An Abstract Machine for Classes of Communicating Agents Based on Deduction·

Pierre Bonzon

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

Abstract. We consider the problem of defining executable runs for classes of communicating agents. We first define an abstract machine that generates runs for individual agents with non-deterministic plans. We then introduce agent classes whose communication primitives are based on deduction. Contrary to other more theoretical work, their operational semantics are given by an abstract machine that is defined purely in sequential terms. This machine readily offers straightforward opportunities for implementing and experimenting prototypes of collaborative agents.

1 Introduction

According to the *theory of knowledge* [4], communication can be viewed as the act of upgrading the state of knowledge in a multi-agent system. At the low end of the spectrum is *distributed knowledge.* This situation arises when the deduction of some fact by a single agent requires information that is disseminated among the other agents. At the high end, *common knowledge* implies publicity, i.e. full reciprocal awareness of some fact by all agents. A system's performance obviously depends on its state of knowledge. Accordingly, in many applications, the focus is on trying to upgrade the system's state of knowledge through communication. Towards this end, an approach is to rely on an external model of knowledge based on possible world semantics. But in this solution there is "no notion of the agents computing their knowledge and no requirements that the agents be able to answer questions based on their knowledge"[4]. Agents should however be able to compute, and not just communicate their knowledge.

Turning away from the theoretical approach just outlined, practical agent communication models (such as those advocated by KQML and ACL of FIPA) are generally based on *speech act theory* [11]. They thus rely on the mental attitudes of agents. Ideally, these communication models should be coupled with comprehensive core agent models enjoying a minimum "understanding" for these various attitudes. Unfortunately, the current agent models that can be used as background theory are not so expressive. They generally lack many of the required "mind components" (e.g., those corresponding to such actions as making an offer, a promise, a request, etc.).

· Revised version from *Intelligent Agent VIII*, LNAI vol. 2333, Springer Verlag 2002

Most current multi-agent models therefore integrate poorly expressive core agent models with inadequate, overly expressive communication models. As a result, many proposed communicative actions are difficult (if not impossible) to match with the agent's semantics. While balanced integration should be sought, we do not know of any attempt to establish a formal correspondence between subsets of KQML or ACL, on one hand, and an agent model comprising at least *beliefs*, *desires* and *intentions*, such as *AgentSpeak(L)* [10]) or any other instance of the BDI model, on the other hand. Current literature on rational agents [12] does not even mention the problem.

Recently, a completely new approach has been advocated by Hindricks and al. [6]. Their logical communicative primitives do not correspond to any speech act in particular. They are defined as simple and "neutral" actions enjoying a well-defined and clear semantics that can be used for many different purposes, including the implementation of speech acts. Hindriks and al. further argue that speech act theory should not be viewed as a repository for all kinds of different communicative acts. Computational equivalents for speech acts do not necessarily have to be included in an agent communicative language, as done in KQML or ACL. Speech act theory may instead provide a set of abstract descriptions of communicative actions. These should be used then to specify and verify the behaviours of agents. It would then be up to the programmer to satisfy this specification using basic communicative actions.

An important feature in this new approach is the use of synchronised pairs: in order for two agents to communicate, both parties must first agree to an exchange (e.g., by independently using protocols based on these synchronised pairs). They will then wait until the exchange is completed before proceeding with their remaining activities. As an example (that will be developed later), this can be used in collaborative models to synchronise successive negotiation rounds as well as the successive steps involved in each round.

We shall follow and further simplify this logical approach. In short, Hindriks and al. consider the exchange of two messages between a *sender* and a *receiver* as a way for the receiver, either to use data provided by the sender to answer a query of his own, or to use his data to answer a query from the sender. These two types of messages are represented by two pairs i.e., *tell/ask* and *req/offer* respectively. In both cases, no data is sent back to the sender, and the formal semantics captures the processing done by the receiver (roles however may be switched, as we shall illustrate). This processing requires either a *deductive* (for *tell/ask*) or *abductive* (for *req/offer*) reasoning based on the receiver local state.

