# Cognitive Agents Learning by Communicating

Clement Jonquet, Stefano A. Cerri

# Cognitive Agents Learning by Communicating

**Clement Jonquet - Stefano A. Cerri**

LIRMM - University Montpellier 2
161, rue Ada
34392 Montpellier Cedex 5 - France
{cerri, jonquet}@lirmm.fr

*ABSTRACT. Cognitive Agent communication is a research field in full development. We propose here an extension and an implementation of the STROBE model, which regards the Agents as Scheme interpreters. These Agents are able to interpret messages in a dedicated environment including an interpreter that learns from the current conversation. These interpreters evolve dynamically, progressively with the conversations, and thus represent evolving meta level Agent knowledge. We illustrate this theoretical model by a "teacher-student" dialogue experimentation, where an Agent learns a new performative at the completion of the conversation. Details of the implementation are not provided here, but are available.*

*RÉSUMÉ. La communication entre Agents cognitifs est un domaine de recherche en pleine effervescence. Nous proposons ici un modèle, basé sur le modèle STROBE, qui considère les Agents comme des interpréteurs Scheme. Ces Agents sont capables d'interpréter des messages dans un environnement donné incluant un interpréteur qui apprend de la conversation. Ces interpréteurs peuvent en outre évoluer dynamiquement au fur et à mesure des conversations et ils représentent la connaissance de ces Agents au niveau méta. Nous illustrons ce modèle théorique par une expérimentation de dialogue de type « professeur-élève », où un Agent apprend un nouveau performatif à l'issue de la conversation. Les détails de l'implémentation ne sont pas fournis ici, mais sont disponibles.*

*KEYWORDS: Agent communication, message dynamic interpretation, reflexivity, reification, reifying procedures, STROBE model, ACL*

*MOTS-CLÉS : communication Agent, interprétation dynamique de message, reification, réflexivité, reifying procedures, STROBE*

## 1. Introduction

Knowledge communication or knowledge exchange is something very important for all human societies; it provides evolution and adaptation of these societies through time. One can't imagine where humanity would be if each generation had to learn again how to cut flints or control fire. Fortunately, this does not happen because human beings have both a learning and an important adaptation ability which is neither foreseeable nor measurable. It is not the same for data-processing entities! Indeed, to ensure knowledge communication, learning and adaptability of Agent societies is a hard research subject still in its infancy. We try to identify, here, a small stone to this enormous building by proposing a model of knowledge learning (we can also say knowledge exchange), based on the communication. Our work is the result of combining two research domain ie. Language interpretation and Agent communication. Our idea is to benefit from the learning side effect of communication. Certainly, the goal of education is to change the interlocutor's state. **This change is done after evaluating (1st domain) new elements brought by the communication (2nd domain)**. We propose in this article a learning-by-

being-told model that allows carrying out this change. Precisely, we present a model, based on the STROBE model [CER 99], which regards Agents as Scheme interpreters. These Agents are able to interpret communication messages in a given environment, including an interpreter, dedicated to the current conversation. Moreover, we will show how these interpreters, which represent the Agents, can dynamically adapt their way of interpreting messages. Thus, an Agent learns, through a communication, more than simple Information, it modifies its way of seeing this Information and acquires the ability to integrate new one. We will illustrate this by "teacher-student" dialogue experimentation.

The rest of the paper is organized as follows: section 2 proposes an outline of problems about communication and learning in a MAS (Multi-Agent System), here we will also try to define an "ideal" scenario for the evolution of Agents societies, and Agents communication in the future; section 3 and 4 present our model which regards Agents as Scheme interpreters and provides them a set of pairs (environment, interpreter). Section 5 introduces the devices that enable us to make Agent interpreters evolve dynamically, specifically the mechanisms of reflexivity and reification. The experimentation itself will be developed as well as the mechanism of reifying procedures, in section 6. Finally, we will attempt to discuss interests and extensions of our model.

## 2. Communication and learning in MAS

Simply grouping together several Agents is not enough to form a MAS, it is the communication between these Agents that makes it. Communication allows co-operation and coordination between Agents[1] [FER 95]. Defining and modelling communication have always been difficult. Nowadays, we can find many communication languages but could one say they are adapted to the Agent world or to new forms of communication such as those proposed on the Web? We can hardly take traditional languages or even current programming paradigms and adapt them to the Web and Agents. **We have to develop new architectures and new languages designed for the Web and the Agents**. Indeed, traditional languages are frozen and it is often very difficult to make them move. To be effective, the communication mechanisms must be intrinsic to a language[2]. For example, for knowledge learning, the *data* level is not enough, the *control* level and the *interpreter* level of these programs are necessary. A Java object, for instance, is able to store data but it can hardly modify its own structure. Our goal is to provide a model that can do it and, at the same time, as simply as possible. In our approach, expressive power of language constructs has the same priority as cognitive simplicity for building effective, new applications solving complex problems on the Web.

