

Agents as Scheme Interpreters: Enabling Dynamic Specification by Communicating

Clement Jonquet - Stefano A. Cerri

LIRMM - University Montpellier II
161, rue Ada
34392 Montpellier Cedex 5 - France

{cerri, jonquet}@lirmm.fr

Abstract

We proposed in previous papers 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 therefore representing evolving meta-level Agent's knowledge. When the Agent's interpreter is a nondeterministic one, the dialogues may consist of subsequent refinements of specifications in the form of constraint sets. The paper presents a worked out example of dynamic service generation – such as necessary on Grids – by exploiting STROBE Agents equipped with a nondeterministic interpreter. It shows how enabling dynamic specification of a problem. Then it illustrates how these principles could be effective for other applications. Details of the implementation are not provided here, but are available.

Keywords

Agent communication, message dynamic interpretation, reflexivity, reifying procedures, STROBE model, ACL, dynamic specification, constraints and dialogue.

Résumé

Nous avons proposé dans de précédents papiers une extension et une implémentation du modèle STROBE, qui considère les Agents comme des interpréteurs Scheme. Ces Agents sont capables d'interpréter des messages dans des environnements donnés incluant un interpréteur qui apprend de la conversation et donc qui représente l'évolution de sa connaissance au niveau méta. Quand ces interpréteurs sont non déterministes, le dialogue consiste à raffiner les spécifications d'un problème par des ensembles de contraintes. Ce papier présente un exemple de génération dynamique de service – tels qu'ils sont nécessaires dans le Grid – exploitant des Agents STROBE équipés d'un interpréteur non déterministe. Il montre comment réaliser la spécification dynamique d'un problème. Puis il illustre comment ces principes peuvent

être intéressants pour d'autres applications. Les détails de l'implémentation ne sont pas fournis ici mais sont disponibles.

Mots Clef

Communication Agent, interprétation dynamique de message, réflexivité, procédures réifiées, STROBE, spécification dynamique, contraintes et dialogue.

1. Introduction

Knowledge communication is 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's societies is a hard research subject still in its infancy. We try to identify, here, a research agenda by proposing a model of knowledge learning, based on 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 STROBE [4] [5], 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. We will show how communication enables to dynamically change “values” in an environment and especially how these interpreters, which represent the Agents, can dynamically adapt their way of interpreting messages. Specially as, we consider these interpreters as

representing Agent's meta-level knowledge, we may deduce that our method enables for dynamically modifying this knowledge by changing any part of the Agent's interpreter. 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 a "teacher-student" dialogue experimentation. We think that considering Agents as interpreters is an interesting point of view, and we show it is effective for domains such as the Web, Grid Computing and e-commerce dialogues (considering Agents as nondeterministic evaluators). The purpose of the paper is to show that on the Web, the classical software engineering approach of specifying and subsequently coding may be challenged by one that intertwines specification and execution: **the dialogue**, that we show, allows carrying out these two steps at the same time (dynamic specification).

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 Agent communication in the future; Sections 3 and 4 present our model which regards Agents as Scheme interpreters and provides them a set of pairs (environment interpreter). Section 5 briefly reminds our method to make Agent interpreters evolve dynamically. A "toy example" of performative learning is also presented. Section 6 illustrates a scenario of e-commerce dialogue and the dynamic specification of a problem, after presenting a nondeterministic evaluator. Finally, in section 7, 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 [9]. 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 Agents on the Web.** 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. 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 and a language that enables to 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 adapt during communication** [4]. §5 will briefly explain how reflection can help us to dynamically modify and, by consequence, adapt our Agents. We have also to consider the fact that Agents can interact with each other or with humans following the same principles [5] [18]. **What is important is the Agent's representation of its interlocutors.** §4 will show how the Cognitive Environment concept from STROBE [4] may help us. 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 others 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 [8]. These ACLs are based on the speech act theory¹. 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.

