Compiling dynamic agent conversations[1]

Pierre Bonzon

HEC, University of Lausanne
1015 Lausanne, Switzerland
pierre.bonzon@unil.ch

Abstract. We consider defining executable dialogues for communicating agents. Towards this end, we introduce agent classes whose communication primitives are based on deduction. Their operational semantics are given by an abstract logical machine that is defined purely in sequential terms. These agents communicate under the control of plans requiring a synchronization flag. These plans can be rewritten as dialogues with an implicit synchronization. Reversibly, dialogues can be compiled back into plans and then executed on the sequential machine. Sub-dialogues can be entered from any dialogue, such achieving dynamic conversation structures.

1 Introduction

Communication in multi-agent systems can be exemplified using the FIPA proposal [3]. In this approach, a set of normative *communicative acts* (or messages) is first defined. They represent the buildings blocks of a dialogue between agents. At an aggregated level, agent *interaction protocols* define generic sequences of messages representing a complete conversation (or dialogue) between agents. This enables agents to anticipate each other response according to some *conversation patterns.* Although the FIPA specifications contain predefined protocols, they do not impose upon agent to follow these standards (i.e. agent developers can adopt their own protocols). Conversations that are defined in this way have a fixed structure that can be laid down using some kind of graphical representation. Formalisms that have been proposed for this purpose include *graphs* generated by deterministic finite automata [3], colored Petri *nets* [6], or UML sequence *diagrams* [7].

This approach can be qualified as *static*, in the sense that every possible move must be made explicit beforehand and expressed in terms of alternative sequences of communicating acts (with possible feedbacks or resume). In particular, there is no provision for *composing* existing protocols on demand e.g., it is not possible for a given protocol to call another protocol at run time, and thus define *dynamic* conversation structures. In many ways, this situation is truly reminiscent of the early days of computer programming i.e., at the time when programs were monolithic objects lacking decomposition into procedures, function or subprograms. These programs, just like FIPA protocols, were truly static objects, and there was no such thing as a stack of activation records reflecting the dynamic embedding of successive procedure calls.

1 Revised version from *Advances in Artificial Intelligence*, LNAI, vol 2479, Springer Veralg 2002

We strongly believe that there is a definite need for dynamic conversation structures i.e., for a model of agent conversation that would include the embedding of successive and/or nested protocol calls within a multi-agent system state. First of all, a much higher degree of modularity and reusability could be thus achieved. Secondly, and perhaps more importantly, this should allow for an easier diagnosis and recovery from the various deadlocks that can occur in a conversation i.e., when one party fails to answer as expected. We have been looking therefore for a model of *dynamic conversation structures.* Furthermore, we wanted to come up with a *conversation language* that would lead to directly executable specifications.

As a prerequisite, a formal model of agent communication at the message level is required. Recently, Hendriks and al. have advocated a new approach based on synchronized pairs of communication primitives [5]. Defined as "neutral" actions enjoying a well-defined and clear semantics, these logical primitives can be used for many different purposes, including the implementation of speech acts. In order for two agents to communicate, both parties must first agree to an exchange (e.g., by independently using an external exchange protocol based on these synchronized pairs). Each exchange then involves either a deductive or an abductive task performed independently by one of the agents. As an example (that we develop later), this can be used in collaborative models to synchronize successive negotiation rounds as well as the successive steps in each round.

In order to get executable specifications, the operational semantics of the complete model must be given in sequential terms. Towards this end, we first extended a general model of individual agent with sensing [9] to include the notion of *non-deterministic plans* (or *nd-plans* in short). Originally given in abstract functional terms, this model was developed into a set of concrete procedures that represent a sequential abstract machine generating runs for non-deterministic agents. We then integrated within this framework a simplification of the communication model [5]. In the resulting model [1], two agents communicate under the control of nd-plans requiring each a synchronization flag. These plans are directly executable on our abstract machine (in essence, a sequential machine with deductive capabilities).

Our goal in this paper is to try and obtain executable specifications of agent dialogues that do not require an explicit synchronization. Towards this end, we shall consider a communication language that will be defined in terms of *branching sequences.* This language will allow us to rewrite nd-plans as *dialogues.* Reversibly, dialogues can be compiled back into nd-plans, and then executed on the original machine. Our main result is that dialogues can be *deterministically* and *simply sequentially* executed using a single pair of synchronization and sequencing flags. Furthermore, as sub-dialogues can be entered from any dialogue, we thus achieve dynamic conversation structures.

The rest of this paper is organized as follows: in section 2, we review the definition of deductive communicating agents. Section 3 proposes a language for writing agent dialogues that are equivalent to nd-plans. Section 4 gives its operational semantics via compiling functions for translating dialogues into nd-plans. This approach is finally illustrated with an example of a multi-round negotiation.

