Conscious behavior through reflexive dialogs

Pierre Bonzon

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

Abstract. We consider the problem of executing conscious behavior i.e., of driving an agent's actions and of allowing it, at the same time, to run concurrent processes reflecting on these actions. Toward this end, we express a single agent's plans as *reflexive* dialogs in a multi-agent system defined by a virtual machine. We extend this machine's planning language by introducing two specific operators for reflexive dialogs i.e., *conscious* and *caught* for monitoring beliefs and actions, respectively. The possibility to use the same language both to drive a machine and to establish a reflexive communication within the machine itself stands as a key feature of our model.

1 Introduction

Intelligent behavior, presumably, is strongly related to the concept of consciousness. In order to achieve true machine intelligence, one ought therefore to address the issue of modeling and executing *conscious behavior*. In the absence of a commonly agreed meaning for this term, we use it here in a somehow restricted sense i.e., to refer to the goal of "driving an agent's actions and of allowing it, at the same time, to run concurrent processes reflecting on these actions". We shall therefore not attempt to model the truly reflexive concept of "conscious to be conscious", which would be required for instance to capture the concept of "consciousness of having beliefs".

When compared to classical AI problems, such as automatic planning and/or reasoning, the amount of research devoted so far to this subject is rather limited. Surprisingly, we could hardly find more than a few references pertaining to recent work done in this area [5] [8] [15]. Provided that consciousness essentially functions as a mirror, the lack of formal models for the intelligence itself suggests that models of conscious behavior could not have anything to reflect upon. Furthermore, as this reflection seems to rely on internal linguistic representations relating beliefs and mental attitudes [6] [7], the lack of adequate formal languages for this purpose indicates why conscious behavior cannot be easily reproduced.

Comprehensive *agent* models, if available, could however well replace disembodied intelligence theories as the basis for modeling conscious behavior. More precisely, we envision that any system capable of mechanizing an agent's behavior could be first extended to reflect its selection of actions i.e., to somehow notify the agent of its choices. The language used by the system itself to plan actions should then be extended to use in turn these internal notifications. For an agent to be

conscious would then simply mean being able to recognize and acknowledge internal notifications at will. This overall process could be iterated to represent the concept of “conscious to be conscious”, and so on. Consciousness, taken as a whole, could thus be considered as the resulting "closure relation". As already indicated, we shall however refrain ourselves from exploring this concept, and be content with a possible implementation of the first iteration only.

Our recent proposal, that introduces formal communication primitives within a multi-agent system [3] and defines a language for agent *dialogs* [4], is an example of an agent model that can be used for this purpose. In this approach, consciousness will primarily function as a *monitor* of actions and beliefs (in the weak sense of the word and not in Hoare’s strict sense) that allows for the triggering of new actions. As such, this concept of consciousness is truly reminiscent of previously introduced artifacts, such as *demons*[1]. Our basic idea that departs from these earlier attempts is as follows: in order to catch internal notifications, any conscious agent will engage in *multiple* ongoing conversations *with itself*. The way for an agent to engage in multiple conversations with other agents have already been discussed in [4], and will be reviewed below. As a result, each communicating agent will be considered as a multithreaded entity interleaving concurrent conversations. The introduction of *reflexive* dialogs (in a non-traditional sense i.e., of having a dialog “with oneself” instead of "about itself") then simply requires an extension of their synchronization processes. From there, any agent’s plan will be represented as a *reflexive* dialog.

This model, which will be thoroughly developed in this paper, definitively reflects a "static" capability i.e., that of *being* conscious of explicit beliefs and of actions performed in full awareness. Different approaches might be possible. As an example, Baars [1] develops a concept of consciousness very much akin to a form of discovery and learning i.e., the "dynamic" process of *getting* conscious of facts resulting from myriads of sensations. We shall further relate these two approaches in our conclusion.