In reaction to all these languages, many models of communication have been proposed. An alternative to the way of considering Agents is presented in [17]. Among other things, STROBE [4] is interested in significant principles for a communication being based on the three simple Scheme primitives: *STream*, *Object*, and, especially interesting to us, *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.

¹ This model comes from language philosophy [1][22].

Now let us imagine an “ideal” scenario, such as those outlined in [12], of what could be a MAS 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 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 analyze 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 [6][7], an Agent can also transmit to another the interpreter allowing 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 interpreters, can modify them dynamically to evolve progressively during the conversations.

3. Agents as Scheme interpreters

The proposed model considered Agents as interpreters for both messages and their content. This idea comes from STROBE that considers Agents as REPL interpreters (Read, Eval, Print, Listen)² While it communicates, each Agent executes a REPL loop that is overlapping with the others ones (cf. Figure 2). **This principle is very important because it regards Agents as autonomous entities whose interactions are functionally controlled by a 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

² 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.

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 view, 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 (by abstracting on the environment [4]) and on the control model (by abstracting on continuations [21]). 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's 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 his 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 Environments [3]. In reality, it allows an Agent to take into account or not (according to specific criteria) 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 a given environment and with a given interpreter both dedicated to the conversation. 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 their representations

Their global environment is private and does not change. It is cloned³ 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.

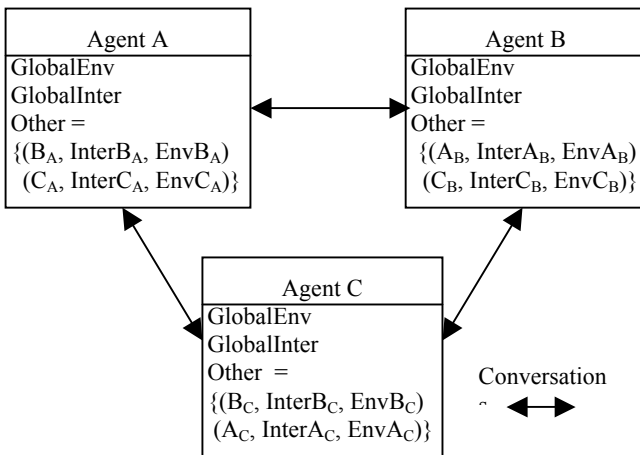


Figure 1. Agent attributes for representing others

Now we have both our Agent's 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 Agent's interpreter(s)⁴.

³ Not necessarily if we consider an Agent that would take back a conversation in another context, already existing. For example, if an Agent wants to continue a specific conversation interrupt above for any reasons.

⁴ Note that in fact, our model considers Agents, not as a single interpreter, but as a set of interpreters.

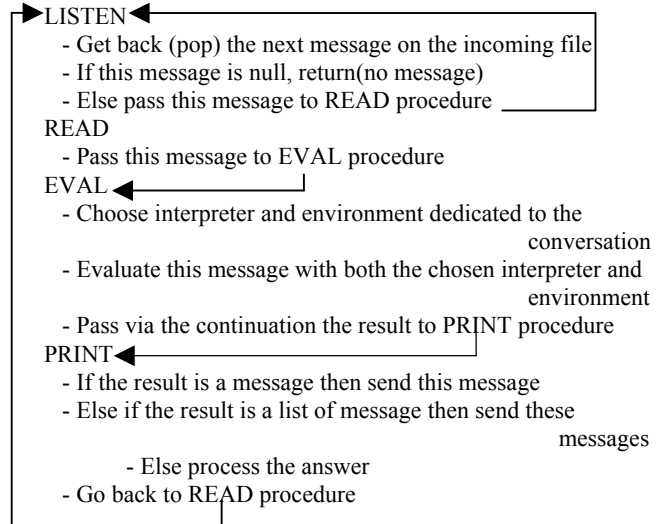


Figure 2. The REPL loop of our Agents

5. Dynamic modification of an interpreter: a “teacher-student” dialogue experimentation

5.1. Reflection and reifying procedure

The meta-evaluator we used comes from the famous article written by Jefferson and Friedman, “A Simple Reflective Interpreter” [10]. 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 allows a user's program to access its execution context⁵** 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-procedure⁶, corresponding to the two steps of the environment evaluation [2]. Storing our Agents' interpreters in environment, we can use the reifying procedure mechanism to modify them. For more details about this method of dynamic interpreter modification we invite the reader to refer to other submitted papers [13][14] and [15].

