An extension to l-calculus for Distributed Functional Actor Programming

Najat Rafalia, Jaafar Abouchabaka
Faculté des sciences, Kénitra, BP. 133, Maroc
Département d'Informatique
Labo de Recherche en Informatique et télécommunication
arafalia@yahoo.com, abouchabaka3@yahoo.fr

ABSTRACT

In this paper, we present the l_{AD}-calculus witch is an elementary functional distributed actor language with a new approach of message communication between actors in a distributed environment. This strategy is based on a static analysis which allows determining the parts of a message that must be transmitted. The actors we consider have a functional script and manipulate the terms of the l_{AD}-calculus. The expressions of this language correspond to those of the l-calculus extent by some actor primitives.

General Terms: Algorithms, Information Treatment, Computer Sciences.

Keywords: Actor Model, Communication, Distributed System, Concurrent Programming, Functional Programming.

Academic Discipline: Computer Sciences.

1. INTRODUCTION

We consider a distributed system of actors. The actors we consider have a functional script and manipulate the terms of the l_{AD}-calculus which is an elementary functional distributed actor language, described in section3. The expressions of this language correspond to those of the l-calculus extent by some actor primitives.
Actors communicate by exchanging messages. A message is a functional term. The message transmission causes the message address to be put in the receiver mail queue. This implantation doesn't involve any problem when the actors are in the same site because they share the common memory. If the actors are in some distant sites, they could not get to each other site memory. So, sending the message reference is insufficient.
We can code and send all the message by tram of octets. This strategy presents some inconveniences in particular for the manipulation of *complex linear structures*. This is the case of the languages using a lazy evaluation. Although we can omit the cost of coding, decoding and sending these structures, we can't omit the difficulty to represent them as a linear stream of characters: consider a tree. To transform a tree into a stream of data, one must specify a traversal order (usually a preorder, depth-first, left-to-right traversal of the tree). A consumer that only needs a portion of the tree may be forced to examine useless portions before it can receive the needed portion. In case in which unneeded portions of the tree are infinities, the consumer may never receive the portion of the tree it needs. Therefore, it's imperative to be sure that all which is sent, will be exploited at most.
We think on a lazy strategy of message communication between actors in a distributed environment.
For each actor susceptible of receiving a message **m** which is a functional term, we determine the part of **m**, that must be sent. This is accomplished by a static analysis of the application code.
When a transmission of a message **m** is valued, we first transmit which the static analysis has detected necessary. During the execution, if other portions are detected necessaries to pursue the treatment, the consumer asks for them dynamically and the producer sends them.
The organization of this paper is as follow:
In the next section, we present the actor model. In section 3, we describe the l_{AD}-calculus. We present our lazy strategy of term communication in section 4. Finally, we conclude by the related work in the last section.

2. THE ACTOR MODEL

The actor model is a model of concurrent computation. It was first described by the group of Massachusetts Institute of Technology (MIT)[7].

Actors

Actors are independent concurrent objects that cooperate and interact by sending asynchronous messages.
An actor is completely described by :

- Its **mail address**, to which there correspond a sufficiently large mail queue, and
- Its **behavior**, which is a function of the accepted messages

Abstractly, we may picture an actor with a **mail queue** on which all communication are placed in the order in which they arrive, and an **actor machine** which points to a particular cell in the mail queue. We represent this as in figure 1.
When an actor machine accepts the n^{th} communication in a mail queue, it will create a new actor machine, $\mathbf{X_{n+1}}$**(become(**$\mathbf{X_{n+1}}$**)**. This new machine will carry out the replacement of the actor. The replacement behavior will process the $(n+1)^{th}$ communication. The mail address of the actor remains invariant.
The actor may also send a communication (**m**) to a specific target actor (**a**) (**send(a,m))**. It also creates a new actor with an initial behavior $\mathbf{X_0}$**(create(**$\mathbf{X_{00}}$**)).** We represent this pictorially as in figure1:

International Journal of Management & Information Technology
Volume 3 No.3, 2013, ISSN 2278-5612