Deduction is a well understood task for which semi-decidable procedures can be easily implemented (e.g. under the form of a meta-interpreter within a logic programming framework). Abduction is a much more complex and difficult process. In order to define and implement *executable* runs involving deductions only, we shall give up the pair *req/offer* and define instead a simplified *call/return* pair that, similarly to *tell/ask*, relies on deduction only. When used in combination with the *tell/ask* pair, this new simplified pair will lack the full power of abduction. It still allows for the implementation of various communication protocols.

As logical communicative primitives do not involve mental attitudes, the corresponding core agent model can be kept simple. In the 3APL language of Hindriks and al., individual agents are multi-threaded entities consisting of *beliefs* and *goals*. This means that an agent may have multiple goals that are executed in parallel.

A multi-agent system itself is again a multi-threaded system of multi-threaded entities. The corresponding operational semantics rely on concurrent programming concepts that are left implicit and thus achieve a seamless integration of the communication and the core agent models. But unless we first implement a true concurrent programming language that in addition offers all the required functionality (including deductive reasoning capabilities), we are left short of executable specifications.

We believe however that is both possible and worthwhile to try and get executable specifications by defining the complete model in purely sequential terms. The first issue we face is the choice of a core agent model. As already mentioned above, logical communication primitives do not require mental attitudes. One therefore does not need to distinguish between *goals* and *beliefs*.

In order to plan an agent's actions, at least two competing approaches are possible i.e., *static* planning and *reactive* (or dynamic) planning. While static planning involves explicit goals and means-end analysis, reactive planning is based on conditions-action rules without explicit goals. The logical specification of an agent's primitive actions required by static planning is an unrealistic prerequisite. Agents are not likely to reason about the effect of their actions on the environment. Furthermore, they will generally not react to environmental changes by designing complex plans from scratch. We therefore favour the reactive approach. In this framework, agents select rules from sets of *predefined* plans. As agents must be ready to reconsider their choices at any time, the issue is to enforce timely reactions leading to an appropriate *change* of plans.

Towards this end, we propose to incorporate the concept of plan within an existing model of reactive agents. Following Wooldridge & Lomuscio [13], a general model of agent with sensing can be given in abstract functional terms. In order to get concrete executable specifications, we shall first develop these functional definitions into a set of procedures. These procedures will represent an abstract machine generating runs for individual non-deterministic agents. The concept of plan is introduced next. Given by *logical implications* similar to conditions-action rules, an agent's set of predefined plans can be looked at as a logical agent's program. As in other logical agent models, the agent must first deduce the action it intends to take. But in contrast to other logical agent models e.g., such as Golog and/or ConGolog that are based on the situation calculus [2] [8], our agents deduce only one action at a time**.** We believe that this framework offers a valuable alternative to the approach just mentioned, especially if reactivity and communication are at stake. To substantiate this claim, we shall use our approach to implement our model of communicating agents based on deduction.

We shall not concern ourselves with the corresponding declarative semantics, and will be content with the operational semantics defined by the abstract machine. The benefits that follow from this approach are:

- extensions can be built on top of this machine: as an example, we will define and implement a model of multi-agent system as interleaving of individual agent runs
- the synchronisation operations for integrating this core system with the communication part can be made explicit and put under the agent's control

– using a step-wise refinement approach, this abstract machine may be implemented on any platform and in any particular programming environment: this is illustrated in the appendix outlining a Prolog implementation.

In summary, the extension of an agent model with sensing to include non-deterministic plans, the reduction of communication primitives to deductive reasoning, and their integration within a concrete multi-agent system are the main contributions of this paper.

The rest of this paper is organised as follows: in section 2, we reproduce the functional definition of an agent's run with sensing. Section 3 proposes a corresponding concrete model. Section 4 introduces agents with plans. Section 5 defines agent classes whose communication is based on deduction.