Let us further point out here the differential aspects pertaining to the consciousness of beliefs, on one hand, and of actions, on the other. As an agent’s beliefs are considered part of his local state, they can be represented by logical formulas that are stored in his memory. The consciousness of an agent’s beliefs is therefore *persistent* i.e., can be solicited at any time. In contrast, any action that he chooses to perform either have an effect on the environment or lead to an updating of his local state, and usually has no direct trace in his memory (unless, of course, it gives rise to an ad hoc new belief, as will be shown at the end of this paper). As such the consciousness of an agent’s actions is *volatile* i.e., must be caught “*on the fly”* when these actions occur. The differentiation just made will lead us to the definition of two distinct operators i.e., *conscious* and *caught* for monitoring beliefs and actions, respectively.

The rest of this paper is organized as follows: in section 2, we review previously published material in order to provide the reader with a basic understanding of our concept of agents dialog and its associated virtual machine. Section 3 shows how to represent any single agent’s plans as a reflexive dialog. Section 4 proposes the language extension allowing for the agent to reflect on his actions. Finally section 5 introduces the virtual machine extensions needed to notify an agent of his actions.

---

1 « a procedure that watches for some condition to become true and then activates an associated process »[14]

2 A model of social agents with reactive and proactive capabilities

In this section, we review the agent model introduced in [3] [4].

2.1 A language for agent dialogs

In order to get, first, an intuitive feeling for the language introduced in [4], let us consider the solution of the *two-agent meeting-scheduling* problem presented in [11]. In this problem, one agent is designated as the *host* and the other one as the *invitee*. Both agents have free time slots to meet e.g., if l^i refers to agent's *i* local state

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

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

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

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

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 an 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 and *save* the meeting time or *resume"*

The corresponding, implicitly synchronized *dialogs* are 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 dialog are listed before the messages. As it can be seen in this example, each dialog consists of a *branching sequence* of messages i.e., a *sequence* with an *end alternative* containing *guarded messages.* Similarly to lists, branching sequences can

have an embedded structure. Unless they are resumed (with a *resume* message), dialogs are exited at the end of each embedded branching sequence (e.g. the above example, after the *execute* messages). Actions interleaved with messages can be executed with an *execute* message. Sub-dialogues can be entered with an *enter* message, similarly to ordinary procedures (for an example, see section 2.4). 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>]*.

2.2 A virtual machine for executing dialogs

As thoroughly developed in [4], the language introduced in section 2.1 can be compiled and executed on a *virtual* machine. This compilation amounts to *rewriting* each dialog into a non-deterministic *plan* (see below), and at the same time generates the necessary conditions to ensure its sequential execution. The abstract machine itself defines the run of a class of agents as a loop interleaving individual agent run cycles. It is given by the following procedure

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

where *e* represents the state of the environment, *l* the *local state* of a class of agents defined by a vector $l = [l^{Class},l^1 \ldots l^n]$, and the components l^{Class} and l^i are the local state

of the class and its members identified by an integer $i=1...n$, respectively. We assume that predicate *agent* is such that $l^{Class} \vdash agent(i)$ whenever agent i belongs to the class. The language defining each l^i includes a set P of non-deterministic plan names (*nd-plan* in short) and four predicates *plan, priority, do* and *switch*. For each agent i, its current nd-plan $p^i \in P$ refers to a set of implications "*conditions*" $\Rightarrow$ $do(p^i, a)$ or "*conditions*" $\Rightarrow$ $switch(p^i, p^{i'})$, where a is an action (for an example, see section 2.4). Similarly to plans, *processes* of explicit priority n encompass implications "*conditions*" $\Rightarrow do(n, a)$. We 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., that $l^i \vdash plan(p_0^i)$ and $l^{Class} \vdash priority(n_0)$, respectively.

In each *run* cycle, initial plans p_0^i are activated by a procedure $react^i$. Synchronization of message pairs occurs globally through a procedure $process^{Class}$. These procedures are defined as:

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

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

In these procedures, the state transformer functions τ^i and τ^{Class} are used to interpret actions and synchronize operations, respectively. As their names imply, nd-plans are not executed sequentially. At each run cycle, procedure $react^i$ will be called with the (possibly variable) initial plan p_0^i deduced for each agent. In each recursive $react^i$ call, the agent's first priority is to deduce and carry out an action a from its current plan p^i. Otherwise, it may switch from p^i to $p^{i'}$. If the *switch* predicate defines decision trees rooted at each p_0^i, then $react^i$ will go down this decision tree. This mechanism allows an agent to adopt a new plan whenever a certain condition occurs, and then to react with an appropriate action. As a result, actions will be chosen one at a time. In contrast to nd-plans, dialogs must be executed sequentially. Towards this end, the rewriting of dialogs into nd-plans generates implications of the form "*conditions*" $\Rightarrow$ $do(p^i, a)$ only. In other words, there will be no switching of plans, and the state transformer function τ associated with the message *enter* (for entering sub-dialogs) will use instead a stack to reflect the dynamic embedding of dialogs. The *conditions* in each implication include both a synchronization and a sequencing condition (for further details, see [4]), and the action a always incorporate a predefined message together with some updating operations related to the conditions. As a result, a sequential execution of dialogs will be emulated, though actually each action will still be deduced one at the time using the mechanism defined for nd-plans.

As for procedure $process^{Class}$, it will execute, in descending order of priorities, all processes whose conditions are satisfied. Processes will be used for the purpose of

synchronizing message pairs. Synchronization occurs when the two messages forming a pair (i.e., *tell(r,φ)*/ask*(s,ψ)* or *call(r,φ)/return (s,ψ)* issued by sender *s* and receiver *r*) are acknowledged. It is triggered by two processes defined as

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

where the acknowledgment flag *ack(i,message)* is raised in l^{Class} when the *message* issued by agent *i* is interpreted using function τ^i (for further details, see [3]). Similarly to actions, the synchronizing operations *tellAsk* and *callReturn* are interpreted by state transformer functions τ^{Class} that uses l^{Class} as a blackboard.

2.3 Engaging in multiple conversations

Agents should be allowed to engage in multiple conversations. A possible solution is to define a parallel operator that can be used at the message level i.e., to interleave possible concurrent messages [10] [11]. We favor the simpler solution whereby each agent is a multi-threaded entity interleaving concurrent conversations. In this extended model, dialogs are now syntactic entities that, once compiled into plans, can be associated with multiple conversations implemented as concurrent threads. Just as our multi-agent system was implemented as a multi-threaded entity of agents using predicate *agent*, our multi-threaded agent will be implemented as a multi-threaded entity of conversations using an additional predicate *conversation* as follows

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

where $l^i \vdash conversation(j)$ means "conversation thread *j* is attached to agent *i* " and p_0^{ij} is the initial compiled plan associated with thread *j* of agent *i*. An additional primitive message *concurrent* can then be used in any dialog to create a new conversation thread when required (for an example, see the end of section 2.4). The synchronization processes must be redefined accordingly as follows

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

where the acknowledgment flag *ack(i^j, message)* is raised when the message issued by conversation thread *j* of agent *i* is interpreted. Note that messages are still addressed to a given agent rather than to a specific thread of that agent.

As dialogs can include *execute* messages that allow in turn for the execution of any action, the language just reviewed constitutes a general model of social agents with sensing, reactive and proactive capabilities [16] (this latter capability deriving from

the deduction of variable initial plans p_0^i). Its "operational semantics" is defined by the virtual machine together with the compiling functions contained in [4].

2.4 Example: a vacuum cleaner robot

To illustrate the concepts just reviewed, let us consider a vacuum cleaner robot that can choose either to *work* i.e., *move* and *suck* any dirt on sight, or to go back *home* and wait. Let us further assume that the robot must *stop* whenever an *alarm* condition is raised. These three behaviors correspond to three possible *nd-plans*, i.e. *work*, *home*, and *pause*. The robot's overall behavior can be represented by a decision tree rooted at a single *initial* plan and defined by the following implications, where the predicate *in(X,Y)* and *dirt(X,Y)* are used to mean "*the agent is located at (x,y)*" and "*there is dirt at (x,y)*", respectively, and the action *stop*, *move*, *back*, and *suck* have the obvious corresponding meaning:

```
  alarm                          ⇒ switch(initial,pause)
 ¬alarm                          ⇒ switch(initial,start)
   true                          ⇒ do(pause,stop)
   dirt(_,_)                     ⇒ switch(start,work)
 ¬dirt(_,_)                      ⇒ switch(start,home)
   in(X,Y)∧  dirt(X,Y)           ⇒ do(work,suck(X,Y))
   in(X,Y)∧ ¬dirt(X,Y)           ⇒ do(work,move(X,Y))
   in(X,Y)                       ⇒ do(home,back(X,Y))
```