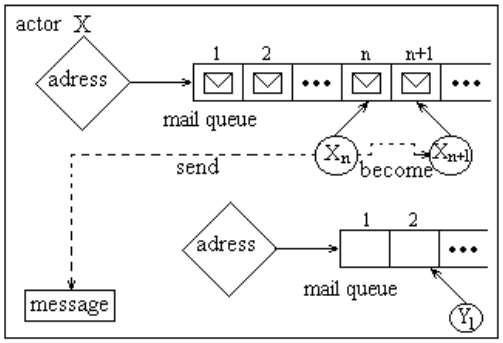

Figure 1: Abstract representation of an actor and its possible actions

3. DISTRIBUTED IMPLEMENTATION OF THE MODEL

The actors constitute a concurrent model for programming. Actors could be distributed on several sites. These sites are joined by a mechanism of communication like in figure 2.
The create primitive (**create(X_{00})**) allocates a unique mail address to the newly created actor, and creates a process which represents the computation potency of this created actor. The system makes an adequacy between the actor and its mail queue. So, the name of an actor and its mail queue address become synonymous.
Each new actor is created in the site having the minimal number of process. It's important to allow the programmer to ignore the details concerning the physical location of the actors in different processors which constitute the network of the program execution. To that effect, every site has a **server actor** (or **actor of communication)**. This actor manages the distribution of the actors and the communication between sites. The messages towards distant sites are addressed to the server actor of the sender site in order to treat and send them to their destination.
In the receiver site, the server actor treats the external messages that arrive and transmit them to their receivers (see figure 2).

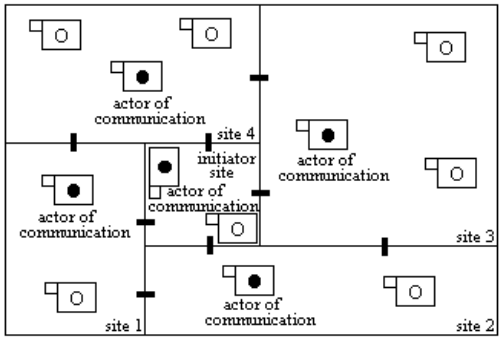

Figure 2: General architecture of communication system

4. I_{AD}-CALCULUS: AN ELEMENTARY FUNCTIONAL DISTRIBUTED ACTOR LANGUAGE

We present the I_{AD}-calculus, a distributed extension of the I-calculus, for the actor model. The I-calculus is an elementary functional language [3].The I_{AD}-calculus constitutes a low level functional distributed actor language to which the high level actor languages could be compiled.
To represent data and the messages, I_{AD}-calculus integrates a structure of term which corresponds to a tree. It also integrates a pattern-matching mechanism in order to recognize the messages.
The actor behavior is an expression of the I_A-calculus extended by the primitives for the creation and the manipulation of the actors and those for the pattern matching and the construction of the terms.

I_{AD}-calculus Syntax

The I_{AD}-calculus expressions are constructed by the terms of the algebra engendered by a set of constructors, a set of variables and the mechanisms of abstraction and application.
The abstraction on a variable of the I_{AD}-calculus is generalized to an abstraction on a pattern which is a term of the algebra engendered by the constructors. The I_{AD}-calculus contains also the primitives **send**, **create** and **become** for the manipulation of the actors.

- **send(a,m)** to send the message **m** to the actor **a.**
- **become(b)** to replace the behavior of the actor executing this primitive, by the behavior **b.**
- **create(b)** to create a new actor with an initial behavior **b**.