2 A model of deductive communicating agents

As a prerequisite for the interpretation and/or compilation of executable agent dialogues, we first need to define and implement a model of communicating agents. Towards this end, we used and simplify the proposal made by Hendriks and al. [5]. Following a purely logical approach, they introduced two pairs of neutral communication primitives i.e., *tell/ask* and *req/offer* that correspond to data exchanges enjoying a well-defined semantics and can be used for many different purposes. In each pair, *r* is designated as the *receiver* and *s* as the *sender.* The first pair can be defined as follows:

> Message *tell(r,φ)* from sender *s* provides *r* with data φ, and message *ask(s,ψ)* from receiver *r* expresses his willingness to solve a query ψ using any data sent by *s*. Both messages are sent without reciprocal knowledge of what the other agent wants or does. In particular, the data φ volunteered by *s* is not given in response to *r*'s asking. If these two messages are put together through some kind of external handshake or synchronization, then by using both his own knowledge and the data φ told by *s*, receiver *r* will try and answer his query ψ. Formally, receiver *r* will compute the most general substitution θ such that
>
> $l^r \cup \varphi \vdash \psi\theta$
>
> where l^r represents the local state of the receiver. As the unknown substitution θ stands after the deduction sign $\vdash$, i.e. within the conclusion, this amounts to a classical *deductive* task.

According to Hendricks & al., ψ in ask*(s,ψ)* can contain free variables but φ in *tell(r,φ)* must be closed; furthermore, $l^s \vdash \varphi$ is not required (i.e., *s* is not required to be truthful or honest). We shall illustrate this type of exchange through a simple example. Let the local state l^r of *r* be such that

$l^r \vdash father(abram,isaac) \wedge father(isaac,jacob)$

and let us consider the following pair of messages

message sent by *s*: $tell(r, \forall XYZ\ father(X,Y) \wedge father(Y,Z) \Rightarrow grandfather(X,Z))$
message sent by *r*: $ask(s, grandfather(X,jacob))$.

In this first scenario, *s* tells *r* a closed implication, and *r* asks *s* for some data that could allow him to find out who is the grandfather of *jacob.* Using the data sent by *s*, *r* is then able to deduce the substitution *X=abram.*

In contrast, the second pair can be defined as follows:

> Message *req(r,φ)* from sender *s* requests *r* to solve query φ, and message *offer(s,ψ)* from receiver *r* expresses his willingness to use data ψ for solving any query submitted by *s.* When put together, these two messages will lead the receiver *r* to find the possible instantiations of his free variables in ψ that allow him to deduce φ. Formally, receiver *r* will compute the most general substitution θ such that
>
> $l^r \cup \psi\theta \vdash \varphi$.
>
> As the unknown substitution θ stands before the deduction sign $\vdash$, i.e. within the premises, this amounts to a non-classical *abductive* task.

According to Hendricks & al., φ in *req(r,φ)* must be closed but ψ in *offer(s,ψ)* can contain free variables; furthermore, $l^r \vdash \psi$ is not required, but $l^r \vdash \neg\psi$ is not allowed. To illustrate this second type of exchange, let us consider the following pair of messages

message sent by *s*: *req(r, ∃X grandfather(X,jacob))*
message sent by *r*: *offer(s, father(X,Y) ∧ father(Y,Z) ⇒ grandfather(X,Z))*.

In this second scenario, *s* requests *r* to find out if there is a known grandfather for *jacob*. Independently, *r* offers *s* to abduce a substitution for the free variables in his ψ that would allow him to answer. In this case, using his knowledge contained in l^r and the implication he offers, *r* can abduce the same substitution as before.

In both of the above exchanges, no data is sent back to *s*, and the corresponding formal semantics captures the processing done by *r* only. In other words, the sender will not be aware of the results of the receiver's computation. For the sender to get this result, a reversed exchange (e.g. an *ask/tell*) is needed. While this is perfectly appropriate for the first type of exchange (after-all, the sender who volunteers data is not necessarily interested in the receiver's computations), we feel that, in the second case, the sender who expresses a need for data should automatically benefit from the receiver's computations. Furthermore, as abductions are difficult to achieve and implement, we favor exchanges that do not rely on abduction. Giving up the *req/offer* pair, we thus defined and implemented instead a simplified *call/return* pair that relies on deduction only and makes the results of the receiver's computations available to the sender. By doing so, we did end up with a less powerful model. It is interesting to note however that all *req/offer* examples given in [5] can be expressed as *call/return* invocations. In particular, if the receiver's local state includes closed forms of his offer ψ, then a *req(r,φ)/offer(s,ψ)* pair reduces to a *call/return* pair. This new pair is defined as follows:

In the *call(r,φ)/return (s,ψ)* pair, both φ and ψ can contain free variables. This exchange is then interpreted as the sender *s* calling on *r* to instantiate the free variable in his query φ. Independently, the receiver *r* is willing to match his query ψ with the sender's φ and return the instantiations that hold in his own local state. Formally, receiver *r* will *deductively* compute the substitution θ s. t.

$\varphi\theta = \psi\theta$ and $l^r \vdash \psi\theta$.

As indicated above, this information will be made available to the sender, i.e. substitution θ will be sent back to *s*.

To illustrate this, let us suppose that we now have

$l^r \vdash$ *father(abram,isaac)∧father(isaac,jacob)∧*
∀XYZ father(X,Y) ∧ father(Y,Z) ⇒ grandfather(X,Z)

message sent by *s*: *call(r, grandfather(X,jacob))*
message sent by *r*: *return(s, grandfather(X,Y))*.

This exchange is to be interpreted as *s* calling on *r* to find out the grandfather of *jacob* i.e., to instantiate the free variable in his query. Independently, *r* is willing to match the sender's call and to return the substitutions that hold in his local state. Once

again the substitution *X=abram* will be found. In contrast to the previous exchanges however, this information will be sent back to the sender.