We can notice some interesting requirements that an Agent communication should provide: If we consider communication effects on the interlocutors, then we must consider that Agents can change their goal or point of view during the communication. Therefore, **they must be autonomous and should themselves adapt their way of "thinking" during communication** [CER 99]. We have also to consider the fact that Agents can interact with each other or with humans following the same principles [MCC 89] [CER 00]. **What is important is the Agent's representation of its interlocutors**. Semantics of exchanged data is also to take into account. For the moment, the concept of ontology is the main answer to this question but as we will see others ones exist.

Historically, MASs were equipped with an integrated communication language working in an ad-hoc way. Nowadays, the MAS community tends to use Agent Communication Languages (ACLs) applicable to as many Agents interactions as possible. Indeed, providing a strong ACL with a strong semantic gives a large advantage for MAS creation and evolution [DIG 00]. These ACLs are based on the speech act theory[3]. Traditionally, KQML or FIPA-ACL messages provide an element that corresponds to the ontology used in the communication. This makes ACLs independent of any vocabulary and gives to the interlocutor the relation between the concept and the meaning of the message content elements. Nowadays, a specific or ad-hoc ACL is not incorporated in MASs but an ontology is built and given in messages parameters. We propose in this paper an alternative to this established practice. ACLs are often criticized on the number and the kind of performatives they provide. Our experimentation proposes an example of solution to this problem. It illustrates a technique to diffuse new performatives in MAS, by enabling Agents to learn-by-being-told.

---

[1] Even in a reactive approach where the communication is done via the environment.
[2] It means that the language must be naturally and previously communication oriented with devices or structures it proposes.
[3] This model come from language philosophy ([AUS 70], [SEA 71]).

In reaction to all these languages, many models of communication have been proposed. An alternative to the way of considering Agents is presented in [MAR 01]. Among other things, STROBE [CER 99] is interested in significant principles for a communication being based on the three simple Scheme primitives: STReam, OBject, and Environment. It supports different points such as interlocutor representation, history conservation of a conversation, learning-by-being-told etc. We will often refer to STROBE proposals because our model is inspired from it.

Now let us imagine an "ideal" scenario, such as those outlined in [IST 01], of what could be a SMA in a few years. It considers all the Web entities as Agents of the same society that can naturally communicate together and transmit their knowledge. This society could extend without any limit. In this MAS, each Agent is initialised with a necessary minimum of knowledge (to interact) and with a special knowledge characterizing it and transmittable to the others. These Agents have a set of messages interpreters that represent their knowledge and its evolution in time. They progressively learn by communicating. They even learn how to learn and how to teach! For each Agent with which they communicate, they have a specific representation of this Agent, which enables to take into account what they learn while keeping their original behaviour and beliefs intact. Of course, Agents may analyse the representations they have of other Agents in order to decide to change their own. Therefore, knowledge may be transferred step by step in the whole society. From a co-operation/coordination point of view an Agent can require another one to interpret on its behalf a program and return the result. As in Grid architectures [DER 01][CER 02], an Agent can also transmit to another the interpreter that allows it to realise its task. Moreover, these interpreters can be transmitted before a conversation, as nowadays ontologies are transmitted. Considering the fact that no interpreter evolves in the same way, because two conversations are never the same, this society of evolutionary Agents would progressively acquire an extraordinary knowledge richness. Its evolution becomes totally unforeseeable. In this society, all tasks or jobs are realised by dialoguing. New Agent integration is done naturally and gradually. It would not be possible to prove theoretically that an Agent achieves a particular task; the only way would be to look at the emergent solutions which appear when a problem arises.

We try in this article to propose some ideas to progress towards this still utopian scenario. Among other things, we will see how an Agent, which has several messages interpreters, can modify them dynamically to evolve progressively during the conversations.

## 3. Agents as Scheme interpreters

The proposed model considers Agents as interpreters (for both messages and their content). This idea comes from STROBE that considers Agents as REPL interpreters (Read, Eval, Print, Listen)[4]. While it communicates, each Agent executes a REPL loop that is overlapping with the others ones. This principle is important because it regards Agents as autonomous entities whose interactions are functionally controlled by a concrete messages evaluation procedure. Our model uses Scheme for the messages contents as well as for their representation. In this way we can use the same interpreter to evaluate the message and its contents. Example of Scheme expression represented by messages:

```
> (define x 2)      ⇔          > (assertion (define x 2))
: x                 ⇔          : (ack x)
> x                 ⇔          > (request x)
: 2                 ⇔          : (answer 2)
```

Assuming this point of view because all the advantages related to Scheme for knowledge representation profit to Agents, in particular the control model provided by first class procedures and first class continuations, and the memory model provided by first class environments. For example, STROBE proposes an environment structure preserving the history of values, i.e. bindings are not any more pairs like (var val) but are stream like (var val1 val2…valn…). This structure becomes accessible to Agents represented by interpreters.