These implications can be directly interpreted on the virtual machine of section 2.2. Each run cycle will be initiated with the single *initial* plan and then go down the decision tree. Each action will be deduced “just on time”, thus ensuring a truly reactive behavior. Equivalently, this robot can be specified by the following dialogs (that, at this point, do not involve any communication):

```
dialog(initial,  [],
                 [((alarm       | [enter(pause)]);
                   (not alarm   | [enter(start)]))])
dialog(pause,    [],
                 [execute(stop)])
dialog(start,    [],
                 [((dirt(_,_)       | [enter(work)]);
                   (not dirt(_,_)   | [enter(home)]))])
dialog(work,     [X,Y],
                 [((in(X,Y),dirt(X,Y)          | [execute(suck(X,Y)),
                                                  resume(initial)]);
                  (in(X,Y),not dirt(X,Y)       | [execute(move(X,Y)),
                                                  resume(initial)]))])
dialog(home,     [X,Y],
                 [((in(X,Y)                    | [execute(back(X,Y)),
                                                  resume(initial)]))])
```

The rewriting of dialogs into nd-plans generate conditions that will ensure their sequential execution, similarly to that of ordinary procedures (e.g., after entering *start*

from *initial*, the control will be transferred to either *work* or *home*, and so on). Unless explicitly directed to resume at some point, dialogs are exited at the end of each embedded branching sequence, and the dialog that was left on entering is then resumed by default. In the above example, the *initial* dialog is explicitly resumed after the execution of each action, thus enforcing the same reactive behavior as before.

As it can be seen in the above examples, the conditions in the implications defining nd-plans are identical to the guards in the guarded messages of the corresponding dialog. Furthermore, the *do* and *switch* predicates are used for the same purpose as the *execute* and *enter* messages, respectively. Intuitively then, a non-deterministic plan can thus be represented by a non-communicating dialog. Reversibly, any dialog can be compiled back into a nd-plan that do not contain any *switch* predicate. As already indicated, this arises because the state transformer functions for interpreting messages uses a stack to reflect the dynamic embedding of dialogs resulting from successive *enter* messages.

As a first step towards allowing an agent to reflect on his behavior, let us now try and express possible parallel tasks in dialogs. Recalling from the end of section 2.2 the possibility for a dialog to create *concurrent* conversation threads, our robot behavior can be further represented by the following dialogs

```
dialog(initial,   [],
                  [concurrent(pause),
                  [concurrent(start)])
dialog(pause,     [],
                  [((alarm | [execute(stop)]))])
dialog(start,     [],
                  [((not alarm | [concurrent(work),
                                  concurrent(home)]))])
dialog(work,      [X,Y],
                  [((dirt(_,_),in(X,Y),dirt(X,Y)        | [execute(suck(X,Y)),
                                                          resume(work)]);
                    (dirt(_,_),in(X,Y),not dirt(X,Y)    | [execute(move(X,Y)),
                                                          resume(work)]))])
dialog(home,      [X,Y],
                  [((not dirt(_,_),in(X,Y)              | [execute(back(X,Y)),
                                                          resume(home)]))])
```

where concurrent guards are pushed one level "below" (i.e., the guards of concurrent conversation threads are checked on entry in these threads). Dialogs *work* and *home* can now resume *themselves*, as they are now concurrent together with dialog *pause* waiting for an eventual alarm. Let us further note that these two dialogs actually *must* resume themselves: if the *initial* dialog were to be resumed instead, as in the preceding example, superfluous new concurrent threads would then be created.

Before proceeding, it is noteworthy to point out here the role of guards in guarded messages. If a guard does not get satisfied (e.g., as in the case of the *pause* dialog, when no alarm has been received yet), the associated message is not sent. If there are no other alternatives, the dialog gets simply suspended until the guard gets satisfied. The guarded message as a whole thus acts as a *monitor* or *demon* i.e., it will stand alive, watch and wait until its associated message can eventually be sent.