3 A language for agent dialogues

In order to first get an intuitive feeling for the language we have implemented, we shall start with an illustrating example. Towards this end, let us consider a simplified version of the *two-agent meeting-scheduling* example of [5]. In this application, one agent is designated as the *host* and the other one as the *invitee*. Both agents have free meeting slots e.g.,

$l^{host} \vdash meet(13) \wedge meet(15) \wedge meet(17)$

$l^{invitee} \vdash meet(14) \wedge meet(16) \wedge meet(17)$

and they must find their earliest common slot (in this case, *17*). We shall make use of a predicate *epmeet(T1,T)* meaning *"T1 is the earliest possible meeting time after T "*, defined as

$meet(T1) \wedge (T1 >= T) \wedge \neg(meet(T') \wedge (T' >= T) \wedge (T' < T1)) \Rightarrow epmeet(T1,T).$

The solution involves successive negotiation *cycles*. The host has the responsibility of starting each cycle with a given lower time bound *T*. A cycle comprises three steps, each step involving a exchange of messages. In the first step, the host initializes a *call/return* exchange calling on the invitee to find out his earliest meeting slot *T1* after *T*. In the second step, roles are swapped: the invitee initializes a *call/return* calling on the host to find out his earliest meeting slot *T2* after *T1*. In the third step, the host either confirms an agreement on time *T2* (if *T1=T2*) by initializing a *tell/ask* exchange, or starts a new cycle with *T2* as his new lower bound.

This solution can be informally expressed as follows:

"start with a *call/return* exchange,
proceed with a *return/call* exchange,
conclude with a *tell/ask* exchange or *resume*"

A formal rewriting under the form of two synchronized dialogues is then

dialog*(invite(Invitee, T), [T1,T2],*
*[**call**(Invitee,epmeet(T1,T)),*
__return__(Invitee,epmeet(T2,T1)),
*((T1=T2 | [**tell**(Invitee,confirm(T2)),*
__execute__(save(meeting(T2)))]);
*(T1\=T2| [**resume**(invite(Invitee,T2))]))])*

dialog*(reply(Host),[T,T1,T2],*
*[**return**(Host,epmeet(T1,T)),*
__call__(Host,epmeet(T2,T1)),
*((T1=T2 | [**ask**(Host,confirm(T2)),*
__execute__(save(meeting(T2)))]);
*(T1\=T2| [**resume**(reply(Host))]))])*

where "," and ";" are sequence (or conjunctive) and alternative (or disjunctive) operators , respectively. Variables start with capital letters, and variables that are local to a dialogue are listed before the messages. As it is quite apparent in this example, each dialogue consists of a *branching sequence* of messages i.e., a *sequence* with an *end alternative.* Similarly to lists, branching sequences can have an embedded structure. Unless they are resumed (with a *resume* message), dialogues are exited at the end of each embedded branching sequence. Actions interleaved with messages can be executed with an *execute* message. Although this simple example does not make use of this possibility (for a more complex example, see section 5), sub-dialogues can be entered (with an *enter* message) from any dialogue, such achieving *dynamic conversation* structures. The corresponding BNF syntax is given below in fig. 1

<dialog> ::= **dialog**(<dialogName>(<dialogParams>),<varList>,<branchSeq>)

<varList> ::= [] || [<varName>|<varList>]

<branchSeq> ::= [] || [<alt>] || <seq>

<alt> ::= <guardMes> || (<guardMes>;<alt>)

<seq> ::= [<mes>|<branchSeq>]

<guardMes> ::= (<guard>|<branchSeq>)

<mes> ::= <messageName>(<messageParams>)

<messageName> ::= **ask** || **tell** || **call** || **return** || **execute** || **enter** || **resume**

Fig. 1. BNF productions

As usual, "|" separates the head and tail of a list i.e., $[m_1|[m_2|\ldots[]]]=[m_1,m_2,\ldots]$. We also use "|" to isolate the guard in a guarded message. To avoid confusion, we use "||" as metasymbol for representing choices. We leave out the definitions for *names, parameters* and g*uards,* these being identifiers, first order terms and expressions, respectively. Branching sequences permit end alternatives, but do not allow for starting or middle alternatives i.e., cannot contain the list pattern *[<alt>|<branchSeq>]*.

4 Compiling dialogues into nd-plans

In order to define the operational semantics of our language, we shall first implement a multi-agent system under the form of an *abstract machine*. In this system, agents using the primitive messages defined in section 2 will communicate under the control of non-deterministic plans using a synchronization flag. These plans are equivalent to dialogues with an implicit synchronization. Reversibly, dialogues can be compiled back into plans and then executed on the sequential machine. As it is common, we shall not distinguish here (at least until the end of section 6) between the term *dialogue* and *conversation.*

4.1 An abstract machine for communicating agents with plans

Let us consider a multiagent system consisting of a *class* of identical agents. Similarly to classical *objects*, we shall distinguish the *class* itself, considered as an object of type "*agent class*", and its class *members* i.e., the objects of type "*agent instance*". The class itself will be used both as a repository for the common properties of its members (including the definition of the abstract machine), and as a blackboard for agent communication. We shall consider purely communicating agents i.e., the environment will be ignored, and thus there will be no agent *sensing*. We will also delay the dynamic embedding of plans until we introduce the compilation of dialogues. There will thus be no need yet to consider local variables, or to maintain stacks of activation records. The *local state* of a class of agents will be defined by a vector $l = [l^{Class}, l^1 \ldots l^n]$, where the components l^{Class} and l^i are the local state of the class and its members identified by an integer $i=1\ldots n$, respectively. We will make use of a predicate *agent* and assume that $l^{Class} \vdash agent(i)$ whenever agent *i* belongs to the class.