To implement this model we wrote a Scheme meta-evaluator that recognizes a certain language (for message content interpretation) and we added to it a messages interpretation module (for message interpretation). Scheme is very useful because it is easy to conceive the interpretation process by abstracting both on the memory model

---

[4] Traditionally, an interpreter Loop is made of three step (REP) but in our case, we consider that Listen is important because it characterises the choice of the message to process (see *Scheduling Algorithms* section 7).

(by abstracting on the environment [CER 99]) and on the control model (by abstracting on continuations [QUE 00]). It clarifies the three levels learning or knowledge representation that we can find in all languages: *data*, *control* and *interpretation*. *Data* level learning consists in assigning values to already existing variables, or defining new data, ie. Expression such as `(define a 3)` or `(set! a 4)` to define data; *control* level learning consists in defining new functions abstracting on the existing one, ie. Expression such as `(define foo (lambda …))` or `(define (foo …) …)`. And finally, the most interesting for us, *interpretation* level learning or, meta-level learning, consists in making evolve the Scheme interpreter itself, that means, modify `evaluate` procedure by adding new special form process. That is why, while making evolve its interpreter, an Agent learns more than simple Information; it completely changes its way of perceiving and processing this Information. Here is the difference between learning a datum and learning how to process a class of data.

## 4. Representation of the others

STROBE message evaluation cause on the receiver Agent a certain behaviour. In particular, updating its "partner model". Actually, for the representation of each partner, STROBE proposes to have a partner model to be able to rebuild its internal state. This model proposes to interpret each dialogue in a pair of environments: the first, private, belongs to the Agent and the second represents the current partner model. This is called Cognitive Environment [CER 96] It allows an Agent to take into account or not (according to specific criteria) an information. Our work exploits this concept. In fact, it is based on because, as the Cognitive Environment concept provides to the Agents a global environment (or private) and several local environments representing the others, **our proposal provides to the Agents not a message evaluator but several evaluators, including a global (or private) one and a specific one for each Agent they have a representation of**. Thus, messages interpretation is done in both a given environment and an interpreter. Our work is based on this concept of Cognitive Environment because, to be accessible, these interpreters must be themselves stored in these environments. Thus our Agents have the three following attributes:

- GlobalEnv their global environment.
- GlobalInter their global interpreter.
- Other = {(name, interpreter, environment)} a set of triplets corresponding to the representations of the others.

Their global environment is private and does not change. It is cloned[5] when a new conversation starts, and it is the clone, stored in an element of Other, which is progressively modified along the conversation. Figure 1 illustrates these representations. It means that, for the Agent A, $Inter_B$ is at the beginning a copy of $GlobalInter_B$ and evolve differently during the conversation.



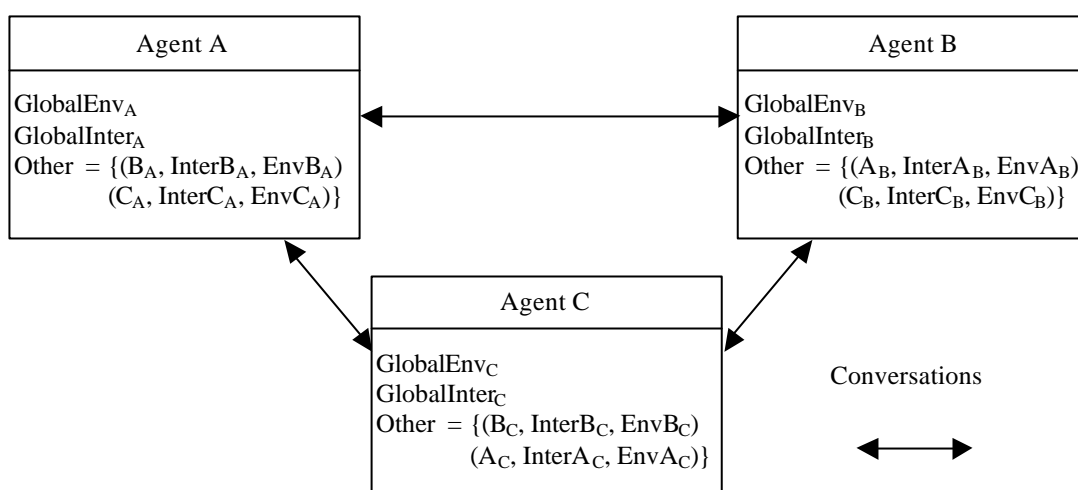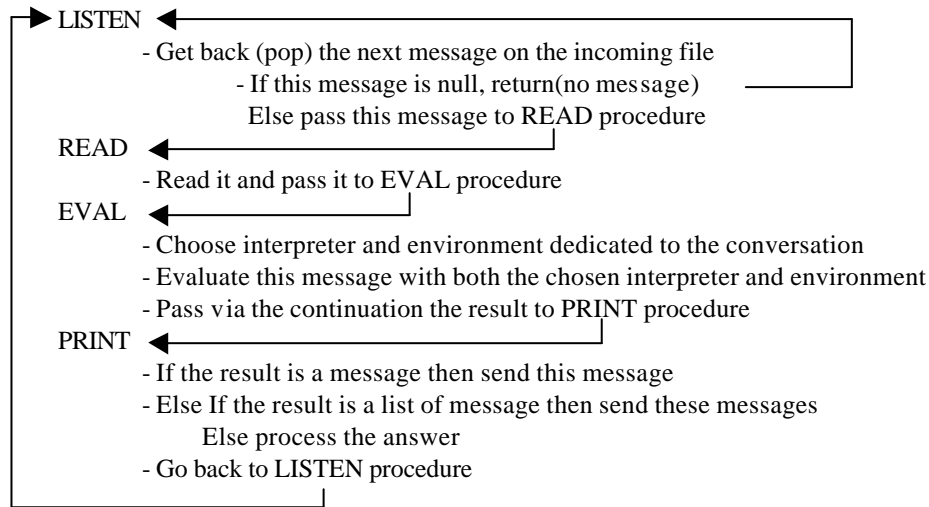**Figure 1. The three attributes that define an Agent and its representations.**

