I-dialogue: Modeling Agent Conversation by Streams and Lazy Evaluation

Clement Jonquet and Stefano A. Cerri
LIRMM (Laboratory of Informatics, Robotics, and Microelectronics of Montpellier) & CNRS
University of Montpellier II
161 rue Ada 34392 Montpellier Cedex 5 - France
{jonquet,cerri}@lirmm.fr

ABSTRACT

This paper defines and exemplifies a new computational abstraction called *i-dialogue* which aims to model communicative situations such as those where an agent conducts multiple concurrent conversations with other agents. The i-dialogue abstraction is inspired both by the *dialogue* abstraction proposed by [17] and by the STROBE model [4]. I-dialogue models conversations among processes by means of fundamental constructs of applicative/functional languages. (i.e. streams, lazy evaluation and higher order functions). The i-dialogue abstraction is adequate for representing multi-agent concurrent asynchronous communication such as it can occur in service providing scenarios on today's Web or Grid. A Scheme implementation of the i-dialogue abstraction has been developed and is available.

1. INTRODUCTION

Current computing systems are more and more distributed. The emergence of the Web and tomorrow's Grid is a proof of it. As they are by definition dispersed, these systems often consist of interacting entities. For that reason interaction modeling is a key issue in distributed system development. In the Distributed Artificial Intelligence (DAI) and Multi-Agent Systems (MAS) communities, these interactive entities, considered autonomous and intelligent, are called Agents. In this paper we describe *i-dialogue*, an abstraction of the interaction between several processes inspired from O'Donnell's *dialogue* [17]. The dialogue and i-dialogue combinators are higher order functions that structure 2-agent's and n-agent's conversations as networks of communicating processes. These abstractions deal directly with different sequences of inputs, different sequence of outputs, and state of the agents concerned. Theses abstractions are useful to describe interaction between several agents, whatever the type of the agents (i.e. Artificial Agents (AA) or Human Agents (HA)). The kernel of the model consists on the assumption that messages are elements of an input and output "stream". More specifically, [17] shows how programming environments can be defined using dialogues. His popular 1985 paper gives an application, called RIE (Recursive Interactive Environment), in which the traditional REP (Read Eval Print) interaction loop of applicative languages is implemented by the dialogue abstraction. The aim of O'Donnell's paper was to help programmers to develop useful and smart programming environments. The aim of this paper is to help to model agent's conversations with a simple, powerful and elegant abstraction: i-dialogue.

Part of the paper is used to present a general implementation of the i-dialogue abstraction for applicative languages such as Daisy or LISP/Scheme. This implementation is based on simple features of theses languages: higher order functions (specifically function as argument), streams (as sequences that are infinitely long) and lazy evaluation (to support streams). We show how streams are a smart data structure to model agent conversation because they allow defining recursive data structures that can be used to represent sequences that are infinitely long such as the inputs and outputs of the dialogue and i-dialogue abstractions. The paper also argues that this new abstraction fits with the STROBE model [4, 15], an agent representation and communication model (highly inspired by applicative/functional research) which suggest among others, to model agent conversations by streams.

The rest of the paper is organized as follows: Section 2 introduces the concept of agent and explains the importance of interaction in MAS. Section 3 determines the notations and conventions used in this paper. Section 4 recalls the principles of O'Donnell's dialogue abstraction and explains the concepts that are going to be extended further in the paper. This part of the paper is highly inspired from O'Donnell's paper. Before presenting the concepts of a general implementation of dialogue and i–dialogue, this section makes a little state of the art of the notions of streams and lazy evaluation. Then, the extension of dialogue in section 5 presents the i-dialogue abstraction by generalizing the three agents case. Section 6 looks on related work, and especially establishes a relation between i-dialogue and the STROBE model. It also shows how the i-dialogue abstraction suits for service and gives an example. Section 7 concludes the paper and gives some perspectives for further work with i-dialogue.

2. AGENTS AND MAS

As [7] defines it, an agent (i.e. AA) is a physical or virtual autonomous entity which is capable of acting in an environment[1], which can communicate directly with other agents, which possesses its own state and skills and can offer services, and whose behavior tends towards satisfying its objectives. We assume first that AA are just software programs and their associated processes that at least are autonomous, distributed and able to communicate asynchronously with their environment and others agents by means of a communication language that is independent from the content of the communication and from the internals of agents[2].

Traditionally, agent conversations are modeled by communication protocols or conversation policies [11, 6]. These protocols represent the interaction structure and specify rules that must be respected during the conversation. Using protocols, an agent interprets messages from a conversation one-by-one, changing at each step its own state, and following the protocol to produce the next message in the conversation. The main advantage of this approach is the semantic description of the conversation (via logical expression of preconditions, postconditions and mental states (Believe, Desire, Intentions)) [10]. But, this approach has weaknesses, especially interoperability, composition and verification of protocols. Agents are forced to follow a policy restricting their autonomy and the dynamic interpretation of messages. For example, ACLs and communication protocols presuppose agent sincerity and thus assume that an agent know some information about the internal state of its partners. Moreover, the only way for an agent to consider the entire conversation is to look at the protocol, which was previously determined (before the conversation) and which cannot change dynamically. Therefore, agents are obliged to fit fixed conversations while it should be conversations which fit dynamically changing agents. By contrast, other approaches, for example [18], try to explain that the next answer message of a conversation can not be foreseen, and should be determined only after the interpretation of the previous incoming message. Instead of modeling mental states and their dynamics in conversation, these approaches deal directly with speech acts rather than on hidden intentions. The i-dialogue abstraction is inscribed in this second idea by providing a simple, yet powerful abstraction to model message exchanges in agent communication.

Simply grouping together several agents is not enough to form a MAS. It is interaction between these agents, with the environment, and with users that makes it. Interaction allows cooperation and coordination between agents [7]. However, interaction is strongly related to agent's autonomy. In order to enhance agents' autonomy, it is interesting to develop communication models: i) which expect agents to communicate without being knowledgeable of the internal believes of their partners, with whom they only handle output messages as interface [19]; ii) which allow agents to handle the entire conversation dynamically and not simply interpret messages one-by-one following a previously determined structure. The i-dialogue abstraction respects these ideas dealing only with some input sequences of messages (i.e. the partners output sequences of messages).

[1]The term environment is here used with its MAS meaning, that is to say, the world surrounding an agent and in which it progresses.

[2]These languages are called Agent Communication Languages, ACLs, such as KQML or FIPA-ACL. They are speech act oriented. See for example [11].