Messages exchanged between class members must use a data transport system. We shall abstract this transport system as follows: any message sent by an agent (this message being necessarily half of an exchange between a sender and a receiver, as introduced in the previous section, with the exception of the *resume* message to be used to reenter the same plan) will be first posted in the class. The class itself will then interpret the message's contents, wait for the second half of the exchange (thus achieving synchronization), and finally perform the computation on behalf of the receiver. Each message will be *blocking* until the exchange's completion i.e., no other exchange of the same type will be allowed between the sender and the receiver before the exchange is completed.

Let us assume that the language defining each l^i includes a set $P = \{p_1, p_2, \ldots\}$ of non-deterministic plan names (*nd-plan* in short) and four predicates *plan, priority, do* and *switch*. For each agent *i*, its current plan $p^i \in P$ refers to a set of implications "*conditions*" $\Rightarrow$ $do(p^i, a)$, where *a* is an action. In the case of communicating agents, actions will be identified with messages, and conditions will include a synchronization flag *sync* referring to the successful execution of the preceding message. As an example, let us consider the plans corresponding to the dialogues of section 3. In order for the first cycle in each plan to be started, we will assume that flags *sync(dialog(invite(Invitee,T)))* and *sync(dialog(reply(Host)))* have been raised beforehand. The corresponding host and invitee plans i.e., *invite(Invitee,T))* and *reply(Host))*, are defined as follows

sync(dialog(invite(Invitee, T)))$\Rightarrow$
*do(invite(Invitee,T),***call***(Invitee,epmeet(T1,T)))*

sync(call(Invitee,epmeet(T1,T))) $\Rightarrow$
*do(invite(Invitee,T),***return***(Invitee,epmeet(T2,T1)))*

sync(return(Invitee,epmeet(T2,T1)))$\wedge$ *T1=T2* $\Rightarrow$
*do(invite(Invitee,T),***tell***(Invitee,confirm(T2)))*

sync(return(Invitee,epmeet(T2,T1)))$\wedge \neg$*(T1=T2)* $\Rightarrow$
*do(invite(Invitee,T),***resume***(invite(Invitee,T2)))*

sync(dialog(reply(Host))) $\Rightarrow$

*do(reply(Host),**return**(Host,epmeet(T1,T)))*

sync(return(Host,epmeet(T1,T))) ⇒
*do(reply(Host),**call**(Host,epmeet(T2,T1)))*

sync(call(Host,epmeet(T2,T1)))∧ T1=T2 ⇒
*do(reply(Host),**ask**(Host,confirm(T2)))*

sync(call(Host,epmeet(T2,T1)))∧ ¬(T1=T2) ⇒
*do(reply(Host),**resume**(reply(Host)))*

Message *resume*, used by an agent to reenter a plan, is interpreted by a state transformer function τ^i defined as

$\tau^i([l^{Class},\ldots l^i,\ldots],$ ***resume(p)***$) = [l^{Class},\ldots l^i - \{plan(\_),sync(\_)\} \cup \{plan(p),sync(dialog(p))\},\ldots]$

Similarly, *processes* of priority *n* encompass implications "*conditions*" ⇒ *do(n, a)*. Let us further assume that each agent's initial nd-plan p_0^i and the class highest priority n_0 can be deduced from *l*, i.e. $l^i \vdash plan(p_0^i)$ and $l^{Class} \vdash priority(n_0)$. The abstract machine that defines the run of a class as a loop interleaving individual agent run cycles is then defined by the following procedure

procedure $run^{Class}(l)$
loop for all *i* ***such that*** $l^{Class} \vdash agent(i)$ ***do***
  if $l^i \vdash plan(p_0^i)$
  then $react^i(l,p_0^i)$;
if $l^{Class} \vdash priority(n_0)$
then $process^{Class}(l,n_0)$

In each cycle, initial plans p_0^i are activated by a procedure $react^i$. Synchronization occurs though a procedure $process^{Class}$. These procedures are defined as:

procedure $react^i(l,p^i)$
if $l^i \vdash do(p^i, a)$
then $l \leftarrow \tau^i(l,a)$
else if $l^i \vdash switch(p^i, p^{i\prime})$
  then $react^i(l,p^{i\prime})$

procedure $process^{Class}(l,n)$
if $l^{Class} \vdash do(n, a)$
then $l \leftarrow \tau^{Class}(l,a)$
else if $n > 0$
  then $process^{Class}(l,n-1)$

In each $react^i$ call, the agent's first priority is to carry out an action *a* from its current plan p^i. Otherwise, it may switch from p^i to $p^{i\prime}$, a recursive call to $react^i$ leading in turn to the same options. In any case, the next run cycle will again deduce and activate (possibly different) initial plans p_0^i. If the *switch* predicate defines directed acyclic graphs rooted at each possible p_0^i, corresponding thus to a hierarchy of plans with decreasing priorities and thus ensuring termination, then $react^i$ will always select the applicable nd-plan that has the highest *implicit* priority. This feature allows directing an agent to adopt a new plan whenever a certain condition occurs. It is however not required to implement purely communicative agents and the ***else*** branch of

react will thus be ignored in the sequel. Similarly, $process^{Class}$ will execute the process that has the highest *explicit* priority.