---

[5] Not necessarily if we consider an Agent that would take back a conversation in another context, already existing (with this environment and this interpreter).

The next figure (Figure 2) show our Agents behaviour, consisting in applying the REPL loop see below.

```
┌──► LISTEN ◄──────────────────────────────────────┐
│        - Get back (pop) the next message on the incoming file │
│            - If this message is null, return(no message) ──────┘
│            Else pass this message to READ procedure
│    READ ◄──────────────────────────────────┐
│        - Read it and pass it to EVAL procedure │
│    EVAL ◄──────────────────────────────┐
│        - Choose interpreter and environment dedicated to the conversation
│        - Evaluate this message with both the chosen interpreter and environment
│        - Pass via the continuation the result to PRINT procedure
│    PRINT ◄─────────────────────────────────────┘
│        - If the result is a message then send this message
│        - Else If the result is a list of message then send these messages
│              Else process the answer
│        - Go back to LISTEN procedure
└─────────────────────────────────────────
```

**Figure 2. The REPL loop of our Agents.**

Now we have both our Agent structure and their representations structure. In order to ensure learning by communication, it is now necessary to clarify devices that allow us to dynamically and progressively modify, by conversations, Agent interpreter(s).

## 5. Dynamic modification of an interpreter

### 5.1. The Scheme meta-evaluator

The meta-evaluator we used comes from the famous article written by Jefferson and Friedman, "A Simple Reflective Interpreter" [FRI 92]. This evaluator is user friendly (both easy to use and learn) and provides the reflexivity property we need. Moreover, it proposes the reifying procedure mechanism, which as we will further show, allows to access to an user's program execution context[6] and, if it is required, to modify it. This evaluator has the signature `(evaluate expression environment continuation)`. It is mainly defined by two procedures `evaluate` and `apply-procedure`[7], corresponding to the two steps of the environment evaluation.

### 5.2. Reflexivity and reification

About reflexivity, the evaluator proposed in [FRI 92] is already implemented in a reflexive way. In order to be evaluated by itself the evaluator code is written in the sub-language of Scheme that it processes. But Jefferson and Friedman also propose, in their article, an architecture called reflexive tower, a tower of interpreters. The interpreter at the bottom of the tower executes user input, and every other interpreter in the tower executes the interpreter immediately below it. Two mechanisms allow "moving" in the tower in order to evaluate the code at any level: reflexivity allows going up and reification allows going down. These two points are particularly interesting to us.

Reifying procedures provide the mechanism required to access to the execution context of the interpreter below. The main idea is that user's programs could have the same access rights as the interpreter itself. Owing to that, procedures implementing the evaluator can be accessed and modified by user's programs in the same way as the environment and the continuation. This property makes reifying procedures the ideal tool to dynamically modify our interpreters. Indeed, access to the execution context means access to the environment in which procedures defining the evaluator are stored and thus, means being able to modify them. Furthermore, as we saw it, reification is the method to go down in the evaluation levels. So, to apply a reifying procedure we need two

---

[6] The execution context of an expression is defined by the environment of evaluation of this expression and the continuation to give to this evaluation.
[7] Corresponding to `eval` and `apply` of [ABE 96] and `evaluate` and `invoke` of [QUE 94].

levels of evaluation. Therefore, the whole experimentation presented after must be evaluated by our meta-evaluator to carry out the first call to evaluate, the second being carried out by the Agent itself when it receives a message. Thus, reflexivity of this one is obligatory: our meta-evaluator must be able to be evaluated itself.