In the expressions of the I_{AD}-calculus, it's necessary to distinguish the actor behaviors which could contain functional computations, the primitives **send** and **become**, from the messages or communicable values which are the results of a pure functional computation. These values could be described by the following syntax:
m = x variables, symbols, address, numbers
| C(m,m,….,m) construction of terms

|Create(B{Acquaintances}) creation of an actor
| self the individual actor address

Note that the messages must be partially valued in order to be filtered. The mechanism of patter-matching takes charge of this lazy valuation.
The result of the **create** operation is an actor address which is a communicable value.
The **self** variable designates implicitly the actor which executes the behavior and allows to this actor to send to himself a message.
The behaviors have the following syntax:
B{acquaintances}= I**P.Fa** abstraction on a pattern
||P.Fa, IP.Fa, …., IP.Fa composition of abstractions on several patterns
The actions **Fa** have the following syntax:
Fa = send(**m**,**m**); **Fa**
| create(B{f'{acquaintances}) ; **Fa**
| become(B{acquaintances})

Example 1*:* **consultation and change of the cell value**
The following behavior **Cell{v}** is a behavior of an actor **Cell** which sends the initial value "**v**" to an actor **a** when it receives the message **Pair(Get,a)**. When **Cell** receives the message **Pair(Set,n)**, it changes its initial value "**v**" by the value "**n**"**.**

Cell{v} = IPair(Get, a). send(a,v); become(Cell{v})
IPair(Set, n). become(Cell{n})

The following expression creates an actor 'A' and sends him the message pair(Set,4) :
A = create(Cellf{0}), send (A, Pair(Set,4))

5. LAZY STRATEGY OF MESSAGE COMMUNICATION

The execution of a transmission **send(a,m)**, consists of the valuation of the receiver actor **a** and that of the message **m**.
If the valuation of the actor **a,** detects that this later is in a distant site, then, the transmission of the message address is not sufficient because distant actors could not get to each other site memory. In this case we opt for a lazy transmission between distant sites.
The general problem is presented as follow:
Given a message C(C_1,;C_2;..., C_n) destined to an actor **a***, what are in this message the necessary levels to accomplish the pattern-matching and the treatment of the message by the actor* **a***.*
Our lazy communication strategy consists of two phases:

- **Static analysis phase**: it's accomplished at the time of the compilation. It allows determining the parts of the message, that are necessaries for the pattern-matching and the treatment of this message. These parts are expressed in the level number of the tree which represents the message.
- **Dynamic transmission phase**: because the static analysis is not always informative, several parts of the message are not detected necessary at the time of the compilation. So, this phase allows completing these needs in the execution.

Static Analysis
The static analysis concerns in fact, all the patterns of the application behaviors. The analysis of an initial behavior involves, through the **become** primitive, to analysis the replacement behaviors starting by this initial behavior. It consists of four principal steps:

- **Marking** : through each pattern, the marking phase marks the necessary parts in a message which will be filtered by this pattern and treated by the action corresponding to the successful pattern-matching. An action corresponds to a sequence of several **send** and several **create**, ended by a **become**.
- **Flattening :** marking is don behavior by behavior. A part can be necessary in a behavior and not necessary in other one. The flattening step allows to "**flatten"** the results of marking concerning the same pattern which appears in an initial behavior and in other replacement behaviors from this initial behavior. The ended set of replacement behaviors can be determined through the application code. This warrants the termination of our algorithms.
- **Compilation of the patterns**: the necessary in a pattern is expressed by a number of levels. This phase consists of associating to each pattern the number of its necessary levels.
- **Compilation of the send**: the ad-equation between the patterns and the messages **m** figuring in the **send(a,m)**, and the use of the precedent phase results, allow to compile the transmissions **send(a,m)** into **send(a,m,L)**, where **L** is the number of **m** levels which are necessaries to accomplish the pattern-matching and the treatment of **m** by **a**. In the following, we detail each step.

Marking
The **marking** step consists of **mark "necessary"** or "**not necessary"** each variable of the pattern. In order to formulate the notion of **necessary** and **not necessary**, we conceive an **abstract** domain **AbsP**, which is composed by the **abstract** values of the patterns.
AbsP := 0 Abstract value of a necessary pattern variable
| 1 Abstract value of a not necessary pattern variable
| AbsP(AbsP, …, AbsP) Abstract value of a construction of patterns
We also define a function of abstraction b_{b0} relatively to an initial behavior b_0. b_{b0} associates to each pattern which appears in b_0 its abstract value $b_{b0}(p)$. $b_{b0}(p)$ corresponds to the marking result of **p** when this marking starts from the initial behavior b_0. b_{b0} is defined as follows:
b_{b0}: **P →AbsP**