3 Representing agent plans as reflexive dialogs

According to their intuitive meaning (and as defined by their compiling functions in which guarded messages are rewritten into implications), guards are required to be deductible from the agent local state. Let us now recall our introductory discussion about the *persistence* of beliefs v/s the *volatility* of action consciousness. We thus have to conclude that, in contrast to the monitoring of beliefs illustrated in the preceding example, guarded commands cannot be used to monitor actions. Looking for an alternative and general solution that will apply to both cases, our basic idea is to try and catch internal notifications by allowing any agent to engage in *multiple* conversations *with itself*. Towards this end, we first need to be able to process *reflexive* dialogs (in a non-traditional sense of the word i.e., a dialog "with oneself"). An ad hoc extension of the synchronization processes presented in section 2.2 simply requires two additional synchronization processes, defined as follows:

$ack(s^j,tell(k,\varphi)) \wedge ack(s^k,ask(j,\psi))) \Rightarrow do(2, tellAsk(s^j,s^k,\varphi,\psi))$

$ack(s^j,call(k,\varphi)) \wedge ack(s^k,return(j,\psi)) \Rightarrow do(1, callReturn(s^j,s^k,\varphi,\psi))$

where the sender and the receiver are the threads *j* and *k* attached to agent *s*. Reflexive messages are thus sent by and addressed to specific threads of a given agent.

To illustrate this, let us now define a generic dialog *conscience* as follows:

```
dialog(conscience(P),    [Thread],
                         [((P     | [return(Thread,P)]))])
```

Suppose that this dialog has been attached to a given agent, and receives the message **call**(conscience(P),P) sent by a concurrent conversation (named *Thread*) attached to the same agent. This dialog will then either *return* the belief P and exit, if this belief holds in the agent's local state, or wait, in the contrary case. Let us further consider a new message *conscious* that can be macro expanded as follows

```
conscious(P)  ==>  (concurrent(conscience(P)), call(conscience(P),P))
```

This message will first create a conscience thread and then work as described above i.e., like a monitor for beliefs that will stand alive, watch and wait until expected beliefs are effective. If we assume that reflexive dialogs are precompiled to macro expand *conscious* messages and to include the generic dialog *conscience,* then our last example of section 2.4 can be rewritten as follows:

```
reflexive(initial,  [],
                    [concurrent(pause),
                     concurrent(start)])
reflexive(pause, [],
                    [conscious(alarm),
                     execute(stop)])
reflexive(start,    [],
                    [conscious(not alarm),
                     concurrent(work),
                     concurrent(home)])
```

```
reflexive(work,  [X,Y],
                 [conscious(dirt(_,_)),
                  conscious(in(X,Y)),
                  ((dirt(X,Y)          | [execute(suck(X,Y)),
                                          resume(work)]);
                   (not dirt(X,Y)      | [execute(move(X,Y)),
                                          resume(work)]))])
reflexive(home,  [X,Y,]
                 [conscious(not dirt(_,_)),
                  conscious(in(X,Y)),
                  execute(back(X,Y)),
                  resume(home)])
```

Let us stress here that this solution leads to the same behavior as before. In other words, in this example, the monitoring of individual beliefs using reflexive dialogs instead of guarded messages does not bring anything new. The explicit modeling of conscience threads does however open the door to more complex consciousness models relying on mental attitudes e.g., could lead to model such things as a troubled, selective or biased conscience. A similar scheme, introduced in the next section, will allow for the monitoring of actions.

The verification of dialog protocols expressed as concurrent reentrant threads guarded with **conscious** monitors could be quite an intricate task. In order to prevent concurrent threads to get "stuck", their monitors should not get bound by external variables. In the above example, the monitors of concurrent threads *work* and *home* get bound by local variables *X* and *Y*. In contrast, actions *suck(X,Y)* and *move(X,Y)* cannot be expressed as concurrent reentrant threads because their monitor would be bound by external variables *X* and *Y*.