3. NOTATIONS

In order to express theses abstractions, we use a Daisy inspired language syntax to ensure continuity with [17]. This syntax is briefly repeated in appendix A. The rest of the notations used are:

- X, Y (uppercase letters) represent agents
- x, y (lowercase letters) represent elements of sequences (i.e. messages)
- x_{Yi} is the i^{th} message from X to Y
- ξ, ψ (Greek letters) represent states of agents X, Y.
- f^X_Y is X's transition function dedicated to Y
- R_X is X's function producing a result value from a state
- I^X_Y is X's input sequence of messages from Y
- O^X_Y is X's output sequence of messages to Y
- then $O^X_Y = (x_{Y1}, x_{Y2}, \ldots, x_{Yn}) = I^Y_X$
- and $O^Y_X = (y_{X1}, y_{X2}, \ldots, y_{Xn}) = I^X_Y$

4. THE DIALOGUE ABSTRACTION

4.1 Description of dialogue

A dialogue is an interactive session between two agents, A and B, which take turns sending messages to each other as figure 1 shows.

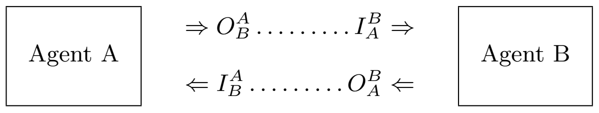
Agent A
$\Rightarrow O^A_B \ldots\ldots\ldots I^B_A \Rightarrow$
$\Leftarrow I^A_B \ldots\ldots\ldots O^B_A \Leftarrow$
Agent B

Figure 1: Streams in dialogue between two agents

Each agent has a state that contains personal information (internals) and information about the history of the conversation. One of the agents (A) begins the dialogue by sending the first message to the other (B). Initially A is in state α_j, $j \geq 0$[3]. Each agent model the conversation with the partner with the dialogue abstraction. During the conversation, each agent computes a new state and a new output from its previous state and the last input it received from the other agent, using its transition function (f^A_B for agent A and f^B_A for B):

$$f^A_B : \left[\alpha_{j+k}\ I^A_B\right] \rightarrow \left[\alpha_{j+k+1}\ O^A_B\right]$$

$$f^B_A : \left[\beta_k\ I^B_A\right] \rightarrow \left[\beta_{k+1}\ O^B_A\right]$$

[3]If agent A has just been created then $j = 0$, but if A has already had a dialogue with some other agent then $j \geq 0$. As agent B does not start the dialogue, we consider it as "new", thus its beginning state is β_0.

Thus the transition functions f_B^A and f_A^B and the initial state α_j and β_0 define, for each agent, a sequence of states and a sequence of outputs. The 4-tuple of sequences:

$$\text{A}: \quad (\alpha_j \ \alpha_{j+1} \ \ldots) \quad O_B^A = (a_{B1} \ a_{B2} \ \ldots a_{Bn} \ \ldots) = I_A^B$$

$$\text{B}: \quad (\beta_0 \ \beta_1 \ \ldots) \quad O_A^B = (b_{A1} \ b_{A2} \ \ldots b_{An} \ \ldots) = I_B^A$$

specifies a complete history of the dialogue (figure 2). Note furthermore that every agent has a result function (R_A and R_B) which it is used to produce a final "dialogue value" according to their last state, if the agent decides to terminate the dialogue.

A dialogue between A and B, initiated by A, is defined to be a 4-tuple of sequences. Advantages of this representation are quoted from [17]:

> The advantage of this definition is that it doesn't rely on an explicit notion of time, I/O or side effects. The ordering of sequence elements corresponds to their relative order in time. Similarly, the definition of each sequence element as the value of a function applied to other sequence elements (except for the initially defined elements) replaces the notion of sending messages with explicit I/O. Finally, it is not necessary to view the state of a participant as a variable which must be changed through a side effect upon each communication. Instead of destroying old state values, we view state as a sequence of values.

Agent A	Agent B
$(initial)\alpha_j$	$\beta_0(initial)$
(send to B)$a_{Bj} \Rightarrow$	
	$[\beta_1, b_{A0}] = f_A^B : [\beta_0, a_{Bj}]$
	$\Leftarrow b_{A0}$(send to A)
$[\alpha_{j+1}, a_{Bj+1}] = f_B^A : [\alpha_j, b_{A0}]$	
(send to B)$a_{Bj+1} \Rightarrow$	
$\vdots$	
$[\alpha_{j+k}, a_{Bj+k}] =$ $f_B^A : [\alpha_{j+k-1}, b_{Ak-1}]$	
(send to B)$a_{Bj+k} \Rightarrow$	
	$[\beta_{k+1}, b_{Ak}] =$ $f_A^B : [\beta_k, a_{Bj+k}]$
	$\Leftarrow b_{Ak}$(send to A)

Figure 2: Dialogue between two agents A and B

4.2 History of streams and lazy evaluation

Until now we used the term "sequence" to define inputs and outputs. This paragraph gives some details about the type of sequence the dialogue (and i-dialogue) abstraction needs. The dialogue implementation must be able to operate on sequences that represent state and communications in which the elements are not evaluated before they are really needed. Those sequences are potentially infinite lists called streams.

As [1] explains, streams allows to model systems and changes on them in terms of sequences that represent the time histories of the systems being modeled. They are an alternative approach to modeling state. As a data abstraction, streams are sequences as lists are. The difference is the time at which the elements of the sequence are evaluated: list elements are evaluated when the list is constructed where as stream elements are evaluated only when they are accessed. Streams are a clever idea that allows to use sequence manipulations without incurring the costs of manipulating sequences as lists. The basic idea consists in constructing a stream only partially, and to pass the partial construction to the program that consumes the stream. Stream processing allows to model stateful systems without ever using assignment or mutable data. It is often easier to consider the sequence of values taken on by a variable in a program as structure that can be manipulated, rather than considering the mechanisms that use, test, and change the variable. This has important implications, both theoretical and practical, because we can build models that avoid the drawbacks inherent in introducing assignment (side effects) . The stream formulation is particularly elegant and convenient because the entire sequence of states is available as a data structure that can be manipulated with a uniform set of operations [1]. For Burge [2], a stream is a functional analog of a coroutine and may be considered to be a particular method of representing a list in which the creation of each list element is delayed until it is actually needed. [2] discusses the use of streams as a method for structured programming and introduce a set of functional stream primitives for this purpose. In [8] mechanism for the maintenance of streams based on continuations is presented. Stream processing is also presented in [13] where streams are represented by pair of functions (enumerator and selector).