b_{b0} (x) = 1 ; x is an elementary pattern (or variable of pattern)
$b_{b0}(C(C_1, C_2,..., C_n)) = b_{b0}(C)(b_{b0}(C_1), b_{b0}(C_2),..., b_{b0}(C_n))$
$b_{b0}(C)(b_{b0}(C_1), b_{b0}(C_2),..., b_{b0}(C_n))$ is an abstract constructor which represents the abstract value of the constructor $C(C_1, C_2, ..., C_n)$. $b_{b0}(C)$ = **1** because the rat C of the pattern $C(C_1, C_2, ..., C_n)$ is necessary at least for the pattern-matching.
C_i(i=...n) can be an elementary field whose abstract value is **0** or **1**, or a constructor $C_i(C_{i1},C_{i2},...,C_{in})$ whose abstract value is an abstract constructor $b_{b0}(C_i)(b_{b0}(C_{i1}),b_{b0}(C_{i2}),...,b_{b0}(C_{in}))$. The abstract value $b_{b0}(C_i)$ is **0** or **1,** it's determined by the marking algorithms which we summarize as follow :
We consider the marking of a pattern $C(C_1, C_2, ..., C_n)$ in an initial behavior b_0.

Marking of the necessary to perform the pattern-matching

A message is a functional term which corresponds to a tree. The necessary for its pattern-matching by a given behavior is determined by comparing **breadth wise from left to right**, the different patterns of this behavior. Therefore, each pattern is also a tree, so, this comparison allows determining the level where we can distinguish a pattern from the other ones and then decide which pattern will filter a given message.

Marking the necessary for a message processing

After the pattern matching of a message, the action of the receiver actor consists of a sequence of some **create** and **send** primitives ended by a **become** primitive. So that, we must determine in this message the necessary fields for the processing of the **send**, the **create** and the **become** primitives.

Execution of a send primitive

In order to execute a **send(a,m)** primitive, we must value the receiver actor **a** and the message **m**. The message **m** can depend on the fields of the pattern which we are marking, i.e. m=**f**(C_i,..,C_p) (p<=n) (where C_i,..., C_p are among C_1,...,C_n or among some fields of C_1,...,C_n). So, the valuation of **m** consists of the function **f** valuation. The fields among C_i,...,C_p, which are necessaries to value **f**, are determined by the **strictness analysis** of **f** in its arguments.

Execution of a create primitive (create(b_1{C_i,...,C_p}))

We are interested in the create primitive like create(**b_1{**C_{ii},...,C_p} i.e. in the case where several acquaintances of **b_1** are fields of the patterns which we are marking, (the same remark is valid in the case of a **become** primitive). The newly created actor can be in the same site as the creator actor or in a distant site, this depends on the number of the process in the creator actor site. In the second case, we send the initial behavior **b_1{**C_i,...,C_p}. An environment composed by a set of closures which bind the acquaintances to their values, is associated with this initial behavior. We send with **b_1{**C_1,...,C_p}only the address of this environment, the receiver actor will ask for the values of some acquaintances if need be. So, at this level the $(C_i)_i$ are not marked necessaries or not necessaries.

Execution of a become primitive (become(b_1{C_i,...,C_p}))

In order to determine the need in the fields C_i,...,C_p we analyse the behavior **b_1{**C_i,...,C_p}). This analyses is made recursively through the cases (1), (2) and (3). It concerns only the **send(a,m)** primitives and the **become(b_2{C'_j,...,C'_q})** primitives which appear in the actions of the behavior b_1, but it don't concern the patterns in **b_1**, because at this level we are still marking the pattern $C(C_1,C_2,...,C_n)$ in b_0. At lest one field among C_i,...,C_p must appear among the $(C'_i)_i$ and **m** must be a function of some fields among C_i,...,C_p.