2 Abstract functional definitions

Following Wooldridge & Lomuscio [13], an environment *Env* is a tuple $\langle E, vis, \tau_e, e_0 \rangle$, where

- $E=\{e_1, e_2, \ldots\}$ is a set of *states* for the environment
- $vis: E \rightarrow 2^E$ is a *visibility* function
- $\tau_e : E \times Act \rightarrow E$ is a *state transformer* function for the environment, with *Act* a set of *actions*
- $e_0 \in E$ is the *initial state* of the environment

and an agent *Ag* is a tuple $\langle L, Act, see, do, \tau_a, l_0 \rangle$, where

- $L=\{l_1, l_2, \ldots\}$ is a set of *local states* for the agent
- $Act=\{a_1, a_2, \ldots\}$ is a set of *actions*
- $see: vis(E) \rightarrow Perc$ is the *perception* function mapping *visibility sets* to *percepts*,
- $\tau_a : L \times Perc \rightarrow L$ is the *state transformer* function for the agent
- $do: L \rightarrow Act$ is the *action selection* function, mapping agent local states to *actions*
- $l_0 \in L$ is the *initial state* for the agent.

An *agent system* is a pair *{Ag, Env}* whose set of *global states G* is any subset of $L \times E$ i.e., $g_i = \langle l_i , e_i \rangle$. A *run* of a agent system is a (possibly infinite) sequence of global states $(g_1, g_2, \ldots)$ over *G* such that

- $\forall i,\ g_i = \langle \tau_a(l_{i-1}, see(vis(e_{i-1}))), \tau_e(e_{i-1}, do(l_i)) \rangle$

3 A concrete model of non-deterministic agents with sensing

Let *S* be the set of sentences of first order logic with arithmetic whose set of predicates includes the predicate *do/1*, and let $L = 2^S$ and *Perc=S*. If we incorporate the selection of actions and the mapping of visibility sets within the functions τ_e and τ_a, then we get two new functions $\tau_{e,do} : E \times L \rightarrow E$ and $\tau_{a,see,vis} : L \times E \rightarrow L$. Equivalently, these new functions can be seen as procedures with side effects i.e.,

$\tau_{a,see,vis} : L \times E \rightarrow L \ \Rightarrow$ ***procedure*** *sense(l,e)* with side effects on *l*

$\tau_{e,do} \quad : E \times L \rightarrow E \quad \Rightarrow$ ***procedure*** *react(e,l)* with side effects on *e*

We can define these procedures as follows

procedure *sense(l,e)*
if *"the environment produces percept p"*
then $l \leftarrow \tau_a(l,p)$

procedure *react(e,l)*
if $l \vdash do(a)$
then $e \leftarrow \tau_e(e,a)$

We write $l \vdash do(a)$ to mean that the formula *do(a)* can be proved from the formula *l*, meaning in turn that *a* is an applicable action. An agent's run is defined by

procedure *run(e,l)*
loop *sense(l,e);*
react(e,l)

Although environments are supposed to evolve deterministically, the choice to be made among possible actions, i.e. among all a_i such that $l \vdash do(a_i)$, is left unspecified. Consequently, the *run* procedure can be seen as a *non-deterministic* abstract machine generating runs for logical agents. In short, it is a concrete model of non-deterministic agents.

4 Non-deterministic agents with plans

Intuitively, an agent's plan can be described as an ordered set of actions that may be taken, in a given state, in order to meet a certain objective. As above, agents whose choice among possible plans (i.e. those that are applicable in a given state) is left unspecified are non-deterministic. Let us assume that the set of constant symbols and predicates of *S* include a set $P = \{p_1, p_2, \ldots\}$ of non-deterministic plan names (*nd-plan* in short) and three predicates *plan/1, do/2* and *switch/2*. Let us further extend the definition of an agent's global state to include its current active nd-plan *p*. We finally have the following new procedures:

procedure *react(e,l,p)*
if $l \vdash do(p, a)$
then $e \leftarrow \tau_e(e,a)$
else if $l \vdash switch(p, p')$
then *react(e,l,p')*

procedure *run(e,l)*
loop *sense(l,e);*
if $l \vdash plan(p_0)$
then *react(e,l,*p_0*)*

In each *react* call, the agent's first priority is to carry out an action *a* from its current plan *p*. Otherwise, it may switch from *p* to *p'*. When adopting a new plan, a

recursive call to *react* leads in turn to the same options. In *run*, the initial plan p_0 is chosen in each cycle. If the environment has not changed in between, then the agent is bound to adopt its last active plan again. On the other hand, if the environment has changed, and if the embedding of plans reflects a hierarchy of priorities (i.e., the structure that can be associated with the *switch/2* predicate is that of directed *acyclic* graphs rooted at each initial plan) then it will select a plan that has the *highest implicit priority*.