For the applicative/functional language community a stream could be seen as a list which is built using a non-strict constructor. Peter J. Landin first originated the idea of a non-strict data structure. Most applicative/functional languages as Scheme [1] or LISP are applicative order languages, that means that all the arguments of functions are evaluated when the function is applied. In contrast, normal order languages (such as Daisy [14]) delay evaluation of function arguments until the actual argument values are needed. Delaying evaluation of function arguments until the last possible moment (e.g. until they are required by a primitive, or printed as an answer) is called lazy evaluation. Lazy evaluation, which is the key to making streams practical, comes from Algol 60, and was used to implement streams firstly in [16]. Lazy evaluation for streams was introduced into LISP by [9], which shows that with this mechanism, streams and lists can be identical. A lazy evaluator only computes values when they are really required avoiding computing values that are not really needed. For example, lazy languages allows to define the `if` special form as a classical function, allowing it to benefit of the first class proprieties of functions (only in Scheme).[4]

[4]In the literature about lazy evaluation the reader could find the three "calls-by" terms about the order of evaluation. *Call-by-value* argument passing, as it is the case in applicative language, where a function application is done with the values of each argument. *Call-by-name* argument passing, as it is the case in lazy language, where function application is done with the names of each argument. *Call-by-need* argument passing, which correspond to the call-by-name principle combined with memoization (values are memorized to be compute only once).

To complete this small overview of streams, we should make a little difference between Abelson and Sussman's stream definition and Burge's one. The first one defines streams as a pair which head part (car) is evaluated and tail part (cdr) is delayed. The second one defines them as list in which the creation of each list element (even the head) is delayed. The second definition is called "lazy list". It permits to create delayed versions of more general kinds of list structures, not just sequences, but for examples trees [12].

Features of applicative/functional programming seem particularly adapted for modeling agent conversation. For example, lazy evaluation is relevant to express the natural retarded aspect of interactions: an agent may delay the production of the next message until it interprets its partner's reaction to its first message. As streams allow to define recursive data structures, we can use them to represent sequences that are infinitely long such as the input and output of the dialogue and i-dialogue abstractions. Streams used in dialogue an especially in i-dialogue are even lazy lists as it is explain latter.

4.3 Implementation of the dialogue function

The dialogue abstraction provides a useful way to think about communications and state. It can be implemented efficiently by a simple recursive function with the techniques of applicative/functional programming. This function has vocation to be run by each agent implied in a 2-agent conversation. Figure 3 shows an implementation of dialogue. The dialogue function has four parameters:

1. *inputs.* It is a stream of input messages (I_B^A for agent A, I_A^B for agent B).
2. *initial-state.* It is the initial state of the agent running a dialogue function (α_j for agent A, β_0 for agent B).
3. *step-fcn.* It is the transition function that defines the actions of the agent (f_B^A for agent A, f_A^B for agent B). The *step-fcn* must return a stream of four elements:
   (a) A stream of unused inputs, which is usually the tail of *inputs*. That value will be used in the next step of the dialogue unless the dialogue terminates.
   (b) A stream of outputs messages sent to the partner.
   (c) A new state.
   (d) A boolean value that is true if the agent wishes to terminate the dialogue and false otherwise.

   The dialogue function repeatedly applies *step-fcn* to the current values of *inputs* and *state* in order to find the new *outputs'* and *state'*.
4. *result-fcn.* It is a function which the agent uses to produce a "dialogue value" with its last state (R_A and R_B).

The dialogue function returns a stream[5] of three elements:

1. *unused-inputs.* The agent removes the input elements that it needs from *inputs*, and returns the remainder *unused-inputs.*
2. *outputs.* It is a stream of output messages (O_B^A and O_A^B).
3. *result.* A value which is computed by the agent by applying its *result-fcn* to its final state.

```
dialogue : ⟨inputs initial-state step-fcn result-fcn⟩ ≡
  letrec
    run ≡ λ ⟨inputs state⟩ .
      let
        (inputs' outputs' state' done') ≡
          stef-fcn : ⟨inputs state⟩
      in
        if done'
          then (inputs' outputs' result-fcn:state')
          else
            let
              (inputs" future-outputs" result") ≡
                run : ⟨inputs' state'⟩
            in
              (inputs"
              append-ll : ⟨outputs' future-outputs"⟩
              result")
  in
    run : ⟨inputs initial-state⟩
```

Figure 3: Definition of the function *dialogue*

Then, in a case such as the one illustrated by figure 1, calls to dialogue are, and produce:

Agent A: *dialogue* : $\langle I_B^A\ \alpha_j\ f_B^A\ R_A\rangle \rightarrow (I_B^A\ O_B^A\ val)$

Agent B: *dialogue* : $\langle I_A^B\ \beta_0\ f_A^B\ R_B\rangle \rightarrow (I_A^B\ O_A^B\ val)$

4.4 The dialogue abstraction limits

The dialogue abstraction was adequate for the interaction of two processes but today's distributed system requirements oblige to consider interaction between more than two agents. Therefore, the limit of the dialogue abstraction is intrinsic to it: it does not model more than two agents conversation. Indeed, several uses of dialogue (executed serially or in parallel) do not model conversation among several agents but several different dialogues that an agent has with each partner at the same time. For example, two dialogues executed serially do not model a four agents conversation as the first one must terminate before the second one starts. In the same way, two dialogues executed in parallel do not model a four agents conversation as the different inputs and outputs are not intertwined. The processing of inputs of the first dialogue always produces outputs of the first dialogue and same thing respectively for the second dialogue. We need to extend the dialogue abstraction to model conversations where processing of one agent inputs possibly produces not the outputs for this agent but another outputs intended to another agent. It is the scope of this paper. The i-dialogue

[5]This sequence must be a stream because the *unused-inputs*, which can include future inputs from a future dialogue, has not to be evaluated since the messages inside have not been produced yet.