The state transformer function τ^s used to interpret message *tell(r, φ)* sent by *s* is

$\tau^s([l^{Class},\ldots l^s,\ldots],$ ***tell(r,φ)***$) =$
if $busy(tell(r,\varphi)) \notin l^s$
then $[l^{Class} \cup \{ack(s,tell(r,\varphi))\},\ldots\ l^s \cup \{busy(tell(r,\varphi))\},\ldots]$
else $[l^{Class},\ldots\ l^s,\ldots]$

The functions for messages *ask(s,ψ)*, *call(r,φ)* and *return(s,ψ)* are similarly defined. According to these functions, each message is thus first posted in the class, a *busy* flag is raised in the agent state, and the message waits to be synchronized. Synchronization occurs when two messages belonging to the same pair have been acknowledged. This synchronization is triggered by two *priority processes* defined as

$ack(s,tell(r,\varphi)) \wedge ack(r,ask(s,\psi))) \Rightarrow do(2,$ ***tellAsk(s,r,φ,ψ)***$)$
$ack(s,call(r,\varphi)) \wedge ack(r,return(s,\psi)) \Rightarrow do(1,$ ***callReturn(s,r,φ,ψ)***$)$

The state transformer function τ^{Class} achieving synchronization is:

$\tau^{Class}([l^{Class},\ldots l^s,\ldots l^r,\ldots],$***tellAsk(s,r,φ,ψ)***$) =$
if $l^r \cup \varphi \vdash \psi\theta$
then $[l^{Class} - \{ack(s,tell(r,\varphi)), ack(r,ask(s,\psi))\},\ldots$
$l^s - \{busy(tell(r,\varphi)), sync(\_)\} \cup \{sync(tell(r,\varphi))\},\ldots$
$l^r - \{busy(ask(s,\psi)), sync(\_)\} \cup \{sync(ask(s,\psi\theta))\},\ldots]$
else $[l^{Class},\ldots\ l^s,\ldots l^r,\ldots]$

$\tau^{Class}([l^{Class},\ldots l^s,\ldots l^r,\ldots],$***callReturn(s,r,φ,ψ)***$) =$
if $\varphi\theta = \psi\theta$ ***and*** $l^r \vdash \psi\theta$
then $[l^{Class} - \{ack(s,call(r,\varphi)), ack(r,return(s,\psi))\},\ldots$
$l^s - \{busy(call(r,\varphi)), sync(\_)\} \cup \{sync(call(r,\varphi\theta))\},\ldots$
$l^r - \{busy(return(s,\psi)), sync(\_)\} \cup \{sync(return(s,\psi\theta))\},\ldots]$
else $[l^{Class},\ldots\ l^s,\ldots l^r,\ldots]$

In short, all the flags are removed and a new *sync* flag carrying the computation results is raised. To ensure a simple execution scheme, a single such synchronization flag is used at any time.

4.2 Compiling dialogues

The concrete operational semantics for the complete language of fig. 1 are finally given below in fig. 2. This definition takes the form of compiling functions for translating dialogues into nd-plans. It closely follows the BNF syntax given above, with an *exit* message being automatically added at the end of each branching sequence. Each compiled message is assigned a unique *sequence* number. This number is used to define the sequencing flag *seq(D(I))* that will be raised when message with sequence number *I* from dialogue *D* is executed. This flag in turn will be used as a *sequencing condition* for the next message (recall that our abstract machine does not have a pro-

gram counter, and that execution is triggered by deduction). Sequencing conditions are required to serialize successive messages that may have the same synchronizing conditions and thus could otherwise not be distinguished (recall also that synchronization occurs when the two messages belonging to a primitive pair have been acknowledged: two distinct messages whose preceding pairs are identical will thus have the same synchronizing condition). As an alternative solution, the sequence number could be introduced in the synchronization flag.

As stated by the implication compiled by $comp_{mes}$, the condition for the execution of message *P(X)* is *var(Var)∧Sync∧seq(D(I))*, where *Sync* is its synchronizing condition, *seq(D(I))* its sequencing condition (with *I* referring to the preceding message), and *Var* is the list of local variables from the current dialogue. When this condition is

$comp_{dialog}(dialog(D,Var,\mathbf{BranchSeq})) = \{dialog(D,Var)\} \cup comp_{branchSeq}(D,Var,\mathbf{BranchSeq},sync(dialog(D)),0,0,N)$

$comp_{branchSeq}(D,Var,\mathbf{[]},Sync,I,J,J+1) = \{var(Var)\wedge Sync\wedge seq(D(I)) \Rightarrow do(D,save(seq(D(J+1)))) \wedge do(D,save(var(Var))) \wedge do(D,\mathbf{exit(D)})\}$

$comp_{branchSeq}(D,Var,\mathbf{[Alt]},Sync,I,J,N) = comp_{alt}(D,Var,\mathbf{Alt},Sync,I,J,N)$

$comp_{branchSeq}(D,Var,\mathbf{Seq},Sync,I,J,N) = comp_{seq}(D,Var,\mathbf{Seq},Sync,I,J,N)$