This achieves a simple way to instruct an agent to adopt a new plan whenever a certain condition occurs, without having to tell it explicitly when to switch from its current plan. The corresponding switching logic is thus easier to define than if the last active plan was kept at each cycle: in this case, the *run* procedure would involve less overhead, but explicit switching conditions should be given for each plan an agent may switch to from its current plan (i.e., the structure associated with the *switch/2* predicate would be that of directed *cyclic* graphs).

Example

Let us consider the mail delivery robot of Lespérance & al. [7]. Let *start* be its single initial plan. Its first priority is to handle a new order. It will then unconditionally switch to plan *check* to see if any order has to be cancelled. Depending on his current state, it will then switch to either plan *move* (and go on moving without conditions) or to plan *motionControl* (and search for a new customer). This second plan will lead in turn to switch to either *tryServe* or *tryToWrapUp*. Should a new order or a cancellation arise while it is attending other business, it will then automatically switch to the corresponding plan even though there are no explicit switching conditions for doing so. This set of plans is given by the following implications and facts:

$$orderState(N,justIn) \Rightarrow do(start,handleNewOrder(N))$$

$$switch(start,check)$$

$$orderState(N,toPickUp) \wedge sender(N,Sender) \wedge suspended(Sender) \Rightarrow do(check,cancelOrder(N))$$

$$robotState(moving) \Rightarrow switch(check,move)$$

$$do(move,noOp(moving))$$

$$\neg robotState(moving) \wedge (orderState(N,toPickUp) \vee orderState(N,onBoard) \vee \neg robotPlace(centralOffice)) \Rightarrow switch(check,motionControl)$$

$$\neg searchedCustomer \Rightarrow do(motionControl,searchCustomer),$$

$$customerToServe(Customer) \Rightarrow switch(motionControl,tryServe(Customer)),$$

$$\neg customerToServe(\_) \Rightarrow switch(motionControl,tryToWrapUp),$$

$$robotState(idle) \Rightarrow do(tryServe(Customer),startGoto(mailbox(Customer)))$$

$$robotState(stuck) \Rightarrow do(tryServe(Customer),resetRobot)$$

$$robotState(reached) \Rightarrow do(tryServe(Customer),freezeRobot)$$

$$robotState(frozen) \Rightarrow do(tryServe(Customer),dropOffShipmentsTo(Customer)) \wedge do(tryServe(Customer),pickUpShipmentsFrom(Customer)) \wedge$$

$do(tryServe(Customer), resetRobot)$

$robotState(idle) \Rightarrow do(tryToWrapUp,startGoto(centralOffice))$

etc…

A particular case: priority processes

When nd-plans form equivalence classes that can be linearly ordered, these classes can be identified with plans of equal *priority*. If priorities are represented by positive integers *n*, then we get a new reaction scheme without explicit switching conditions, where plans define *priority processes 1,2,..,n* . This leads the new procedure

procedure *process(e,l,n)*
if $l \vdash do(n, a)$
then $e \leftarrow \tau_e(e,a)$
else if $n > 0$
then *process(e,l,n-1)*

If procedure *process* is called repeatedly with the highest priority, then the execution of a process *n* will proceed unless the conditions for a process with a higher priority become satisfied. Since we have $l \vdash do(n, a)$, it can be assumed that once a process *n* is selected, then at least one of its action will be executed. Priority processes and plans can be interleaved in many ways. As an example let us consider the following run^+ procedure relying on a new predicate *priority/1* delivering the *current highest priority* n_0:

procedure $run^+(e,l)$
loop *sense(l,e);*
if $l \vdash priority(n_0)$
then *process* (e,l,n_0)*;*
if $l \vdash plan(p_0)$
then $react(e,l,p_0)$

and let use it to implement the mail delivery robot in a way that is very similar to the solution given by Lespérance & al. using ConGolog [7]. The top plans (up to but not including plans *tryServe* and *tryToWrapUp*) can be expressed as priority processes:

$orderState(N,justIn) \Rightarrow do(3,handleNewOrder(N))$

$orderState(N,toPickUp) \wedge sender(N,Sender) \wedge suspended(Sender) \Rightarrow do(2,cancelOrder(N))$

$\neg robotState(moving) \wedge (orderState(N,toPickUp) \vee orderState(N,onBoard) \vee \neg robotPlace(centralOffice)) \Rightarrow do(2,robotMotionControl)$

$robotState(moving) \Rightarrow do(1,noOp)$

In order to activate one the remaining plans, the *robotMotionControl* action must allow for the assertion, within *l*, of either *plan(tryServe(Customer))* or *plan(tryToWrapUp)*. A comparison with the ConGolog solution reveals that:

- in our solution, the entire control mechanism is given in terms of nd-plans and/or processes, whose execution steps are *explicitly* interleaved with sensing: as a

result, autonomous agents whose independent, asynchronous actions must be co-ordinated and/or synchronised may be easily implemented

- in the ConGolog solution, the *mainControl* procedure concurrently executes four ConGolog interrupts that corresponds to our four *priority processes*, but robot motion control relies on sequential procedures that run asynchronously with the rest of the architecture; the environment is thus *implicitly* monitored.

5 Communicating agents based on deduction

As an example of autonomous agents whose independent actions must be synchronised, let us now define and implement a model of communicating agents. As indicated in the introduction, we shall use and simplify the proposal made by Hindriks and al. [6]. Following a purely logical approach, they introduce 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*. In the first exchange, 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 his own query ψ using any data sent by *s*. Both messages are sent asynchronously, 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 a handshake or synchronisation, 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 *deductively* compute the most general substitution θ such that

$$l^r \cup \varphi \vdash \psi\theta .$$

According to Hindriks & 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, 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 his own data ψ for solving any query submitted by *s*. When put together, these two messages will allow the receiver *r* to find the possible instantiations of his free variables in ψ that allow

him to deduce φ. Formally, receiver *r* will *abductively* compute the most general substitution θ such that

$l^r \cup \psi\theta \vdash \varphi$.

According to Hindriks & 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 results, a subsequent reversed exchange (e.g. *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 the receiver's computations), we feel that the sender who, in the second case, expresses a need for data should automatically benefit from the receiver's computations. Furthermore, as abductions are difficult to achieve and implement, we favour exchanges that do not rely on abduction. Giving up the *req/offer* pair, we shall thus define and implement instead a simplified *call/return* pair that relies on deduction only. By doing so, we will end up with a less powerful model. It is interesting to note however that all *req/offer* examples given b Hindriks & al. 99 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 will also be illustrated at the end of this section).

In the new *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 to return the instantiations that hold in his own local state. Formally, receiver *r* will *deductively* compute the substitutions θ such that

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

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.

A concrete model and its implementation

The core model we shall adopt consists of *classes* of identical agents, as defined in section 4. Similarly to classical *object* theory 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 and as a *blackboard* for agent communication.

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 as defined above) 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 synchronisation), and finally perform the computation on behalf of the receiver. As an assumption, 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.

In order to further simplify our presentation, we shall consider purely communicating agents i.e., agents that do not carry out any action other than the exchange of messages. The environment per se will thus be ignored. Formally, the *local state* of a class of agents seen as a whole i.e., including its members, 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 itself and of its members identified by an integer $i=1\ldots n$, respectively. We will use a new predicate *agent/1* and assume that $l^{Class} \vdash agent(i)$ whenever agent *i* belongs to the class.

As communicative actions do not affect the environment, the state transformer function $\tau_e: E \times Act \rightarrow E$ should be replaced by a function $\tau_a: L \times Act \rightarrow L$. Actually, in order to take into account the originator of a communicative action, we shall consider a set of such transformer functions, each function being associated with a given class *Class* or member *i*. We will thus consider the function $\tau^{Class}: L \times Act^{Class} \rightarrow L$ to be used in procedure $process^{Class}$, on one hand, and the functions $\tau^i : L \times Act^i \rightarrow L$, $i=1, \ldots, n$, to be used in procedure $react^i$, on the other.