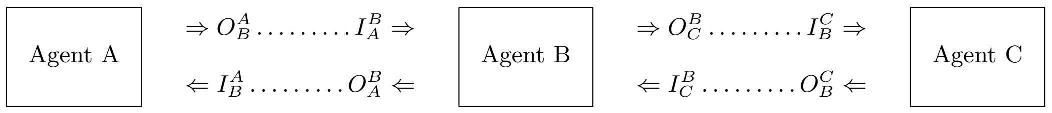
Agent A
$\Rightarrow O^A_B \ldots\ldots\ldots I^B_A \Rightarrow$
$\Leftarrow I^A_B \ldots\ldots\ldots O^B_A \Leftarrow$
Agent B
$\Rightarrow O^B_C \ldots\ldots\ldots I^C_B \Rightarrow$
$\Leftarrow I^B_C \ldots\ldots\ldots O^C_B \Leftarrow$
Agent C

Figure 4: Streams in trialogue between agents A,B and C

$(initial)\alpha_j$ $\quad$ $\beta_0(initial)$ $\quad$ $\gamma_0(initial)$

(send to B)$a_{Bj} \Rightarrow$

$[\beta_1, b_{C0}] = f^B_A : [\beta_0, a_{Bj}]$
(send to C)$b_{C0} \Rightarrow$

$[\gamma_1, c_{B0}] = f^C_B : [\gamma_0, b_{C0}]$
$\Leftarrow c_{B0}$(send to B)

$[\beta_2, b_{A0}] = f^B_C : [\beta_1, c_{B0}]$
$\Leftarrow b_{A0}$(send to A)

$[\alpha_{j+1}, a_{Bj+1}] = f^A_B : [\alpha_j, b_{A0}]$
(send to B)$a_{Bj+1} \Rightarrow$

$\vdots$ $\quad$ $\vdots$

$[\alpha_{j+k}, a_{Bj+k}] = f^A_B : [\alpha_{j+k-1}, b_{Ak-1}]$
(send to B)$a_{Bj+k} \Rightarrow$

$[\beta_{2k+1}, b_{Ck}] = f^B_A : [\beta_{2k}, a_{Bj+k}]$
(send to C)$b_{Ck} \Rightarrow$

$[\gamma_{k+1}, c_{Bk}] = f^C_B : [\gamma_k, b_{Ck}]$
$\Leftarrow c_{Bk}$(send to B)

$[\beta_{2k+2}, b_{Ak}] = f^B_C : [\beta_{2k+1}, c_{Bk}]$
$\Leftarrow b_{Ak}$(send to A)

Figure 5: Trialogue between three agents A, B and C

abstraction aims to realize what was anticipated by the end of section 2 of O'Donnell's paper. Instead of directly connect A's output stream to C's input stream, and vice versa, i-dialogue allows B to process A's output stream to produce a different stream for C's inputs.

5. THE I-DIALOGUE ABSTRACTION

The purpose of this section is to extend the dialogue abstraction from two agents to n agents. In a first time, for the reader comprehension, we propose the *trialogue* abstraction which deals with three agents. Then, we generalize to n agents.

5.1 The trialogue abstraction

Trialogue aims to model an interactive session between three agents, A, B and C, where A and C send messages to B and B to A and C as figure 4 shows. The difference between dialogue is that the transition functions of B, f^B_A and f^B_C, do not produce respectively an output stream for A and B but the opposite, as shown by figure 5.

There are two ways for implementing trialogue. The first one is defined in figure 7[6]. The trialogue function has six parameters: *inputs*$_A$, *inputs*$_C$, *step-fcn*$_A$, *step-fcn*$_C$ and the original *initial-state* and *result-fcn*. An agent has a transition function dedicated to each agent with which it communicates. It gets the message from *inputs*$_A$ and produces the answer message in *outputs*$'_C$ before getting the message from *inputs*$_C$ which is only produced after C has interpreted the message B has just sent. If *step-fcn*$_A$ incites to terminate the trialogue, then *step-fcn*$_C$ is not applied. The trialogue function returns a stream of five elements: *unused-inputs*$_A$, *unused-inputs*$_C$, *outputs*$_A$, *outputs*$_C$, *result*. So, in a case such as the one illustrated by figure 4, A and C run dialogue as in section 4.3, and B call to dialogue is:

$$trialogue : \langle I^B_A\ I^B_C\ \beta_0\ f^B_A\ f^B_C\ R_B\rangle \rightarrow (I^B_A\ I^B_C\ O^B_A\ O^B_C\ val)$$

The second way of implementing trialogue consists in giving as parameters two input streams but a single *step-fcn* processing the two streams. Then this *step-fcn* must return five elements (2 unused-streams, 2 output streams and a result). This way is not detailed here by lack of space. The main drawbacks of this second way are: i) the fact that it is more complicated to program, ii) that with a *step-fcn* processing two streams, all the language and all applications of functions must be lazy (not simply stream processing functions). Indeed, when the *step-fcn* is applied the value of the first stream could be evaluated, but not the value of the second one: its evaluation must be delayed (because the messages inside have not been produced yet). So, in a traditional applicative language where all arguments are evaluated before being applied, the first way should be used while in a lazy language the two ways are suitable.

Remark: Note that in trialogue the *unused-inputs* stream produced by the first *step-fcn* is accessible while the second *step-fcn* is applied. In figure 7, when *s-fcn*$_C$ is applied *in*$'_A$ is accessible. Then, if *s-fcn*$_C$ was written (or is dynamically changed, cf. section 6.1) in order to process it, B can access the messages previously sent by A but not interpreted during *s-fcn*$_A$ application. This remark stays recursively true in i-dialogue.

[6]Due to space economy we use abbreviations in this figure.

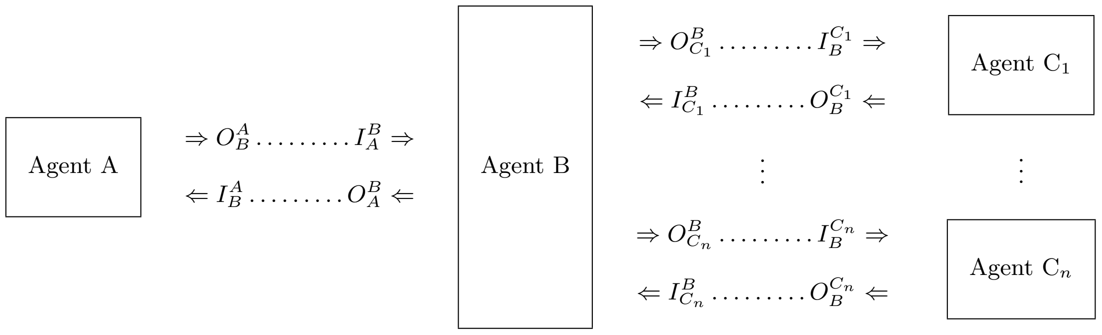
Agent A
$\Rightarrow O_B^A \ldots\ldots\ldots I_A^B \Rightarrow$
$\Leftarrow I_B^A \ldots\ldots\ldots O_A^B \Leftarrow$
Agent B
$\Rightarrow O_{C_1}^B \ldots\ldots\ldots I_B^{C_1} \Rightarrow$
$\Leftarrow I_{C_1}^B \ldots\ldots\ldots O_B^{C_1} \Leftarrow$
$\Rightarrow O_{C_n}^B \ldots\ldots\ldots I_B^{C_n} \Rightarrow$
$\Leftarrow I_{C_n}^B \ldots\ldots\ldots O_B^{C_n} \Leftarrow$
Agent C$_1$
Agent C$_n$