### 5.3. Communication protocol

The messages we consider are inspired by the KQML or FIPA ACL[8] message structure. They will be identified from now on `kqmlmsg`, with the signature:

```
(kqmlmsg performative sender receiver content)
```

In order our meta-evaluator to interpret these `kqmlmsg` messages, it is necessary to add to it a test which recognizes them in the `evaluate` procedure, and a function which treats them by evaluating their contents, `evaluate-kqmlmsg`. Used performatives are: *assertion*, *request*, *order*, *ack*, *answer*, *executed* (cf. table 2 [CER 99]) and, as we will see, *broadcast*, which is the subject of the experimentation below. When an Agent receives a message indexed by an unknown performative it answers by a message with `(no-such-performative performative)` content:

- *assertion* messages modify interlocutor behaviour or some of is representations. Their answers are acknowledgement (*ack*) message reporting a success or an error.
- *request* messages ask for one interlocutor representation, such as the value of a variable or the closure of a function. Their *answer* returns a value or an error.
- *order* messages require the interlocutor to apply a procedure. This interlocutor sends the result as the content of an *executed* message.
- *broadcast* messages consist in sending a message with a pair as content `(perform, content)` that means that the interlocutor must send a message with the performative `perform` and with the content `content` to all is current interlocutors. There is no answer defined for the broadcast messages.[9]

## 6. Experimentation: a "teacher-student" dialogue

The main idea of our work is to benefit from the learning side effect of communication. Indeed, the goal of education is to change the interlocutor's state. This change is done after evaluating new elements brought by the communication. To do that, we will use the reflexivity and reification properties already seen. With our reflective interpreter, we show that an Agent can modify its way of seeing things (i.e. of evaluating messages) by "re-evaluating" its own evaluator during communicating. The experimentation we present here is a standard "teacher-student" dialogue.

An Agent *teacher* asks to another Agent *student* to broadcast a message to all its correspondents. However, *student* does not initially know the performative used by *teacher*. So, *teacher* transmits it a series of messages (*assertion* and *order*) clarifying *student* the way of processing this performative. Finally, *teacher* formulates again its request to *student* and obtains, this time, satisfaction. Figure 3 describes the exact dialogue performed in the experimentation.

For the experimentation we developed some Agents able to communicate, i.e. exchanging messages together producing significant answers (following the defined protocol)[10]. They do not do anything when they do not communicate and their autonomy is defined by the fact that they can learn. They have the following attributes: name, globalEnv, globalInter, other, two files of messages (in/out), and a data structure storing the current conversations. Their behaviour consists in applying the REPL loop (Figure 2). Notice that however, this Agents (and our model) must be completed by a classic learning model and KRS (Knowledge Representation System) in order to process information and infer (gather/conclude) on this one. We did not work on these domains but our model is part of a global logic in Artificial Intelligence.

---

[8] Our protocol being extremely simplified, our messages specify only 4 parameters. However, we could have implemented other ones: language, ontology, in-reply-to, reply-with, etc.

[9] *broadcast* is a meta-performative, in the sense that it requires other Agents to evaluate any performative.

[10] Our Agents are in fact Scheme programmed object as we can see in the Normak article *Simulation of Object-Oriented and Mechanisms in Scheme* [NOR 91]

| TEACHER | STUDENT |
| --- | --- |
| Here is the definition of square procedure:<br><br>`(kqmlmsg 'assertion teacher student '(define (square x) (* x x)))` | Ok, I know now this procedure:<br><br>`(kqmlmsg 'ack student teacher '(*.*))` |
| Broadcast to all your current correspondents:<br><br>`(kqmlmsg 'broadcast teacher student '(order (square 3)))` | Sorry, I don't know this performative:<br><br>`(kqmlmsg 'answer student teacher `'(no-such-performative broadcast))` |
| Ok, here is the method to add this performative to those you know:<br><br>Here is the code you have to generate and add to your `evaluate-kqmlmsg` function:<br><br>`(kqmlmsg 'assertion teacher student learn-broadcast-code-msg)` | Ok, I have added this code in a binding of my environment:<br><br>`(kqmlmsg 'ack student teacher '(*.*))` |
| After, here is the reifying procedure which allows you to change this code:<br><br>`(kqmlmsg 'assertion teacher student learn-broadcast-msg)` | Ok, I know now this procedure:<br><br>`(kqmlmsg 'ack student teacher '(*.*))` |
| Run this procedure:<br><br>`(kqmlmsg 'order teacher student call-learn-broadcast)` | Ok, I have just modified my evaluator:<br><br>`(kqmlmsg 'executed student teacher '(*.*))` |
| Broadcast to all your current correspondents:<br><br>`(kqmlmsg 'broadcast teacher student '(order (square 3)))` | Ok, I broadcast... |