$comp_{alt}(D,Var,\mathbf{GuardMes},Sync,I,J,N) = comp_{guardMes}(D,Var,\mathbf{GuardMes},Sync,I,J,N)$

$comp_{alt}(D,Var,\mathbf{(GuardMes;Alt)},Sync,I,J,N) = comp_{guardMes}(D,Var,\mathbf{GuardMes},Sync,I,J,K) \cup comp_{alt}(D,Var,\mathbf{Alt},Sync,I,K,N)$

$comp_{seq}(D,Var,\mathbf{[Mes|BranchSeq]},Sync,I,J,N) = comp_{mes}(D,Var,\mathbf{Mes},Sync,I,J,K) \cup comp_{branchSeq}(D,Var,\mathbf{BranchSeq},sync(\mathbf{Mes}),K,K,N)$

$comp_{guardMes}(D,Var,\mathbf{(Guard|BranchSeq)},Sync,I,J,N) = comp_{branchSeq}(D,Var,\mathbf{BranchSeq},Sync\wedge\mathbf{Guard},I,J,N)$

$comp_{mes}(D,Var,\mathbf{P(X)},Sync,I,J,J+1) = \mathbf{if}\ P \in \{\mathbf{tell},\mathbf{ask},\mathbf{call},\mathbf{return},\mathbf{execute},\mathbf{enter},\mathbf{resume}\}\ \mathbf{then}\ \{var(Var)\wedge Sync\wedge seq(D(I)) \Rightarrow do(D,save(seq(D(J+1)))) \wedge do(D,save(var(Var))) \wedge do(D,\mathbf{P(X)})\}$

Fig. 2. Compiling functions

checked, the variables in the list *Var* will be unified with the corresponding variables in *Sync*. Before *P(X)* is actually executed, the sequencing condition for the next message will be updated into *seq(D(J+1))*, with *J+1* referring to the current message. The local variables will be similarly updated. As possible instantiations will be carried over from *Sync*, this will allow for the result of the previous message to be taken into account.

The last argument of each compiling function returns the last sequence number assigned by the function. Two input arguments i.e., *I* and *J*, provide the sequence numbers that are required to compile the sequencing conditions for the current and next messages. When compiling end alternatives in function $comp_{alt}$, *I* keeps its value while *J* is set to *K*, the current last sequence number. When compiling sequences in function

$comp_{seq}$, both *I* and *J* are set to *K* (globally, both *I* and *J* work similarly to the split second hand of chronograph i.e., they eventually fly back to *K*, but under different conditions).

When compiling end alternatives, function $comp_{alt}$ similarly keeps its synchronization flag *Sync*. In contrast, when compiling sequences *[Mes|BranchSeq]*, function $comp_{seq}$ introduces a new synchronization flag *sync(Mes)*. The exclusion of starting or middle alternatives in branching sequences precludes the compilation of complex synchronization flags of the form $sync(Sync_1) \vee sync(Sync_2) \vee \ldots$ that otherwise would propagate in parallel and then possibly lead to backtracking on execution. Similar remarks apply for sequencing flags i.e., like pure sequences, branching sequences can be serialized using a single sequencing flag. In other words, this means that messages may have at most one direct predecessor message. But, contrary to pure sequences, they may have multiple successors. A *simple sequential* execution scheme can thus be achieved by updating a single pair of *sync* and *seq* flags at each step. We have the following result, whose proof intuitively follows from the preceding remarks:

> ***Proposition*** Dialogues based on branching sequences can be *simply sequentially* executed (i.e. by using a single pair of synchronization and sequencing flags). Furthermore, if all the guards in a given alternative are mutually exclusive, then this execution will be *deterministic*.

Turning now to the interpretation of primitive messages (recall that *tell*, *ask*, *call*, and *return* have been defined earlier), we have the following new state transition functions, where a stack t^i is used to store agent's *i* current active dialogue embedding:

$\tau^i([\ldots<l^i,t^i>,\ldots],$ ***execute***$(a)) =$
if $\tau^i([\ldots<l^i,t^i>,\ldots], a) = [\ldots<l^{i\prime},t^i>,\ldots]$
then $[\ldots<l^{i\prime}\text{-}\{sync(\_)\}\cup\{sync(execute(a))\},t^i>,\ldots]$
else $[\ldots<l^i,t^i>,\ldots]$

$\tau^i([\ldots\langle l^i,t^i\rangle,\ldots],$ ***resume***$(q)) =$
if $dialog(q,v) \in l^i$
then $[\ldots\langle l^i\text{-}\{plan(\_),var(\_),seq(\_),sync(\_)\}$
$\cup\{plan(q),var(v),seq(q(0)),sync(dialog(q))\},t^i\rangle,\ldots]$
else $\{undefined\}$

$\tau^i([\ldots\langle l^i,t^i\rangle,\ldots],$ ***enter***$(q)) =$
if $dialog(q,v) \in l^i$
then ***if*** $\{plan(p),var(w),seq(p(s))\} \subset l^i$
then $[\ldots\langle l^i\text{-}\{plan(\_),var(\_),seq(\_),sync(\_)\}$
$\cup\{plan(q),var(v),seq(q(0)),sync(dialog(q))\},$
push$(t^i,\{p,w,s\})\rangle,\ldots]$
else $[\ldots\langle l^i\text{-}\{sync(\_)\}\cup\{plan(q),var(v),seq(q(0)),sync(dialog(q))\},t^i\rangle,\ldots]$
else $\{undefined\}$