Figure 6: Streams in i-dialogue between agents A,B and C$_1 \ldots$ C$_n$

```
trialogue : ⟨$in_A$ $in_C$ init-s s-$fcn_A$ s-$fcn_C$ r-fcn⟩ ≡
  letrec
    run ≡ λ ⟨$in_A$ $in_C$ state⟩ .
      let
        ($in'_A$ $out'_C$ state' done') ≡ s-$fcn_A$ : ⟨$in_A$ state⟩
      in
        if done'
          then ($in'_A$ $in_C$ null $o'_C$ r-fcn:state')
          else
            let
              ($in'_C$ $out'_A$ state" done") ≡
                s-$fcn_C$ : ⟨$in_C$ state'⟩
            in
              if done"
                then ($in'_A$ $in'_C$ $out'_A$ $out'_C$ r-fcn:state")
                else
                  let
                    ($in''_A$ $in''_C$ f-$out''_A$ f-$out''_C$ result") ≡
                      run : ⟨$in'_A$ $in'_C$ state"⟩
                  in
                    ($in''_A$ $in''_C$
                    append-ll : ⟨$out'_A$ f-$out''_A$⟩
                    append-ll : ⟨$out'_C$ f-$out''_C$⟩
                    result")
  in
    run : ⟨$in_A$ $in_C$ init-s⟩
```

Figure 7: Definition of the function *trialogue*

5.2 Generalization: the i-dialogue abstraction

The term i-dialogue is the abbreviation of *intertwined-dialogue*. An i-dialogue aims to model conversations between an agent and a group of agents. These conversations are dialogues intertwined together and must be executed in the same time as their inputs and outputs depend each others. These situations (described by figure 6) can not be expressed by the dialogue abstraction. Concretely, the i-dialogue abstraction is a generalization of trialogue. The main idea consists in processing several inputs coming from several agents in a special order and being able to process each input stream to process the next output one.

The implementation of i-dialogue follows the same principles as dialogue and trialogue (1st way). Instead of giving as parameter several $inputs_i$ and several $step\text{-}fcn_i$, i-dialogue takes as parameter a list of input streams, *l-inputs*, and a list of transition functions, *l-step-fcn*. Then, the implementation of i-dialogue consists just in a classic list recursion ! (cf. figure 8). Note that i-dialogue can realize all the calls of dialogue (or trialogue). If both the *l-inputs* and *l-step-fcn* have only one element then i-dialogue is equivalent to dialogue. In a case such as the one illustrated by figure 6, calls to i-dialogue are, and produce:

A: *i-dialogue* : $\langle\langle I_B^A\rangle\ \alpha_j\ \langle f_B^A\rangle\ R_A\rangle \rightarrow (\langle I_B^A\rangle\ \langle O_B^A\rangle\ val)$

B: *i-d...* : $\langle\langle I_A^B I_{C_1}^B \ldots I_{C_n}^B\rangle\ \beta_0\ \langle f_A^B f_{C_1}^B \ldots f_{C_n}^B\rangle\ R_B\rangle$

$\rightarrow (\langle I_A^B I_{C_1}^B \ldots I_{C_n}^B\rangle\ \langle O_A^B O_{C_1}^B \ldots O_{C_n}^B\rangle\ val)$

C$_1$: *i-d...* : $\langle\langle I_B^{C_1}\rangle\ \gamma_{10}\ \langle f_B^{C_1}\rangle\ R_{C_1}\rangle \rightarrow (\langle I_B^{C_1}\rangle\ \langle O_B^{C_1}\rangle\ val)$

⋮ ⋮

C$_n$: *i-d...* : $\langle\langle I_B^{C_n}\rangle\ \gamma_{n0}\ \langle f_B^{C_n}\rangle\ R_{C_n}\rangle \rightarrow (\langle I_B^{C_n}\rangle\ \langle O_B^{C_n}\rangle\ val)$

In the i-dialogue function the ordering of the elements of the lists (*l-inputs* and *l-step-fcn*) is important. Indeed, it corresponds to the semantics of the i-dialogue, that means the order in which the inputs are processed by the agent running the i-dialogue. This semantics is determined by the agent before the starting of the i-dialogue. Thus, if an agent runs an i-dialogue for computing the sum of squares of numbers by interacting with an agent computing sums (A-sums) and an agent computing squares (A-squares) then it must send the numbers firstly to A-squares ant then send the result to A-sums, because the two operations are not commutative.

However, if the second way for implementing trialogue in section 5.1 is extended to i-dialogue (that means instead of using a list of $step\text{-}fcn_i$, using only one *step-fcn* processing all the streams) then the semantics of the i-dialogue is not anymore determined by the order as before, but directly by the *step-fcn*.

Remark – Note that streams used in i-dialogue are even lazy lists because when agent B runs the i-dialogue function with $I_A^B, I_{C_1}^B, \ldots, I_{C_n}^B$, the first elements of $I_{C_1}^B, \ldots, I_{C_n}^B$ cannot be determined because the first elements of $O_{C_1}^B, \ldots, O_{C_n}^B$ needed for $C_1, \ldots, C_n$ to produce an answer, have not been produced yet.

```
i-dialogue : ⟨l-inputs initial-state l-step-fcn result-fcn⟩ ≡
  letrec
    iter ≡ λ ⟨listi listf listui listo state⟩ .
      if null?:listi
        then (listui listo state #f)
        else
          let
            (in' out' state' done') ≡  [car:listf] : ⟨car:listi state⟩
          in
            if done'
              then (append : ⟨listui cons:⟨in' cdr:listi⟩ ⟩
                     append : ⟨listo
                                cons:⟨out'
                                       map:⟨λ x . null cdr:listi⟩ ⟩ ⟩
                     state' #t)
              else
                iter : ⟨cdr:listi
                        cdr:listf
                        append : ⟨listui ⟨in'⟩ ⟩
                        append : ⟨listo ⟨out'⟩ ⟩
                        state'⟩
    run ≡ λ ⟨l-inputs state⟩ .
      let
        (l-inputs' l-outputs' state' done') ≡
          iter : ⟨l-inputs l-step-fcn null null state⟩
      in
        if done'
          then (l-inputs' l-outputs' result-fcn:state')
          else
            let
              (l-inputs" future-l-outputs" result") ≡
                run : ⟨l-inputs' state'⟩
            in
              (l-inputs"
              append-ll-with-list : ⟨l-outputs' future-l-outputs"⟩
              result")
  in
    run : ⟨l-inputs initial-state⟩
```