5.2. Communication protocol

The messages we consider are inspired by the KQML or FIPA ACL 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 process them by evaluating their contents:

⁵ The execution context is made of: expression to evaluate, the environment of evaluation of this expression and the continuation to give to this evaluation.

⁶ Corresponding to eval and apply of [2] and evaluate and invoke of [20].

evaluate-kqmlmsg. Used performatives are: *assertion*, *request*, *order*, *ack*, *answer*, *executed* (cf. table 2 [4]) 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 its 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 its current interlocutors. There is no answer defined for the broadcast messages.⁷

5.3. Example: a “teacher-student” dialogue

This “toy example” experimentation characterizes the technical aspects of our work and shows that this model is viable which means, it can really be implemented. 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 cited. 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 while communicating. The example 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 perform in the experimentation.

For the experimentation we developed some Agents able to communicate, i.e. exchanging messages together producing significant answers (following the defined

⁷ Actually, *broadcast* is a meta-performative.

protocol)⁸. 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 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.

After the last message process, *student* evaluate-kqmlmsg function is modified thus its messages interpreter. The corresponding function code in its environment dedicated to this conversation is changed. Then *student* Agent can process broadcast messages. For more details on teacher messages refer to [13][14] or on the model implementation to [15].

6. Enabling dynamic specifications by nondeterministic evaluation

Before showing how nondeterministic evaluation enables dynamic specifications, we need to introduce the notion of nondeterministic evaluator⁹.

6.1. What is a nondeterministic evaluator?

The key idea is that expressions in a nondeterministic language can have more than one possible value. With nondeterministic evaluation, an expression represents the exploration of a set of possible worlds, each determined by a set of choices. Actually, programs have different possible execution histories and the nondeterministic program evaluator will free the programmer from the details of how choices are made. Nondeterministic evaluation is based on the special form *amb*. The expression (*amb* *exp1* *exp2* ... *expn*) returns the value of one of the *n* expressions *exp_i*.

For example, the expression:

```
(list (amb 1 2 3) (amb 'a 'b))
```

can have six possible values:

```
(1 a) (1 b) (2 a) (2 b) (3 a) (3 b)10
```

⁸ In fact, our Agents are Scheme programmed object as we can see in the Normark article [19].

⁹ The following paragraph is inspired by the famous Abelson and Al. book [2] (chapter 4), and we invite the reader to see it for more details.

¹⁰ The expression (*amb*) without arguments corresponds to a failure.

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

Figure 3. Broadcast teaching “teacher-student” dialogue

The interest of such an evaluator is that functions can call `amb` special form by adding constraints (as predicates) on the values returned by `amb`. These constraints are expressed with the special form `require` defined like this:

```
(define (require p)
  (if (not p) (amb)))11
```

The evaluation of an `amb` expression can be seen as a tree solution exploration where a function is processed until a solution respecting all the constraints is found as long as the complete tree is not traversed. The form `(amb)`, without arguments, corresponds to a leaf of this tree, and then another branch must be explored. To recognize the special form `amb`, we have to modify the traditional evaluator to embody the change. Just like Prolog language, with which, one can call, one by one, for all solutions to a logical expression; with a nondeterministic evaluator, the `try-again` special form allows displaying of the next success evaluation to a function calling `amb`. The traditional interpretation loop is modified in

nondeterministic evaluation, to take into account these backtracks.