**Figure 3 . Broadcast teaching "teacher-student" dialogue.**

Based on these explanations, we can now describe precisely how the process of the new performative learning is carried out. We will detail *teacher* messages sent to *student* in order to modify its interpreter. The first message is `learn-broadcast-code-msg`. This variable corresponds in the `teacher` environment to the new `evaluate-kqmlmsg` function code that *student* must generate to, later, assign it to its function. This code is built by *student* by recovering its `evaluate-kqmlmsg` function body and by adding to it `(if (eq? performative 'broadcast)` and associated treatment. It is a constructivist learning point of view[11]. Thus, *teacher* environment has the following binding:

```
learn-broadcast-code-msg:
 `(define learn-broadcast-code
     (let ((newproc
        (let ((oldproc *evaluate-kqmlmsg code recorvering*))
            *oldproc is updated by adding new code (if (eq? performative…*)
        newproc))
```

The second message sent by *teacher* is the most significant. It defines the reifying procedure `learn-broadcast` that will modify *student* evaluator. *Teacher* environment has also the following binding:

```
learn-broadcast-msg :
 `(define learn-broadcast (compound-to-reifier
                            (lambda (e r k) (evaluate (car e) r k))))
```

---

[11] It is important to notice that it is *student* who rebuild its function with the existing one in its environment and not *teacher* *who* send its `evaluate-kqmlmsg` function code. It allows *student* to keep some previous modification of its function.

It is the `compound-to-reifier` procedure which create the reifying procedure. Then, during its evaluation, `learn-broadcast` is transformed into a call to a compound procedure with the following arguments: list of its arguments, the environment in which it must be evaluated and the continuation which it is necessary to give to this evaluation. Then, the execution context of `learn-broadcast` becomes accessible and modifiable. In our case, `learn-broadcast` evaluates `car` of its arguments list. Finally, *teacher* solicits *student* to apply its function with a call to `set!` (modifing `evaluate-kqmlmsg` by the code generated before). Figure 4 illustrates the steps of evaluation of this reifying procedure. Finally, the last binding is:

**call-learn-broadcast :**
```
'(learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
```

After the last message process, *student* `evaluate-kqmlmsg` function is modified as well as its messages interpreter. The corresponding function code in its environment dedicated to this conversation is changed. Then *student* Agent can process broadcast messages.

When the call to `learn-broadcast` combination is evaluated...

```
(evaluate
   (learn-broadcast (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)                                                      12
```

...`learn-broadcast` is changed by its value in *student* environment...

```
(evaluate
   ((reifier (e r k) (evaluate (car e) r k) *good env*)             13
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```

...then the reifying procedure is transformed in a `compound` procedure...

```
(evaluate
   ((compound (e r k) (evaluate (car e) r k) *good env*)
    (set! evaluate-kqmlmsg learn-broadcast-code))
   *good env*
   *good cont*)
```

...and `evaluate` transfer to `apply-procedure`...

```
(apply-procedure
   (evaluate (car e) r k)
   ((set! evaluate-kqmlmsg learn-broadcast-code))
   *good cont*)
```

...which itself recall to `evaluate` procedure giving access to `*good env*` and `*good cont*`.