Figure 8: Definition of the function *i-dialogue*

6. RELATED WORK AND APPLICATIONS

The utility of applicative/functional features (especially dynamic typing and lazy evaluation) for the Web (the future environment of agents) was previously presented in [3]. Besides, the idea to bring closer features from applicative/functional languages and MAS is also suggested in [5] which presents Java agents using stream of messages. Especially, we think that the principles of i-dialogue abstraction fit multi-agent scenarios because it respects the pre-requisite of communication in MAS i.e. do not pre-suppose about internals of the partner and only deal with messages exchanged. Moreover, it is a useful abstraction for describing the interaction between two "system programs" (AA-AA interaction) but it can equally well describe an interactive session between a human user and a program (HA-AA interaction), or even between two persons (HA-HA interaction). The HA can "start" an i-dialogue with an AA: Each time he receives a message from the AA, the HA enters a new message, which depends both of the received message and of the history of the interaction (i.e., the HA's old state). This kind of characteristic, aiming to introduce/integrate HA in MAS, is more and more important in the evolution of nowadays MAS and even distributed systems in general. MAS community tends to do the necessary shift putting the user towards the center of the agent applications. In order to do that, uniform models for AA-AA and HA-AA interactions are needed. It is one of the principle of the STROBE model presented below.

6.1 The STROBE model

In [4] and more recently in [15], the authors present the STROBE model as an agent representation and communication model. In STROBE, generic communication may be described by means of *STReams* of messages to be exchanged by agents represented as *OBjects* exerting control by means of procedures (and continuations) and interpreting messages in multiple *Environments*. The model is highly influenced by applicative/functionnal constructs. For STROBE agents, message's interpretation is done in a given *Cognitive Environment*[7] and with a given *Cognitive Interpreter* both dedicated to the current conversation (a pair for each acquaintances). Moreover, communication enables agents to dynamically change values in an environment and especially change (at run time) these interpreters to adapt their way of interpreting messages (meta-level learning). As the Cognitive Interpreters are "included"[8] in Cognitive Environments, the state of an agent is the composition of all its Cognitive Environments. Actually, these pairs environment – interpreter play the role of "partner models" in order to be able for an agent to reconstruct as much as possible, the partner's internal state.

One of the original idea of the model is to represent messages by streams. Agents generate the next message to send only after having received the last partner's answer. This allows to remove the necessity of global conversation planning (i.e. communication protocol) substituting it by history memorization and one-step, opportunistic planning.

Note that in i-dialogue the *step-fcn*s take an input stream as argument, but noting implies they should begin by processing the first element of this stream (i.e. the first message send by the partner agent); they may decide to process any element of the input stream at any time. In the same way, the STROBE model suggest to use a *dynamic scheduler* to "choose" the next message an agent wants to interpret.

Therefore, the i-dialogue abstraction seems particularly synergic with the STROBE model: both use streams and lazy evaluation and both aim to model dynamic agent interactions. Moreover, the Cognitive Interpreters are quite the same thing that the *step-fcn*s of the i-dialogue abstraction: they are functions producing an answer message (i.e. output) according to an incoming message (i.e. input) and an environment (i.e. state). What is really interesting in the STROBE model is the fact that these interpreters could evolve dynamically (while communicating) so, by analogy, an unification of the i-dialogue abstraction with the

[7]The term environment is here used with its programming language meaning, that is to say, a structure that binds variables and values.

[8]In the same way that the function `eval` is part of the Scheme language.

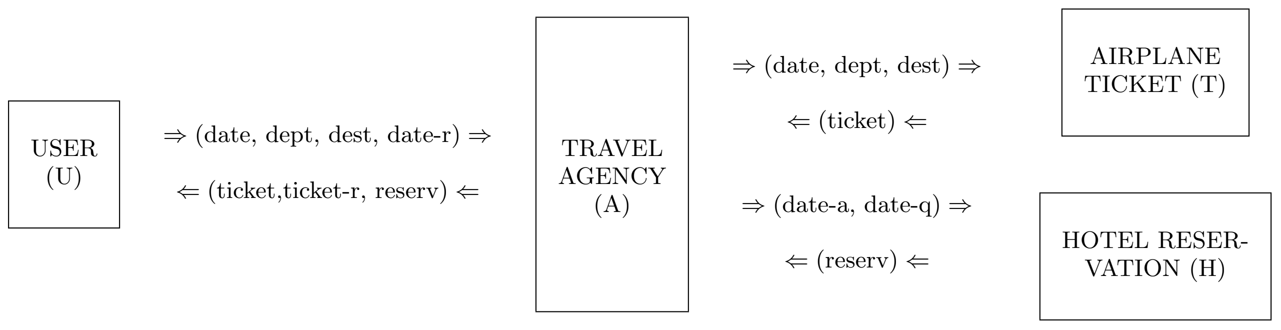

Figure 9: Streams in the travel agency scenario

STROBE model should allow agents to dynamically change *step-fcn*s during the execution of the i–dialogue function. Such a feature seems very attractive for the dynamicity of conversations being modeled by the i-dialogue abstraction.

6.2 Providing services

The *step-fcn*s of i-dialogue represent both the agent evolution functions (by changing the agent state) but also the agent capabilities (by producing outputs according to inputs). Then an agent executing an i-dialogue function provides a service realized by its *stef-fcn*s. Besides, we can nevertheless explain that the i-dialogue abstraction suits for providing services within scenarios such as the ones on today's Web and Grid. Without a doubt, in figure 6 the agent A may be a service user (AA or HA), the agent B may be the service provider and the group of agents $C_1 \ldots C_n$ may represent the network (e.g. the Web or the Grid) to which agent B is connected and which allows it to provide several services. In this scenario, as each C_i agent provides a service to B, i-dialogue models the composition of all theses services. Composed services (or business processes), are the subject of current applied research and a central concern for the future generation of services as it is explained, among others, in [20], a recent overview of Service Oriented Computing.