$\tau^i([\ldots\langle l^i,t^i\rangle,\ldots],$ ***exit***$(q)) =$
if ***not empty***(t^i) ***and top***$(t^i) = \{p,w,s\}$
then $[\ldots\langle l^i\text{-}\{plan(\_),var(\_),seq(\_),sync(\_)\} \cup\{plan(p),var(w),seq(p(s)),sync(enter(q))\},$
pop$(t^i)\rangle,\ldots]$
else $[\ldots\langle l^i\text{-}\{plan(\_),var(\_),seq(\_),sync(\_)\},t^i\rangle,\ldots]$

Contrary to *entered* dialogues, *resumed* dialogues are not stacked. They can thus be used to implement *reentrant monitors* (see below in section 6).

5 Example: a multi-round negotiation

As an example of the complete language, let us consider the extension to *n* agents of the meeting-scheduling problem. This solution involves successive *rounds*, each round involving in turn successive *cycles*. The *host* dialogue initializes each *round*, directly followed by a general agreement or recursively by a new round. In each round, the host will *tell* in turn each of the invitees (again through recursion) to *reply* and then *invite* him for a negotiation *cycle*. Each such cycle will be initialized with the bilateral agreement just reached between the host and the previous invitee. The round itself will end up with a tentative proposal handed over to the main host dialogue.

In their *guest* dialogue, invitees will *ask* for instructions and then either *reply* and recursively *ask* for new instructions, or *ask* for the general agreement. In contrast to the solution involving only two agents, the agreement phase (i.e., *ask* for confirmation and then *save* the information) cannot directly follow a bilateral agreement, and thus is not included in the *invite/reply* but in the *host/guest* dialogues. The dialogues are:

```
dialog(host(Att,T), [T1],
       [enter(round(Att,T,T1)),
        ((T=T1 | [enter(tellAll(Att,turn(end))),
                  enter(tellAll(Att,confirm(T1))),
                  execute(save(meeting(T1)))]);
         (T\=T1|[enter(host(Att,T1))]))])

dialog(round([Att1|AttR],T,T2), [T1],
       [tell(Att1,turn(reply)),
        enter(invite(Att1,T,T1)),
        ((AttR=[]  | [execute(equals(T2,T1))]);
         (AttR\=[] | [enter(round(AttR,T1,T2))]))])

dialog(tellAll([Att1|AttR],C), [],
       [tell(Att1,C),
        ((AttR=[]  | []);
         (AttR\=[] | [enter(tellAll(AttR,C))]))])

dialog(invite(Invitee,T,T3), [T1,T2],
       [call(Invitee,epmeet(T1,T)),
        return(Invitee,epmeet(T2,T1)),
        ((T1=T2  | [execute(equals(T3,T2))]);
         (T1\=T2 | [enter(invite(Invitee,T2,T3))]))])

dialog(guest(Host),[Turn,T],
       [ask(Host,turn(Turn)),
        ((Turn=reply |[enter(reply(Host)),
                       enter(guest(Host))]);
         (Turn=end   | [ask(Host,confirm(T)),
                       execute(save(meeting(T)))]))])
```

```
dialog(reply(Host), [T,T1,T2],
       [return(Host,epmeet(T1,T)),
        call(Host,epmeet(T2,T1)),
        ((T1=T2  | []);
         (T1\=T2 | [enter(reply(Host))]))])
```

6 Conclusion and possible extensions

We proposed a formal language for modeling dynamic agent conversation. The corresponding operational semantics is given by compiling functions that maps dialogues onto non-deterministic plans executable on a sequential abstract machine. The underlying protocol for the exchange of information relies on a flag mechanism to ensure synchronization. One could object that this specification is too low level. We might well try and describe it in a more abstract way, for example by using a transition semantics as done in [5]. We would however be left short of true executable specifications. A number of useful extensions to the basic model are reviewed below.

6.1 Recovering from failures

A *simple failure* to answer (because the deduction involved in a communication primitive did not succeed) or a synchronization that did not occur (because the expected agent was not available, did not anticipate the request, or simply failed) are examples of deadlocks that could prevent dialogues to proceed as expected. We have already implemented a mechanism for *catching* a simple failure to answer. By default, this failure will propagate through embedded dialogues via a forced *exit.* It can be caught on demand in the first calling dialogue where it can be appropriately processed. Timeouts could be similarly handled. As an example, let us consider below the following extension for the *host* dialogue, in which a successful and an unsuccessful deduction lead to catch a status equal to *end* and *fail*, respectively.

```
dialog(host(Att,T), [T1,Status],
       [enter(round(Att,T,T1)|catch(Status)),
       ((Status=end |[((T=T1 | [enter(tellAll(Att,turn(quit))),
                                execute(save(meeting(T1)))]);
                       (T\=T1 |[enter(host(Att,T1))]))]);
        (Status=fail  |[enter(tellAll(Att,turn(quit))),
                        execute(save(failed(meeting)))]))])
```

6.2 Monitoring external commands

The overall behavior of any agent should allow for entering any dialogue on demand. This behavior could be defined as a reentrant dialogue monitoring external interrupts:

```
dialog(monitoring(I), [Act,P,X],
        [ask(I, command(Act(P(|X)))),
         ((Act=enter    |[enter(P(|X)),
                          resume(monitoring(I))]);
          (Act=execute|[execute(P(|X)),
                          resume(monitoring(I))]))])
```

Each agent *i* would have to be associated with a *sensing* procedure implemented as

procedure *sense^i(l)*
if *"the interrupt handler coupled with i receives the command Act(P(|X)))"*
then $l \leftarrow \tau^i(l,$ ***tell***$(i,$ *command(Act(P(|X)))))*

Calls to *sense^i* could then be interleaved with calls to *react^i* within each *run* cycle.