```
(evaluate
   (evaluate (car e) r k)
   ((e ((set! evaluate-kqmlmsg learn-broadcast-code)))
    (r *good env*)
    (k *good cont*)
    *good env*)
   *good cont*)
```

**Figure 4 . Steps of the learn-broadcast reifying procedure evaluation.**

---

[12] *good env* and *good cont* correspond to the environment and the interpreter dedicated to the conversation.

[13] A closure is stored in the environment as (`name type parameter body defenv`). `type` can be `compound`, `reifier` or `primitive`, `defenv` is the definition environment.

## 7. Interests and extension

Our experimentation shows how to add a new performative to the ones already known by an Agent, thus how to modify an Agent messages interpretation function. However, the same principles can be used to modify any part of an Agent interpreter. For instance, we could have made an example which adds `cond` or `let*` to our language recognized by our meta-evaluator. Even more, an Agent could teach to another how to make its evaluator lazy by changing some functions (`evaluate` and `apply-procedure`). With this protocol, our Agents have a set of interpreters that represent their knowledge. Indeed, these interpreters correspond to their recognized sub-languages and thus to their faculties to carry out a task. As seen in the "ideal" scenario, Agents can process programs for others or even exchange their interpreters[14] just like in Grid architectures where is more interesting to move programs than data. Their interpreters can also be transmitted before a conversation just like an ontology. In fact, the "ontology" abstraction in ACLs is, therefore, extended by our model with an abstraction on the ACL itself. This may allow to experiment with ACLs equipped with different semantics [GUE 02] in order to choose, in a specific context the most adequate ACL. The ontology becomes intrinsic to the communication language.

Let us imagine an Agent society or MAS that follows this model. Any Agent can learn something from another. If an Agent is built with a minimum of knowledge (to interact) and a speciality, then it can diffuse it progressively with its conversations. These principles are very interesting for the Web. Let us consider a new Web application server Agent that uses a set of performatives corresponding exactly to its job. If it is built with the potential to teach these performatives then it will easily be integrated into an Agents society by teaching these. Moreover, to make the analogy with XML (eXtensible Markup Language), we can consider a DTD, an XML-Schema or specifically a XSL style sheet as a XML data interpreter. Then the presented model allows these "interpreters" to evolve dynamically and idem for the associated documents XML; giving to the Web dynamism and adaptability. This can easily be done considering the analogy between Scheme and XML, because XML documents are represented by tree and Scheme fit to process on. In the same idea, many languages linking S-expression formalism (Scheme) and XML appear [KIS 02].

The presented work follows the same line as the language C+C [CER 00] and the work reported in [GUE 99] that propose to provide to the Agents a dynamic *Scheduling Algorithm.* When an object answers to a method call it does not care or ask itself why it must answer. However, an Agent behaviour depends on its *Scheduling Algorithm* that enables it to be autonomous and to decide if and when it devotes time to the others Agents. If an object implementation is fixed, an Agent has a scheduling algorithm evolving during the time. Our reflective interpreters can embody dynamic *Scheduling Algorithms*.

Principles presented here could also be useful in others scenarios. For example, instead of the procedure square definition, let us imagine that *teacher* transfers to *student*, a procedure that implements an optimised algorithm to solve a problem. Consider, for instance, `(memo-fib n)`, which is the Fibonacci algorithm version that takes into account the memoization[15] principle [ABE 96], transforming an exponential Fibonacci algorithm into a linear one (an example of dynamic programming). In this case, *student*, after broadcast learning, could choose some agents to perform some heavy computation using the Fibonacci algorithm as follow: It can asks all its current correspondents to process a Fibonacci number and after receiving all the answers, compares answer times to decide which of them are selected. It can do it because it has its own reference for this processing. This idea could be particularly interesting for communication protocols such as *conract net*, or for Grid computing where "effective" agents have to be selected to perform heavy computation.

An other submitted paper [JON 03] describes also how using nondeterministic interpreter in our model can enable dynamic specification of a problem, in order to fit with dynamic service generation scenarios, such as necessary on Grids. The paper give a typical e-commerce scenario example.

---

[14] It comes from the idea quoted from [ABE 96]: "*If we wish to discuss some aspect of a proposed modification to Lisp with another member of the Lisp community, we can supply an evaluator that embodies the change. The recipient can then experiment with the new evaluator and send back comments as further modifications*"

[15] Memoization (also called tabulation) is a technique that enables a procedure to record, in a local table, values that have previously been calculated. This technique can make a vast difference in the performance of a program. A memoized procedure maintains a table in which values of previous calls are stored; when it computes a value, it first checks the table to see if the value is already there, if so, just returns that value. Otherwise, it computes the new value and stores this in the table.