The abstract machine that defines the run of a class of agents as *interleavings* of individual runs 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 this particular definition, messages are first processed (via the calls to $react^i$) and then possibly synchronised without delay (via the subsequent call to $process^{Class}$). In this framework, the $react^i$ and $process^{Class}$ procedures are defined as follows:

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 $process^{Class}(l,n\text{-}1)$

The state transformer function τ^s required to process the message $tell(r,\varphi)$ is:

$\tau^s([l^{Class},\ldots l^s,\ldots], tell(r,\varphi)) =$ **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 $ask(s,\psi)$, $call(r,\varphi)$ and $return(s,\psi)$ are similarly defined. Each message is thus first "acknowledged" by the class, a blocking flag (i.e., *busy*) is raised, and the message waits to be synchronised. Synchronisation occurs when two messages belonging to the same pair have been acknowledged. This synchronisation is triggered by two *priority processes* defined as:

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

The state transformer function τ^{Class} achieving synchronisation by the class is:

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

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

Old flags are removed and a new *sync* flag carrying the computation result is raised. To ensure simple sequential execution, a single such *synchronisation* flag is available at any time for each agent. Thus there will be no trace of successive exchanges, and agent's nd-plans must be designed to use this flag accordingly.

Example: two-agent meeting scheduling

In this simplified version of the *two-agent scheduling* example of Hindriks and al. [6], one agent is designated as the *host* and the other one as the *invitee*. Both agents have free time slots to meet e.g.,

$l^{host} \vdash meet(13) \wedge meet(15) \wedge meet(17) \wedge meet(18)$
$l^{invitee} \vdash meet(14) \wedge meet(16) \wedge meet(17) \wedge meet(18)$

and they must find their earliest common free slot (in this case, *17*). The host has the responsibility of starting each *round* of negotiation with a given lower time bound *T*. A round of negotiation comprises three *steps*, each step involving in turn an exchange of messages.

In the first step (corresponding to the first line of both *invite* and *reply*), the host initialises a *call/return* exchange, calling on the invitee to find out his earliest free spot *T1* after *T*. In the second step (corresponding to the second line), the roles are swapped: the invitee initialises a *call/return* calling on the host to find out his earliest free spot *T2* after *T1*. In the final step the host either confirms an agreement on time *T2* (if *T1=T2*) by initialising a *tell/ask* exchange, or starts a new round with *T2*.

The corresponding host and invitee plans i.e., *invite(Invitee,T)* and *reply(Host)*, are

sync(dialog(invite(Invitee,T)))
⇒ *do(invite(Invitee,T),call(Invitee,epmeet(T1,T)))*

sync(call(Invitee,epmeet(T1,T)))
⇒ *do(invite(Invitee,T),return(Invitee,epmeet(T2,T1)))*

sync(return(Invitee,epmeet(T2,T1)))∧T1=T2
⇒ *do(invite(Invitee,T),tell(Invitee,confirm(T2)))*

sync(return(Invitee,epmeet(T2,T1)))∧ ¬(T1=T2)
⇒ *do(invite(Invitee,T),resume(invite(Invitee,T2)))*

sync(dialog(reply(Host)))
⇒ *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)))*

where the flags *sync(dialog(invite(Invitee,T)))* and *sync(dialog(reply(Host)))* are used to initialise the exchange. Message *resume* (used by an agent to restart a plan) and predicate *epmeet(T1,T)* meaning "*T1 is the earliest possible meeting time after T* " are defined as

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

$meet(T1) \wedge (T1>=T) \wedge \neg(meet(T0) \wedge (T0>=T) \wedge (T0<T1)) \Rightarrow epmeet(T1,T).$

In comparison, Hindriks and al. alternate exchanges *req(Invitee, ∃ T1 epmeet(T1,T)) / offer(Host,meet(T2))* and *offer(Invitee,meet(T4)) / req(Host, ∃ T3 epmeet(T3,T2))* that involve abductive tasks. As ground instances (i.e., *meet(13)*, *meet(14)*, etc.) of the

receivers offers are available in their respective local states, we can alternate instead *call/return* and *return/call* exchanges leading to the same result through simpler deductive tasks. As discussed in the introduction, 3APL synchronisation operations are left implicit. Therefore, the two 3APL "practical reasoning rules" that correspond to our *invite(Invitee,T)* and *reply(Host)* plans do not require any flag. They are however not directly executable by a sequential machine, whereas nd-plans are (see the appendix for the outline of a Prolog implementation). In our further work [1], plans are rewritten as *dialogs* with an implicit synchronisation. As these dialogs can be compiled back into nd-plans, they also represent executable specifications.