In this implementation, each agent *i* will thus first engage in a *tell(i,φ)/ask(i,ψ)* "auto exchange" with its sensing procedure. After retrieving a command i.e., after *ψ=φ=command(Act(P(|X)))*, it will then either *enter* a dialogue or *execute* an action, and then resume its monitoring task.

6.3 Engaging in multiple conversations

Agents should be allowed to engage in multiple conversations. Instead of providing a conversation language with a parallel (or concurrency) operator that could be used at the message level i.e., to interleave possible concurrent messages (as put forward in the languages *3APL* [5] and *ConGolog* [4], among others), we favor the simpler solution whereby each agent is a multithreaded entity interleaving concurrent conversations.

Just as a multi-agent system was implemented as a multi-threaded entity of agents using predicate *agent*, a multi-threaded agent can be implemented within an extended abstract machine using an additional predicate *conversation* as follows

procedure *run^{Class}(l)*
loop for all *i* ***such that*** *$l^{Class} \vdash$ agent(i)* ***do***
  sense^i(l);
  for all *j* ***such that*** *$l^i \vdash$ conversation(j)* ***do***
    if *$l^{ij} \vdash$ plan(p_0^{ij})*
      then *react^{ij}(l,p_0^{ij});*
      etc …

A new primitive message *concurrent* could then be used by any dialogue (such as the monitoring dialogue itself) to create a new conversation thread when required i.e., we would then have

dialog*(monitoring(I), [Act,P,X],*
  *[****ask****(I, command(Act(P(|X)))),*
  *((Act=enter |[****concurrent****(P(|X)),*
    resume*(monitoring(I))]);*
  etc …

In this new implementation, dialogues are considered as syntactic entities that can be attached to multiple conversations implemented as independent threads. To ensure consistency of this extended formalism, the monitoring dialogue itself must be attached to each agent's initial conversation thread.

7 Related work

The subject of modeling agent conversation is relatively new. Apart form the work already mentioned in the introduction i.e., [3] [6] and [7], which concentrates on defining graphical frameworks for representing static conversation patterns, earlier contributions include [2] and [8]. None of this work however seems to address the issue of modeling dynamic conversation structures. Furthermore, as the underlying communication models are either left unspecified or made to rely on speech acts, there is no simple ways to define the corresponding operational semantics leading to directly executable specifications. Finally, if conversations are to be used for modeling the *social ability* of agents, then (as already argued by Hendricks and al [5]) computational equivalents for speech acts should not be included in an agent communication language, as done in KQML or FIPA ACL. Communications primitives should instead be kept neutral, and *mental attitudes* should be allowed to *emerge* eventually as intelligent behavior.

References

1. P. Bonzon, An Abstract Machine for Communicating Agents Based on Deduction, in: J.-J. Meyer & M.Tambe (eds), *Intelligent Agents VIII*, LNAI vol. 2333, Springer Verlag (2002)
2. R. Elio, A. Haddadi and A. Singh, Task Models, Intentions and Agent Conversation Policies, in: *Proc. PRICAI-2000*, LNAI vol. 1886, Springer Verlag (2000)
3. FIPA Specifications, available on line at http://www.fipa.org
4. G. de Giacomo, Y.Lespérance and H. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence*, vol. 121 (2000)
5. K.V. Hendricks, F.S. de Boer, W.van der Hoek and J.-J. Meyer, Semantics of Communicating Agents Based on Deduction and Abduction, in: F.Dignum & M.Greaves (eds), *Issues in Agent Communication*, LNAI vol. 1916, Springer Verlag (2000)
6. M. Nowostawski, M. Purvis and S. Cranefield, A Layered Approach for Modelling Agent Conversations, in: E T. Wagner and O.F. Rana (eds), *Proc. 2nd Int. Workshop on Infrastructure for Agents, MAS, and Scalable MAS, 5th Int. Conf. on Autonomous Agents*, Montreal (2001)
7. J.J. Odell, H.V.D. Parunak and B. Bauer, Representing Agent Interaction Protocols in UML, in: P. Ciancarini and M. Wooldridge (eds), *Agent-Oriented Software Engineering*, Springer Verlag (2001)
8. R. Scott, Y. Chen, T. Finin, Y. Labrou and Y. Peng, Modeling Agent Conversations with Colored Petri Nets, in: *Working Notes of the Worshop on Specifying and Implementing Conversation Policies, 3rd Int. Conf. On Autonomous Agents*, Seattle (1999)
9. M. Wooldridge and A. Lomuscio, Reasoning about Visibility, Perception and Knowledge, in: N.R. Jennings and Y. Lespérance (eds*), Intelligent Agents VI*, LNAI vol. 1757 , Springer Verlag (2000)