This model seems to have a lot of interests for several domains but it is still in its infancy. As already said, it is part of a global logic in Artificial Intelligence and to this model some features need to be added. For example a security one (helping an Agent to make a choice if some local environments are in opposite) or a self-learning one (helping an Agent to decide to modify its own structure (GlobalEnv and GlobalInter)). This could be the subject of further works.

## 8. Conclusion

We tried to show in this paper a learning method for cognitive Agents based on communication. This learning process can be realised by simple communication (*data* or *control* level), or by Agent internal modification (*interpreter* level). If Agents interpret in a dynamic way their messages, they become adaptable and, without any external intervention, can communicate with entities that they have never met before. Moreover, as their evaluator is modified to acquire knowledge, it could be also modified to learn how to teach knowledge. Then knowledge exchange would progressively become possible and pertinent with communications. This paper does not simply propose another programming artefact to add to Agents, but the idea is rather to show a technique, simple, usable and friendly, of autonomous evolution of Agents in a society. This architecture provides dialogue based problem resolution which is, even and especially in human societies, a very promising method.

## 9. References

[AUS 70] Austin J.L., *Quand dire c'est faire*, Edition du seuil, Paris, 1970.

[ABE 96] Abelson H., Sussman G.J., Sussman J., *Structure and Intepretation of Computer Programs*, Second Edition, MIT Press, Cambridge, Massachusetts, 1996.

[CER 96] Cerri S. A., "Cognitive Environments in the STROBE Model". Presented at EuroAIED: *the European Conference in Artificial Intelligence and Education*, Lisbon, Portugal, 1996.

[CER 99] Cerri S.A., "Shifiting the Focus from Control to communication: The STReams OBject Environments (STROBE) model of communicating Agents", In Padget, J.A. (ed.) *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing*, Berlin, Heidelberg, New York: Springer-Verlag, Lecture Notes in Artificial Intelligence, pp. 71-101, 1999.

[CER 00] Cerri S.A., Sallantin J., Castro E., Maraschi D. "Steps towards C+C: a Language for Interactions", In Cerri, S. A., Dochev, D. (eds), *AIMSA2000: Artificial Intelligence: Methodology, Systems, Applications*, Berlin, Heidelberg, New York: Springer Verlag, Lecture Notes in Artificial Intelligence, pp. 33-46, 2000.

[CER 02] Cerri S.A., "Human an Artificial Agent's Conversations on the Grid", *Electronic Workshops in Computing (eWiC)*, 1st LEGE-WG International Workshop on Educational Models for Grid Based Services, Lausanne, Switzerland, September 2002.

[DER 01] De Roure D., Jennings N., Shadbolt, N. "Research Agenda for the Semantic Grid: A Future e-Science Infrastructure" In Report commissioned for *EPSRC/DTI Core e-Science Programme*. University of Southampton, UK, 2001.

[DIG 00] Dignum F., Greaves M., "Issues in Agent Communication: An introduction", Dignum F and Greaves M. (Eds.): *Agent Communication, LNAI 1916*, pp. 1-16, Springer-Verlag Berlin Heidelberg, 2000.

[FER 95] Ferber J., *Les Systemes Multi-Agents, vers une intelligence collective*, InterEditions, Paris, 1995.

[FRI 92] Friedman D.P., Jefferson S., "A Simple Reflective Interpreter", *IMSA'92, International Workshop on Reflection and Meta-Level Architecture*, Tokyo, 1992.

[GUE 99] Guessoum, Z. ,Briot, J.-P. "From Active Objects to Autonomous Agents". IEEE Concurrency, vol. 7-3, pp. 68-76, 1999.

[GUE 02] Guerin, F. "Specifying Agent Communication Languages", *Ph.D. Thesis*, Dept. of Electrical and Electronic Engineering, Imperial College, 2002.

[IST 01] Information Society Technologies Advisory Group (ISTAG), "Scenarios for Ambient Intelligence in 2010", Final Report Compiled by K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten & J-C. Burgelman, IPTS-Seville, February 2001.

[JON 03] Jonquet C., Cerri S. A., "Agents as Scheme Interpreters: Enabling Dynamic Specification by Communicating", submitted to RFIA'04, Toulouse, France, January 2004. Available on http://www.lirmm.fr/~jonquet.

[KIS 02] Kiselyov O., "XML, Xpath, XSLT implementations as SXML, SXPath, and SXSLT", *International Lisp Conference: ILC2002*, San Francisco, CA, Octobre 2002.

[MAR 01] Maraschi D., Cerri S. A., "The relations between Technologies for Human Learning and Agents", In procedings of the *AFIA 2001 Atelier: Methodologies et Environments pour les Systèmes Multi-Agents,* Grenoble: Leibniz-Imag, pp. 61-73, 2001.

[MCC 89] McCarthy J., "Elephant 2000: A Programming Language Based on Speech Acts". *Unpublished draft* Stanford University, www.formal.stanford.edu/jmc/elephant.pdf, 1989.

[NOR 91] Normak K., "Simulation of Object-Oriented and Mechanisms in Scheme", *Institute of Electronic Systems*, Aalborg University, Denmark, 1991.

[QUE 94] Queinnec C., *Les langages LISP*, Interéditions, Paris, 1994.

[QUE 00] Queinnec C., "The Influence of Browsers on Evaluators or, Continuations to Program Web Servers", *ICFP'00*, Montréal, Canada, 2000.

[SEA 71] Searle J., *Les actes de langages, essai de philosophie du langage*, Herman Editeur, Paris 1971.

## 10. Annexe

The presented model has been implemented in a prototype  developed with MIT Scheme 7.7.1, standard R5RS. You will find in http://www.lirmm.fr/~jonquet the Scheme files specifying the meta-evaluator used for experimentations, the `kqmlmsg` message interpreter module, the file implementing our Agents, and finally the first experimentation file.

## 11. Acknowledgements