4 Executing conscious behavior by reflecting on one's own actions

As alluded to in our introductory discussion, the consciousness of an agent's actions is *volatile* i.e., must be caught "*on the fly*" when these actions actually occur. What we need to implement now is a mechanism whereby an agent will first be notified of, and then reflect on each of its actions individually. Once again, our language for agent dialogs will be used for this purpose. In contrast to the consciousness of an agent's beliefs, the sender of the notifications cannot be the agent itself, who will be solely the receiver, and the notifications will be sent by the underlying virtual machine. Towards this end, let us define a generic dialog *reflect* as follows

```
dialog(reflect(P), [Thread],
                   [ask(react,P),
                    tell(Thread,P)])
```

As before, suppose that this dialog has been attached to a given agent and receives the message **ask**(reflect(P),P) sent by a concurrent thread attached to the same agent. This dialog will first send the message **ask**(react,P) to a pseudo thread *react* representing the machine itself (or, more precisely, the *react* procedure of an extended virtual machine, as it will be explained in the next section) and then wait for an answer. Upon receiving a notification, it will in turn answer the asking thread by

sending the message **tell(**Thread,P). Let us further consider a new message *caught* that can be macro expanded as follows

caught(P) ==> (**concurrent**(reflect(P)), **ask**(reflect(P),P))

This message will first create a *reflect* thread and then work as described above i.e., will monitor the notification of actions. If we assume that reflexive dialogs are precompiled to macro expand *caught* messages and to include the dialog *reflect*, then any behavior could be directed to reflect on his own actions, using a concurrent reflexive dialog *introspect* saving ad hoc new beliefs *done(P)* as follows:

```
reflexive(initial,   [],
                     [concurrent("any behavior"),
                      concurrent(introspect)])
reflexive (introspect, [P],
                       [caught(execute(P)),
                        execute(save(done(P))),
                        resume(introspect)])
reflexive ("any behavior", [P],
                           [ …
                            conscious(done(P)),
                            … ])
```

The monitoring of actions just presented represents a first step towards modeling conscious behavior. As illustrated by the *introspect* thread, any number of concurrent threads could similarly be designed to reflect in various ways on the execution of actions. For example, the ad hoc new beliefs *done(P)* created by the *introspect* thread could be monitored in turn to relate the consciousness of *previous* actions to that of the *current* action, and so on.

5 An extended virtual machine for sending notifications

In order to complete the model, the virtual machine presented in section 2.2 must be extended to send internal notifications, when required. This extended virtual machine can be defined as follows:

```
procedure run^Class(e,l)
   loop
      for all i such that l^Class ⊢ agent(i) do
         sense(l^i,e);
         for all j such that l^i ⊢ conversation(j) and j≠reflect(_) do
            if l^i ⊢ plan(p0^ij)
            then react^i(e,l,p0^ij );
                 reflect^i(e,l);
                 if l^Class ⊢ priority(n0)
                 then process^Class(e,l,n0)
```

In order to avoid infinite recursion, this machine is prevented from sending notifications about notifications (in other words, the consciousness is not iterated to represent the concept of "being conscious to be conscious", and so on). The *reflect*

threads are thus not interleaved with other threads, but are executed separately in each cycle using procedure $reflect^i(e,l)$. This procedure is defined in turn as follows:

procedure $reflect^i(e,l)$
if $l^i \vdash conversation(reflect(r))$
  and $l^i \vdash do(reflect(r), a)$
then $(e,l) \leftarrow \tau^i(e,l,a);$
  $reflect^i(e,l)$

As a result of the end recursive call $reflect^i(e,l)$, all ready messages of all concurrent *reflect* threads attached to agent *i* will be sent without delay. The actual notification takes place within an extended procedure $react^i(e,l,p^{ij})$ defined as follows:

procedure $react^i(e,l,p^{ij})$
if $l^i \vdash do(p^{ij}, a)$
then $(e,l) \leftarrow \tau^i(e,l,a);$
  if $l^i \vdash conversation(reflect(r))$
    and $a = (save(\_,\_),save(\_,\_),r)$
  then $ack(i^{react},tell(reflect(r),r) \leftarrow true$
else if $l^i \vdash switch(p^{ij}, p^{ij'})$
  then $react^{ij}(e,l,p^{ij'})$