Example: Consider the `an-element-of` function which returns an element of a list. Its evaluation could be:

```
> (an-element-of '(a b c))
: b
> try-again
: a
> try-again
: c
> try-again
: no more values
```

We could just take a look at the body of the above function; here, the constraint is that the list can not be null. If it is `null?`, then `(amb)` is evaluated and the nondeterministic interpreter explores another branch of the solution tree:

```
(define (an-element-of items)
  (require (not (null? items))))
(amb (car items)
      (an-element-of (cdr items))))
```

Consider now a harder constraint problem, taken from [2]. It is a classic logic puzzle:

¹¹ The form `(if cond exp)` returns `exp` value if `cond` is true. Else it returns no value.

“Baker, Cooper, Fletcher, Miller, and Smith live on different floors of an apartment house that contains only five floors. Baker does not live on the top floor. Cooper does not live on the bottom floor. Fletcher does not live on either the top or the bottom floor. Miller lives on a higher floor than does Cooper. Smith does not live on a floor adjacent to Fletcher's. Fletcher does not live on a floor adjacent to Cooper's. Where does everyone live?”

We can construct the following procedure which determine who lives on each floor by enumerating all the possibilities and imposing the given restrictions via `require` forms:

```
(define (multiple-dwelling)
  (let ((baker (amb 1 2 3 4 5))
        (cooper (amb 1 2 3 4 5))
        (fletcher (amb 1 2 3 4 5))
        (miller (amb 1 2 3 4 5))
        (smith (amb 1 2 3 4 5)))
    (require (distinct? (list baker cooper
                              fletcher miller smith)))
    (require (not (= baker 5)))
    (require (not (= cooper 1)))
    (require (not (= fletcher 5)))
    (require (not (= fletcher 1)))
    (require (> miller cooper))
    (require
     (not (= (abs (- smith fletcher)) 1)))
    (require
     (not (= (abs (- fletcher cooper)) 1)))
    (list (list 'baker baker)
          (list 'cooper cooper)
          (list 'fletcher fletcher)
          (list 'miller miller)
          (list 'smith smith))))
```

Evaluating the expression `(multiple-dwelling)` produces the result:

```
((baker 3) (cooper 2) (fletcher 4) (miller
5) (smith 1))
```

So a nondeterministic evaluator could be useful for constraint programming and as we will see now could allow dynamic specification of a problem.

6.2. Enabling dynamic specification by communication in e-commerce scenario

Our model regards Agents as Scheme interpreters, thus it can regard them as nondeterministic interpreters i.e. able to recognize and process `amb`, `require`, and `try-again` special forms. Indeed, it is interesting for them because they could solve constraint programs just like those seen in the preceding section. But the most interesting point is that our Agents could progressively build such programs with a dialogue and then apply these programs to give answers or to achieve a task for another Agent. The constraints, defining a nondeterministic program, could be determined progressively with the conversation using

tools presented above to dynamically change functions bodies and the way of interpreting them.

Let us consider, for example, a standard e-commerce dialogue to look for a train ticket, like the one presented in [17]. A ticket is characterized by a departure city, a destination city, a price, and a date. A *SNCF* Agent receives a *customer* Agent request for ticket proposals. The dialogue in real situation could be:

- a. Customer: *I want a ticket from Montpellier to Paris*
- b. SNCF: *When?*
- c. Customer: *Tomorrow before 10AM. Please, give me a proposition for a ticket!*
- d. SNCF: *Ok, train 34, departure tomorrow 9.30AM, from Montpellier to Paris, 150€*
- e. Customer: *Is it possible to pay less than 100€ ?*
- f. SNCF: *Ok, train 31, departure tomorrow 8.40AM, from Montpellier to Paris, 95€*
- g. Customer: *Another proposition please?*
- h. SNCF: *Ok, train 32, departure tomorrow 9.15AM, from Montpellier to Paris, 98€*
- i. Customer: *Ok, I accept this one...*

We can note that the interactions **a**, **b**, **c** and **e** deals with the constraints on ticket selection. The interactions **d**, **f** and **h** are function applications of ticket research, with various constraints. The interaction **g** corresponds to a request of the *customer* to get another answer that means to explore another branch of the solution tree. Figure 4 illustrates how this dialogue can be represented into Scheme expressions in order to be realized by our Agents. The *customer* Agent transmits its requests with `require` and `try-again` special forms. *SNCF* Agent starts, at the beginning of the conversation, a new `find-ticket` function construction which is dynamically modified and built progressively during the conversation. These modifications consist in changing a value in the *SNCF* Agent environment representing *customer* Agent.