6 Related work

A popular choice to specify executable agent models is to rely on the *situation calculus*. This extension of the first order calculus was designed to allow reasoning about the effect of actions. In the *Golog* system [7], which is based on this approach, an interpreter verifies if predefined programs are applicable to a given goal. If successful, it then delivers the execution trace corresponding to a sequence of actions that will fulfil this goal. The result is a situation that is considered a final state in a system of state transitions. *ConGolog* [2] represents a development of *Golog* in the direction of concurrent agent programming. In order to allow plans "to be suspended or terminated and new plans devised to deal with exceptional event or condition"[2], this extension introduces concurrent processes with priorities as well as interrupts. If an interrupt gets control from a higher priority process, this interrupt may trigger and its body is then executed (possibly repeatedly i.e., as long as its guard is satisfied). This computational model corresponds in many ways to our runs based either on nd-plans or priority processes. In particular, a *ConGolog* interrupt $<\varphi \rightarrow \sigma>$ associated with a process of priority *n* could be represented in our framework by an implication $\varphi \Rightarrow do(n,\sigma)$. However as ConGolog processes need not be linearly ordered, it may not be obvious how to assign them an explicit priority *n.* Since the basic computational mechanism of Golog/ConGolog is embedded in logic, it is possible to use a model of the action theory to assign semantics to programs. Being essentially at a meta-level, our approach does not allow this.

ConGolog original specifications as an offline interpreter did do not provide facilities for either agent sensing or agent communications, whereas our approach does. Our own framework being an online interpreter based on a reactive agent model, the comparison may thus be misleading in this respect. Recent proposals [3] to allow agents with sensing seem however to close the gap between these two approaches, as the on-line execution model of [3] no longer requires searching for a final state.

We have already sketched in the introduction alternative ways to model communication. We also indicated why we turned to the approach of Hindriks and al. We refer to them for a thorough discussion of the relationship between their proposal and current agent communication models.

7 Conclusion and future work

Most current communication models are overly expressive with respect to the available agent models that can be used as background theory. As a result, many proposed communicative actions are difficult (if not impossible) to match with a given agent's core semantics. Communicating agents based on simple deductive and/ or abductive exchanges, as introduced by Hindriks and al. [6], achieve one of a few existing *balanced integration* we know of. These exchanges were further simplified in section 5 in order to rely on deduction only, and will apply in cases where the receiver's local state includes closed forms of his offers. Contrary to other more theoretical work, based for example on the π-calculus [5],[9], their operational semantics were given here by an abstract machine that is defined purely in sequential terms. It thus readily offers straightforward opportunities for implementing prototypes of collaborative agents. As an example, we have run a solution of the *n-agent* meeting problem (also discussed in [6]). This solution is given under the form of *dialogs* that can be *simply sequentially* executed by updating the single current synchronisation flag for each agent at each step. In order to achieve this result, dialogs expressed in a higher level language with implicit synchronisation must be first compiled into nd-plans.

At the same time, this framework will accommodate the extensions and/or refinements needed to develop full-fledged agents interfaced with real communication software.

Acknowledgement

The author wishes to thank the anonymous referees for their valuable comments. Indeed most of these comments found their way into this revised version.