In fact the analysis is done recursively through the become primitives. We begin by an initial behavior and we pass to the replacement behaviors from this initial behavior. The elementary fields of the patterns are marked necessaries or not necessaries at the time of the strictness analysis of the messages **m** *which appear in the* **send(a,m)** *primitives*.

Formulation of the marking analysis

The marking analysis is in fact done by a **strictness analysis** of the **send(a,m)** and **become(b{...})** primitives in their arguments**.** So, we formulate this analysis by giving an appropriate abstraction to each one of those primitives. We note **f#** the abstract version of **f**, **f** can be a function, a constant, a variable, or an operator, ...

- **Abstraction of a send(a,m) primitive**

send# = & x# m#

We define respectively the abstract operator **&** and | | as the Boolean operators **AND** and **OR**.
We consider **m** as a function of the fields C_1,...,C_p (**m=f(C_1...,C_p)**). We are limited to the simple and "first order" functions.
In order to obtain the abstract version **f#** of the
function **f**, we replace every "predefined" function by its abstract version in the script of **f**. Consider **f** as a function with tow arguments **f**(x,y), **f** is strict in x or x is necessary to value **f** if **f**#(0,1)=0, **f** isn't strict in y if **f**#(1,0)=1. The abstract versions of the arithmetic operator and the **IF** function are given as follows :

=# p q = +# p q = -# p q = ÷# p q = *# p q=& p q
IF# p q r = & p (|| q r)
x = **constant**
| variable
| create(b{acquaintances}) valued in to a
constant (the adress of the creator actor)
| **self** considered as a constant

We also use the following rules:
<constant># = 1
v# = v (v is a variable**).**
So, send(x,m=f(C_1,..., C_i , ..., C_p)) is strict in the argument C_i if send#(x#,f#(1,...,1,0,1,...1))=0.

- **Abstraction of a become(b{…}) primitive**

A behavior is a set of couples including a pattern **P_i** and, the action **A_i co**rresponding to this pattern.
b{...}= {<P_1,A_1>,<P_2,A_2>,...,<<P_n,A_n>}
Each action is a sequence of some **create** and **send** primitives ended by a **become** primitive.
A_i={pc_1,...pc_m,ps_1,....,ps_q,pb}

become#(b{...}) = & A_1#, A_2#, ...,A_n#
A_i# = & ps_1#, ..., ps_qq#,pb#
Flattening
The flattening phase consists of mark as necessary a variable of pattern if it's marked necessary in at least one behavior where appear this pattern. We formulate this by a flattening function **Fl_{b0}** relatively to an initial behavior b_0.
When a pattern **p** appears in an initial behavior **b_0** and in **q** different replacement behaviors **$b_1,b_2,...,b_q$(**q>=1), the result of flattening for the pattern **p** is given by the value of the flattening function **Fi_{b0}** applied to **p**. **Fi_{b0}** is defined as follow:
Fl_{b0}(b_{b0}(p),.. $b_{b.,q}$(p)) = Fl_{b0}(Fl_{b0}**(b_{b0}(p),..., $b_{b(q-1)}$(p)), b_{bq}(p))**
The function **Fl** is defined as follow:
Fl : AbsP X AbsP →AbsP
Fl(b_b (x), $b_{b'}$(x)) =: **1** if **b_b(x) ≠ 0 or $b_{b'}$(x))** ≠ 0
//x is an elementary pattern
else **0**
Fl(b_b(C(C_1, C_2, ..., C_n)), b_b'(C(C_1, C_2, ..., C_n))) = **Fl**(b_b(C), $b_{b'(}$(C))(**Fl**(b_b(C_1), $b_{b'}$(C_1)),...,**Fl**(b_b(C_n), $b_{b'}$(C_n))).
where b_b(C(C_1, C_2, ...,C_n)) and $b_{b'}$(C(C_1, C_2, ..., C_n)) are the marking results of the same pattern C(C_1, C_2, ..., C_n) respectively relatives to the behaviors b and b' which contain C(C_1,C_2,...,C_n). We represent an example of flattening in fig.3.