The next generation of services will consists in dynamically generated services, i.e. services constructed on the fly by the provider according to the conversation it has with the user. Dynamic Service Generation (DSG) is opposed to classical product delivery approach; just like in current life, buying ready-to-wear clothes is analogue to asking for a product, whereas having clothes made by a tailor is analogue to requiring a service to be generated. For going toward this kind of DSG, the notion of service on the Web (Web services, Grid services) has to surpass HTTP protocols, RPC (Remote Procedure Call) and XML standards to be enriched by DAI research and especially by agent communication progress. The main difference between DSG and product delivery is the emerging conversational process managed by agents in the first one: future DSG systems need open and dynamic communication models able to generate these conversational processes which may occur between agents: AA-AA, AA-HA, and also HA-HA, as services may be asked/produced by any kind of agents. The i-dialogue abstraction represents a concrete fundamental model for this challenging goal.

6.3 Example: the travel agency

In this section we illustrate a scenario using i-dialogue with a classical example in service composition: the travel planning! A travel agency agent (A) provides to an user agent (U), both an airplane ticket booking service and hotel reservation service coordinated together, by composing the service provided by an airplane ticket agent (T) and the service provided by a hotel reservation agent (R), as illustrated by figure 9. The *step-fcn*s are detailed below (with their side effects):

$f_U^A : \langle(\text{date dept dest date-r})\ \alpha\rangle \rightarrow$
$\langle()(\text{date dept dest date-r dest dept})\ \alpha\ \text{b}\rangle$

f_U^A: The agent U produces a stream of parameters which contains variables of the trip, i.e. departure date, departure city, destination city and return date. The agent A consumes all this stream and produces a stream for T composed of two sequences defining the two tickets the user requests, i.e. date, departure city, destination city. The state, α, of agent A is not changed by f_U^A.

$f_A^T : \langle(\text{date dept dest})\ \tau\rangle \rightarrow \langle()(\text{ticket})\ \tau'\ \text{b}\rangle$
- Change the database of T

f_A^T: The agent T consumes the entire stream produced by A and for each triplet date, departure city, destination city, it produces a one-way ticket. The state, τ, of agent T is changed by producing ticket(s) (the database of T is updated to consider the new ticket(s) booked). The agent T produces a stream containing the ticket(s).

$f_T^A : \langle(\text{ticket ticket-r})\ \alpha\rangle \rightarrow \langle()(\text{date date-r})\ \alpha'\ \text{b}\rangle$
- Remember ticket and ticket-r
- Extract date and date-r from ticket and ticket-r

f_T^A: The agent A consumes the entire stream produced by T and for each ticket, it extracts the date of the ticket, in order to ask to R an hotel reservation. Note that the date extracted from ticket are not necessarily the same that the date firstly produced by U. The agent A produces a stream of two dates, an arrival date and a quit date. It also changes its state, α, by memorizing the two tickets returned by T.

$f_A^R : \langle(\text{date-a date-q})\ \rho\rangle \rightarrow \langle()(\text{reserv})\ \rho'\ \text{b}\rangle$
- Change the database of R

f_A^R: The agent R consumes the entire stream produced by A and for each pair of date produce a hotel reservation. The state, ρ, of agent R is changed by producing reservation(s) (the database of R is updated to consider the new reservation(s)). The agent R produces a stream containing the reservation.

$$f_R^A : \langle(\text{reserv})\ \alpha\rangle \rightarrow \langle()(\text{ticket ticket-r reserv})\ \alpha'\ \text{b}\rangle$$
- Extract ticket and ticket-r from state

f_R^A: Finally, the agent A consumes the stream produced by R and produces a stream directed to U with the reservation and the two tickets previously memorized. A new state is thus computed. The boolean returns by f_R^A is probably #t to express the fact that for A, the service providing is over.

Remark: In this scenario, to simplify, messages (i.e. elements of streams) consist just of variables. Real agent messages are more complex.

In this scenario, the use of streams implies that the values of date, departure city and destination city variables of O_A^U are determined only when the agent T needs them (i.e. when one of its database primitive forces the evaluation) and not when A produces O_T^A. Actually, the user U firstly asks for a service and then specifies its parameters only when the service is sure to be applied. The user parameters are determined one by one progressively when they are really needed which differs from a traditional RPC as it is the case in classical service approaches (Web service). This idea goes toward DSG.

An integration of i-dialogue and STROBE allowing agents to change dynamically the *step-fcn*s could be very interesting in this scenario by changing at run time, for example, f_R^A and f_A^R. The former could be changed in order to produce a different stream, including for example customer services requested by the user (i.e. parameters of I_U^A, returned in *unused-inputs* when A applied f_U^A, and used after as explained in remark section 5.1), and the latter could be changed to process this new stream. If these changes depends on the A's initiative, then f_R^A changes are internal, but f_A^R changes are externals (does not concerns agent A) and should be accomplished by communicating. These kinds of changes on the way messages are interpreted are considered as "meta-level learning by communicating" and are typical of STROBE agents as illustrated in [15].

6.4 Weaknesses and perspectives

I-dialogue does not directly model multiple conversation where each agent can send messages to any others agent or broadcast messages to all, as it can occur in a "human meeting". Using i-dialogue an agent can decide which part of an input stream it wants to process, but cannot for the moment change dynamically the order in which it process all the inputs streams together. Or, add dynamically one or several new stream(s) to process. The ordering is not dynamic yet. For example in the travel agency scenario, the ordering of agent A (f_U^A, f_T^A, f_R^A) could be changed to include a new agent in the conversation providing a car rental service. The *step-fcn*s may change (as it is explain just before) but also the ordering of the elements of *l-inputs* and *l-step-fcn*. It is a perspective for future work based on streams and lazy evaluation.

7. CONCLUSION

We have presented in his paper the i-dialogue abstraction, a model of multiple concurrent conversations in MAS. Each agent can use i-dialogue locally to represent its conversations. This abstraction is based on simple constructs of applicative / functional languages (i.e. streams, lazy evaluation and higher order functions). It has two main advantages: i) to not presuppose anything about the internals of the partner agent and only deals with outputs stream of messages; ii) to deal with the entire conversation (expressed by potentially infinite and not predetermined stream of messages) and not simply with a message alone.

The main contributions of this paper are:

- To spread the elegant model of O'Donnell to more complex situations implying several entities.
- To consider this quite new model for MAS and within agent communication as it was suggested by STROBE.
- To open a new kind of consideration in composition of services as it can occur in DSG.

8. ACKNOWLEDGMENTS

Work partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th Framework Programme for RTD - project ELeGI, contract IST-002205 (www.elegi.org). This document does not represent the opinion of the European Community, and the European Community is not responsible for any use that might be made of data appearing therein.

