LNAI. Springer-Verlag, 1994.

[18] M. J. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, University of Manchester, 1992.