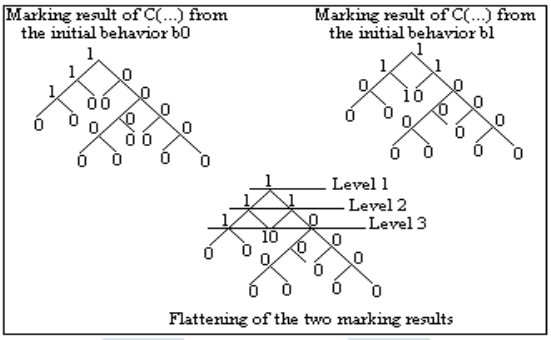

Figure 3: An example of flattening

6. COMPILATION OF PATTERNS

Relatively to an initial behavior b0and at the end of the flattening phase, we know for each knot of the pattern C(C_1,C_2, ...,C_n), its abstract value, **0** or **1**. The number **NecLev(b_0,C(...))** of the necessary levels in the pattern C($C_1$1, C_2, ..., C_n), is defined by the **maximum depth of the knots that are marked necessaries in this pattern**. For example in fig. 3, **NecLev(b_0,C(...))**=3.
If we associate the minimum depth as value to **NecLev(b_0,C(...))** then this later will always be equal to **1** because the rat of the pattern is always marked necessary.
This will increase the number of messages which are necessaries to accomplish the dialogue between the consumer and the producer.
In the code of the application, each pattern **[C(...)]** of each initial behavior b_0 , will be compiled into **[C(...),NecLev(b_0,C(...))]**.

Table of needs

The static analysis determines for each class of actors having the same initial behavior **b_i,** the number of necessary levels in a message **m** which can be filtered and treated by the behavior **b_i.**
We group the different values of **NecLev** in a table called **table of needs** (see table 1). In this table, each value **L=NecLev(b_i,[m])** corresponds to the number of levels in the message **m**, which, each actor having the initial behavior **b**i, needs in order to value **(b_i m)**.
m is the message filtered by the pattern **[m]**.
If the message **m** is not filtered neither by bi nor by any replacement behavior obtained from b_i then **NecLev(b_i,[m])**=0.

Table1: Table of needs

↓	[m_1]	[m_2]	[m_3]
b_1	L_1	0	0
b_2	0	L_2	L_3
b_3	0	L'_2	0

7. COMPILATION OF THE SEND

The ultimate phase of our static analysis concerns the compilation of all the send of the application. At the end of this phase each **send(a,m)** is compiled into **send(a,m,L)**, where **L** is the number of necessary levels of the message **m** in order to be filtered and treated by the receiver actor **a**. **L** is determined by the table of needs according to the initial behavior of **a**. For example, if the initial behavior of **a** is **b_1** 1then **send(a,$m_1$1)** is compiled into **send(a,$m_1$1,L_1).**

Note that if the initial behavior of **a** or the structure of **m** are not known then **send(a,m)** is compiled into **send(a,m,1)**. We send one level because the rat of the message is necessary at least for the pattern-matching. We can't send more because, in this case, we haven't any more information concerning the necessary.
The compilation function **S** is given as follow:
Algorithm 1
S(send(a,m)) =
if **a=create(b)** initial behavior of **a** is **b** and **NecLev(b,m)≠0**
then **send(a,m,NecLev(b,m))**
else
if the initial behavior of **a, B_{init}(a)**, is known
and **NecLev(B_{init}(a),m) ≠ 0**
then **send(a,m,NecLev(B_{init}(a),m))**
else **send(a,m,1)**

8. THE DYNAMIC TRANSMISSION ALGORITHM

At the end of the static analysis, each **send(a,m)** is compiled into **send(a,m L)**. The execution of **send(a,m,L)** consists of sending **L** levels and the address of the remain fields of **m**. These addresses are called distant address. They allow to the server actor to manage the transmission and the concurrent use of the terms which they address.
If one ask for a term through its address, we can send it entirely. When this term has a large or an infinite depth, this transmission will be slow or handicapped. To avoid this problem, we present an algorithm of dynamic transmission in which we distinguish two cases: the term is necessary to finish the pattern-matching or to pursue the treatment of a filtered message.
Algorithm 2
Consider a distant term whose address is **ptr**.