9. REFERENCES

[1] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, USA, 2nd edition, 1996.

[2] W. H. Burge. Stream processing functions. *IBM Journal of Research and Development*, 19(1):12–25, January 1975.

[3] S. A. Cerri. Dynamic typing and lazy evaluation as necessary requirements for Web languages. In *European Lisp User Group Meeting, ELUGM'99*, Amsterdam, Holland, 1999.

[4] S. A. Cerri. Shifting the focus from control to communication: the STReam OBjects Environments model of communicating agents. In P. J.A., editor, *Collaboration between Human and Artificial Societies, Coordination and Agent-Based Distributed Computing*, volume 1624 of *Lecture Note in Artificial Intelligence*, pages 74–101. Springer-Verlag, Berlin Heidelberg New York, 1999.

[5] T. Clark. Implementation of lazy agents in the functional language ebg. Technical report, University of Bradford, West Yorkshire, UK, July 1999.

[6] F. Dignum and M. Greaves, editors. *Issues in Agent Communication*, volume 1916 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin Heidelberg New York, 2000.

[7] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison Wesley Longman, Harlow, UK, 1999.

[8] J. Franco, D. P. Friedman, and S. D. Johnson. Multi-way streams in scheme. *Computer Languages*, 15(2):109–125, 1990.

[9] D. P. Friedman and D. S. Wise. CONS should not evaluate its arguments. In S. Michaelson and R. Milner, editors, *Automata, Languages and Programming: Third International Colloquium*, pages 257–284. Edinburgh University Press, 1976.

[10] F. Guerin. *Specifying Agent Communication Languages*. PhD thesis, Imperial College of Science, University of London, London, UK, June 2002.

[11] M.-P. Huget, editor. *Communication in Multiagent Systems, Agent Communication Languages and Conversation Poliscies*, volume 2650 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin Heidelberg New York, 2003.

[12] J. Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[13] T. Ida and J. Tanaka. Functional programming with streams. In R. E. A. Mason, editor, *Information Processing 83, IFIP 9th World Computer Congress*, pages 265–270, Paris, France, September 1983.

[14] S. D. Johnson. How daisy is lazy: Suspending construction at target levels. Technical Report 286, Indiana University Computer Science Department, Bloomington, Indiana, USA, August 1989.

[15] C. Jonquet and S. A. Cerri. Agents communicating for dynamic service generation. In *1st International Workshop on GRID Learning Services, GLS'04*, pages 39–53, Maceio, Brazil, September 2004. Revised and extended version to appear in AAIJ, September 2005.

[16] P. J. Landin. Correspondence between ALGOL 60 and Church's Lambda-notation: part I. *Communications of the ACM*, 8(2):89–101, February 1965.

[17] J. T. O'Donnell. Dialogues: A Basis for Constructing Programming Environments. *ACM SIGPLAN Notices*, 20(7):19–27, June 1985. Proceedings of the symposium on Language issues in programming environments, Seattle, Washington, USA, June 1985.

[18] P.-M. Ricordel, S. Pesty, and Y. Demazeau. About conversations between multiple agents. In *1st International Workshop of Central and Eastern Europe on Multi-agent Systems, CEEMAS'99*, pages 203–210, St. Petersburg, Russia, June 1999.

[19] M. P. Singh. Agent communication languages: Rethinking the principles. *IEEE Computer*, 31(12):40–47, 1998.

[20] M. P. Singh and M. N. Huhns. *Service-Oriented Computing Semantics, Processes, Agents*. John Wiley & Sons, Ltd, 2005.

APPENDIX

A. DAISY LANGUAGE INSPIRED SYNTAX

A part of this section is quoted from [17]. Some changes have been done specially to make the distinction between "evaluated list" and "lazy list". In the descriptions below, ϵ may be any expression and η may be any identifier list (evaluated or lazy); an identifier list is an identifier or a list of identifier lists.

1. $\langle \epsilon_0\ \epsilon_1\ \ldots\ \epsilon_n \rangle$
   An "evaluated list" is a list of expressions enclosed in angle brackets. Lists are constructed with **cons**, **append** and **map**, and accessed with **car** and **cdr**. The predicate **null?** taste if a list is **null**.
2. $(\epsilon_0\ \epsilon_1\ \ldots\ \epsilon_n)$
   A "lazy list" or a stream (cf. section 4.2 for the exact distinction) is a list of expressions enclosed in brackets. The function **append-ll** is equivalent to **append** but for lazy-list. The function **append-ll-with-list** apply **append-ll** to the elements of the list in arguments. Cf. appendix B for implementation of streams in Scheme.
3. $\lambda\ \eta\ .\ \epsilon$
   Function definitions. A lambda expression evaluates to a closure, as in Scheme. η represents the parameter(s) and ϵ is the body of the function.
4. $\epsilon_0\ :\ \epsilon_1$
   Function applications are written with an infix apply operator ":". The expression to the left of the ":" should evaluate to a closure. In this syntax functions take exactly one parameter, but that parameter may be an evaluated list. For example, inc : 3 and mpy : $\langle 3\ 4 \rangle$ compute 3+1 and 3*4 respectively.
5. **if** ϵ_0 **then** ϵ_1 **else** ϵ_2
   The if expression. ϵ_1 is evaluated if ϵ_0 is true (#t) or ϵ_2 is evaluated if ϵ_0 is false (#f).
6. **let** $\eta_0 \equiv \epsilon_0\ \eta_1 \equiv \epsilon_1\ \ldots$ **in** ϵ
   The **let** expression evaluates the right hand sides of the equations in the existing environment, binds the resulting values to the left hand sides, and evaluates ϵ in the new environment.
7. **letrec** $\eta_0 \equiv \epsilon_0\ \eta_1 \equiv \epsilon_1\ \ldots$ **in** ϵ
   The **letrec** expression is like **let**, except that the right hand sides of the equations are evaluated in the extended environment, rather than the original environment. This makes it possible to define sets of recursive functions or recursive data structures in a **letrec**.

B. SCHEME IMPLEMENTATION

To experiment the work described in this paper, we implemented the i-dialogue function in Scheme, a purely applicative language. Scheme R^5RS is not a lazy language, but it offers the function `force` and the macro `delay` in library which allow, combined with the macro operator (special form) `define-syntax`, to implement streams and lazy lists. Besides, the second way of implementing trialogue cited in section 5.1 was tested and verified with a lazy Scheme meta-evaluator. Some simple examples have been developed. These implementations are available. An integration of i-dialogue and STROBE is in progress over the multi-agent platform Madkit (www.madkit.org).