This idea is interesting because it is the dialogue that builds the computation to be carried out and not the opposite. It is a typical scenario that we could find in many e-commerce applications or others of the same kind where Agents must build a program to find a solution together. The classical approach to program construction (which can be found in traditional software engineering) that specifies the problem before coding it, is changed into a dynamic specification approach during coding. That is to say, specification and realisation are, with our model, carried out at the same time (in one step).

This scenario fits those envisaged by future Grids [7], where the focus is to generate services dynamically, not just serve clients with predefined static services.

CUSTOMER	SNCF
<p><i>I want a ticket from Montpellier to Paris</i></p> <pre>(require (eq? depart montpellier)) (require (eq? dest paris))</pre>	<p>Definition of a new <code>find-ticket</code> procedure :</p> <pre>(define (find-ticket) (let ((depart (amb *ens-ville*)) (dest (amb *ens-ville*)) (prix (amb *ens-prix*) (date (amb *ens-date*))) (require (not (eq? depart dest))) (require (eq? depart montpellier)) (require (eq? dest paris)) (list (list 'depart depart) (list 'destination dest) (list 'prix prix) (list 'date date))))</pre> <p><i>When ?</i></p>
<p><i>Tomorrow before 10AM</i></p> <pre>(require (< date *demain10H*))</pre> <p><i>Please, give me a proposition for a ticket!</i></p> <pre>(find-ticket)</pre>	<p><code>find-ticket</code> procedure modification adding a new constraint. Then procedure execution:</p> <p><i>Ok, train 35, departure tomorrow 9.30AM, from Montpellier to Paris, 150€</i></p> <pre>((depart montpellier) (destination paris) (prix 150) (date *dem9H30*))</pre>
<p><i>Is it possible to pay less than 100€ ?</i></p> <pre>(require (< prix 100)) (find-ticket)</pre>	<p>idem</p> <p><i>Ok, train 31, departure tomorrow 8.40AM, from Montpellier to Paris, 95€</i></p> <pre>((depart montpellier) (destination paris) (prix 95) (date *dem8H41*))</pre>
<p><i>Another proposition please ?</i></p> <pre>(try-again)</pre>	<p><code>find-ticket</code> procedure execution:</p> <p><i>Ok, train 32, departure tomorrow 9.15AM, from Montpellier to Paris, 98€</i></p> <pre>((depart montpellier) (destination paris) (prix 98) (date *dem9H15*))</pre>
<p><i>Ok, I accept this one...</i></p>	

Figure 4. Dialogue between SNCF Agent and customer Agent to find a train ticket

7. Others interests and extensions

The §5 example shows how to add a new performative to the ones already known by an Agent, thus how to modify an Agent's messages interpretation function. However, the same principles can be used to modify any part of an Agent's interpreter. For instance, we could have made an example which adds `let*` or another special form to the 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 interpreters¹² just like in Grid architectures where is more interesting to move program 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 [11] in order to choose, in a specific context the most adequate ACL. The ontology becomes intrinsic to the communication language.

¹² It comes from the idea quoted from [2]: "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"

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 tree. In the same idea, many languages linking S-expression formalism (Scheme) and XML appear [16].

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 principle [2], 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 ask all its current correspondents to process a little Fibonacci number and after receiving all the answers, compare answer times (or compute some statistics) 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 *contact net*, or for Grid computing where “effective” Agents have to be selected to perform heavy computation.