- If this term is necessary to finish the treatment then the term will be completely sent because this necessity was detected during the compilation by the strictness analysis of the treated message.
- If the term is necessary to finish the pattern-matching then we must specify in the request the number

DNecLev(ptr), of the necessary levels. This number is given by:
DNecLev=min – filterlevel
where **min** is the minimum of the level numbers associated, during the compilation, to the patterns which are susceptible of filtering the term.
filterlevel is the level reached by the pattern-matching.
Where the static analysis is not informative **min - filterlevel** can be ≤ 0. In this case, we complete the pattern matching by asking for one level at time. We use the **min** in order to transmit just the necessary. If the pattern-matching is not yet finished then the same procedure is repeated for the next distant address.

9. CONCLUSION

We have presented a lazy strategy of term communication in a distributed environment of actors. It presents the following advantages:

- Use of a reduced number of messages during the dialogue between the consumer and the producer.
- Allow to manipulate infinities structures.
- Uniformity of the transmitted data and the received data.
- Simplicity of the communication: the producer and the consumer exchange the same type of data, so, they will easily communicate.

We have also realized an implementation, a simulation of the static analysis and the dynamic transmission is operational. We simulate in particular, the management of the transmission and the concurrent use of the distant terms.
We are testing it on some consistent benchmarks cost concerning the execution time and the memory space of our communication strategy. This work combines into a global system for the valuation of the actor languages through a distributed virtual machine MVAD. It concerns the definition of the necessary primitives in order to integrate the lazy communication in the MVAD.

REFERENCES:

[1] Kang Lianghan, Cao Donggang. An extension to Computing Element in Erlang for Actor Based Concurrent Program+ming 2012 IEEE, 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops.
[2] Ruben Vermeech, Concurrency in Erlang!§§ & Scala: The actor model. Retrieved from: http://ruben.savane.be/articles/concurency-in-erlang-scala. Last modified January 2012.
[3] Greg Michaelson. An Introduction to Functional Programming Through Lambda Calculus. (first published 1989). Published at August 2011 by Dover Publications.
[4] A.Elfaker, M. Pantel, P. Sallé and X. Massoutié. Vers une Machine Virtuelle pour l'évaluation des langages d'acteurs. LMO'95, Nancy pages 221-239, Octobre 1995.
[5] Gul Agha,Chris Houck and Rajenda Panwar, University of Illinois at Urbana, II 61801, USA. Distributed Execution of Actor Programs Sciences Forth workshop on language and compiler for parallel computing. August 1991.

[6] Henry E.Bal, Adrew S. Tanenbaum, Department of Mathematics and Computer Sciences, Vrije University, Amsterdam, Jennifer G. Steiner, centrum voor wiskinde en Informatica, Amsterdam, the Netherlands. Programming Language for Distributed System. ACM Computing Surveys, Vol 21, September 1989.
[7] Gul Agha (1986), Doctoral Dissertation, Mit Press. Actors: a Model of Concurrent Computation in Distributed-Systems. https://dspace.mit.edu/handle/1721.1/6952.Osl.cs.uiuc.edu.http://osl.cs.uiuc.edu. Retrieved 2012-12-02.

AUTHOR PROFILES

Dr. Najat Rafalia, she has obtained her doctorate in Computer Sciences at Mohammed V University, Rabat, Morocco. Currently she is a professor at Ibn Tofail University, Department of Computer Sciences, Kénitra, Morocco. Her research interests are in distributed systems, concurrent and parallel programming, communication and multi agent systems.
Dr. Jaafar Abouchabaka, he has obtained two doctorates in Computer Sciences applied to mathematics at Mohammed V University, Rabat, Morocco. Currently he is a professor at Ibn Tofail University, Department of computer Sciences, Kénitra, Morocco. His research interests are in concurrent and parallel programming, distributed systems, and multi agent systems.