References

1. P. Bonzon, Compiling Agent Dialogs for Simple Sequential Execution, submitted
2. G. de Giacomo, Y.Lespérance and H. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence*, vol. 121 (2000)
3. G. de Giacomo and H. Levesque, An Incremental Interpreter for High-Level Programs with Sensing, in: H. Levesque & F. Pirri (eds), *Logical Foundations for Cognitive Agents*, Springer (2000)
4. R. Fagin, J. Halpern, Y Moses & M. Vardi, *Reasoning About Knowledge*, MIT Press (1995)
5. Ferber & O. Gutknecht, Operational Semantics of Multi-agent Organizations, in: N.R. Jennings and Y. Lespérance (eds*), Intelligent Agents VI,* LNAI, vol. 1757 , Springer (2000)
6. K.V. Hindricks, F.S. de Boer, W.van der Hoek and J.-J. Meyer, Semantics of Communicating Agents Based on Deduction and Abduction, *Proceedings IJCAI99 Workshop on ACL* (1999)
7. Y. Lespérance, K. Tam and M. Jenkin, Reactivity in a Logic-Based Robot Programming Framework, in: N.R. Jennings and Y. Lespérance (eds*), Intelligent Agents VI,* LNAI, vol. 1757, Springer (2000)
8. H.J.Levesque, R.Reiter, Y.Lespérance, F.Lin & R.Scherl, GOLOG: A Logic Programming Language for Dynamic Domains, *Journal of Logic Programming*, vol. 31 (1997)

9. R. Milner, *Communicating and Mobile Systems*; the π-Calculus, Cambridge Univ. Press (1999)
10. A.S. Rao, AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language, in: W. Van der Velde and J.W. Perram (eds.), *Agents Breaking Away* (MAAMAW '96), LNAI vol. 1038, Springer (1996)
11. R. Searle, *Speech Acts*, Cambridge University Press (1969)
12. M. Wooldridge, *Reasoning about Rational Agents*, MIT Press (2000)
13. M. Wooldridge & A. Lomuscdio, Reasoning about Visibility, Perception and Knowledge, in: N.R. Jennings and Y. Lespérance (eds*), Intelligent Agents VI,* LNAI ,vol. 1757, Springer (2000)

Appendix: towards a Prolog implementation

Both an agent class and its members are represented by *objects* identified as *Class* and *Class(I)*, respectively. The formulas *P* contained in an object are asserted as unit clauses *instance(Object,P)*. Operations on objects are defined by the primitive procedures

```
new(Object)              :- retractall(instance(Object,_)).
insert(Object,P)         :- assert(instance(Object,P)).
remove(Object,P)         :- retractall(instance(Object,P)).
insertList(Object,L)     :- forall((L:List,member(P,List)),
                                insert(Object,P)).
```

As individual agents inherit the properties of their class, each agent's local *state* encompasses both the *private* formulas that are contained in the agent itself, as well as the *public* formulas that are contained in its class. The *state(Object,P)* predicate meaning "*formula P is contained in the local state of Object*" is then defined as

```
state(Object,P)          :- private(Object,P);
                            public(Object,P).
private(Object,P)        :- instance(Object,P).
public(Object,P)         :- Object=Class(I),
                            instance(Class,P).
```

A meta-interpreter *ist(Object,P)* for simple deductions implementing a restricted form of *Object* $\vdash$ *P* is defined as

```
ist(Object,P)  :- state(Object,P).
ist(Object,Q)  :- state(Object,P=>Q),
                  ist(Object,P).
ist(Object,(P,Q))        :- ist(Object,P),
                            ist(Object,Q).
```

Methods representing class or agent actions are contained in the agent's class. Methods are terms *method(Object.Call,Body)*, where *Call* is the name of a method followed by its parameters within parentheses and *Body* contains primitive procedure calls and/or *messages*. Messages sent to an *Object* have again the form *Object.Call*, where *Object* is either *Class* or *Class(I)*. Messages are interpreted by

```
Object.Call      :-state(Object, method(Object.Call,Body)),
                   call(Body).
```

where *Call(Body)* represents a call to Prolog itself. According to this implementation, class actions are activated by messages sent to the class itself with *Object=Class* (e.g.,

in order to achieve synchronisation between agents), and agent actions are activated by messages sent to individual agents with *Object=Class(I)* .

Agent classes and class members are created with predefined messages inserting the required methods and the initial state into the corresponding objects. All procedures defining the sequential abstract machine are implemented as class methods. The run of an individual agent is then obtained by sending the message *Class(I).run.*