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). We illustrated this idea by a couple of “toy example” experimentations. In the second one we show how our model fits with an e-commerce scenario enabling the dynamic specification of constraints by inter-Agents conversations.

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 transfer 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 an architecture, simple, functional 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.

Annex

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.

Acknowledgements

This work was performed to fulfil in part the requirements for a DEA Informatique (M.Sc. in Computer Science) by one of the authors (CJ). The support of the EU project LEGE-WG (Learning Grid Excellence Working Group) is gratefully acknowledged.

References

- [1] Austin J.L., *Quand dire c'est faire*, Edition du seuil, Paris, 1970.
- [2] Abelson H., Sussman G.J., Sussman J., *Structure and Interpretation of Computer Programs*, Second Edition, MIT Press, Cambridge, Massachusetts, 1996.
- [3] Cerri S.A., “Cognitive Environments in the STROBE Model”. Presented at EuroAIED: *the European Conference in Artificial Intelligence and Education*, Lisbon, Portugal (1996)
- [4] Cerri S.A., “Shifting 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.
- [5] 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.

- [6] 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, 16 September 2002.
- [7] 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.
- [8] Dignum F., Greaves M., “Issues in Agent Communication: An introduction”, Dignum F and Greaves M.(Eds.): *Agent Communication, LNAI 1916*, pp1-16, Springer-Verlag Berlin Heidelberg, 2000.
- [9] Ferber J., *Les Systemes Multi-Agents, vers une intelligence collective*, InterEditions, Paris, 1995.
- [10] Friedman D.P., Jefferson S., “A Simple Reflective Interpreter”, *IMSA’92, International Workshop on Reflection and Meta-Level Architecture*, Tokyo, 1992.
- [11] Guerin F. “Specifying Agent Communication Languages”, *Ph.D. Thesis*, Dept. of Electrical and Electronic Engineering, Imperial College, 2002.
- [12] 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.
- [13] Jonquet C., Cerri S.A., “Apprentissage issu de la communication pour des Agents cognitifs”, Submitted to JFSMA’03: Journées Francophones sur les Systèmes Multi-Agents, Mai 2003. Available on <http://www.lirmm.fr/~jonquet>.
- [14] Jonquet C., Cerri S.A., “Cognitive Agents Learning by Communicating”, submitted to Colloque ALCAA: Agents Logiciels, Coopération Apprentissage, Activité humaine, Mai 2003. Available on <http://www.lirmm.fr/~jonquet>.
- [15] Jonquet C., Mémoire de DEA, LIRMM – Université Montpellier II, France, Juin 2003. Available on <http://www.lirmm.fr/~jonquet>.
- [16] Kiselyov O., “XML, Xpath, XSLT implementations as SXML, SXPath, and SXSLT”, *International Lisp Conference: ILC2002*, San Francisco, CA, Octobre 2002.
- [17] Maraschi D., Cerri S. A., “The relations between Technologies for Human Learning and Agents”, In proceedings of the *AFIA 2001 Atelier: Methodologies et Environnements pour les Systèmes Multi-Agents*, Grenoble: Leibniz-Imag, pp. 61-73, 2001.
- [18] McCarthy J., “Elephant 2000: A Programming Language Based on Speech Acts”. *Unpublished draft* Stanford University, <http://www-formal.stanford.edu/jmc/elephant.pdf>, 1989.
- [19] Normak K., “Simulation of Object-Oriented and Mechanisms in Scheme”, *Institute of Electronic Systems*, Aalborg University, Denmark, 1991.
- [20] Queinnec C., *Les langages LISP*, Interéditions, Paris, 1996.
- [21] Queinnec C., “The Influence of Browsers on Evaluators or, Continuations to Program Web Servers”, *ICFP’00*, Montréal, Canada, 2000.
- [22] Searle J., *Les actes de langages, essai de philosophie du langage*, Herman Editeur, Paris 1971.