Any action *a* resulting from the compilation of dialogs has the form of a triplet *a=(save(_,_),save(_,_),r)*, where *r* is one of the predefined message. After executing an action *a*, this extended procedure will check if the message *r* just processed can be matched with a *reflect(r)* thread attached to agent *i*. If so, it will raise an acknowledgment flag $ack(i^{react},tell(reflect(r),r))$ that in turn will be paired for synchronization with the acknowledgement flag $ack(i^{reflect(r)},ask(react,r))$ raised by message **ask**(react,r) from thread *reflect(r)*. As the thread *react* actually does not exist, this amounts to emulating communication between the machine and the agent *i*.

This overall process can be represented by the following picture that elaborates on the abstract, top-level view of an agent as given in Wooldridge's reference paper [16]:

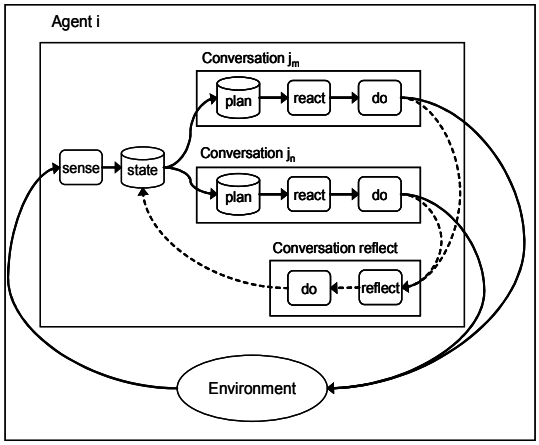


Fig. 2.

6 Related and further work

Previous proposals for representing consciousness either lack an explicit formal model [8] [9], functional specifications towards a possible implementation [5], or both [15]. Using the same language to drive a machine and to reflexively communicate within the machine itself stands as a key feature of our own model. As consciousness basically works as a mirror, we view this duality to be an essential characteristics.

As already pointed out in the introduction, our model definitively reflects a "static" capability i.e., that of *being* conscious of explicit beliefs and of actions performed in full awareness. Baars [1] develops a concept of consciousness very much akin to a "dynamic" process of discovery and learning i.e., that of *getting* conscious of implicit facts resulting from sensations. Intuitively, Baars sees the human brain as being populated by myriads of parallel *unconscious processors* that compete for access into a *global workspace*. This workspace functions as a serial channel of limited capacity that can broadcast information to the unconscious processors. In various (possibly embedded) *contexts,* unconscious processors may form *coalitions* that will then force their way into the global workspace. Baars' theory is only described in general terms, and captured graphically in sets of diagrams of the type given below:

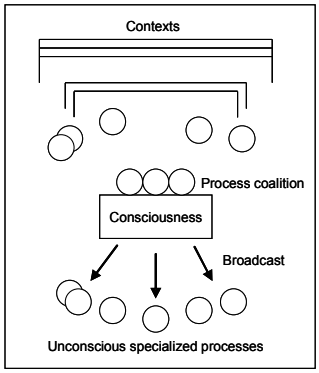


Fig. 3.

Although Baars himself once wrote "we now have a number of computational formalisms that can be used to make the current theory more explicit and testable", we do not know of any attempt to develop the corresponding formalization. These ideas however have already found their way into practical applications [8]. Unfortunately, as in many other artificial intelligence models, the "theory is the program" i.e., theoretical concepts are buried into ad hoc implementations that alone cannot qualify as a formalization of the theory.

Similarities do exist between Baars' theory involving a distributed system of parallel unconscious processors and our model of concurrent threads in a multi-agent

system. First, our use of a blackboard for synchronizing messages is similar to Baars' workspace for broadcasting conscious messages. Other analogies to be found include:

serial information broadcast v/s our *blocking* communication primitives
goal *hierarchies* v/s our plan *decision trees*
dominant goal v/s our *initial* plan deduction
and (possibly) *process coalition* v/s *theory lifting*.

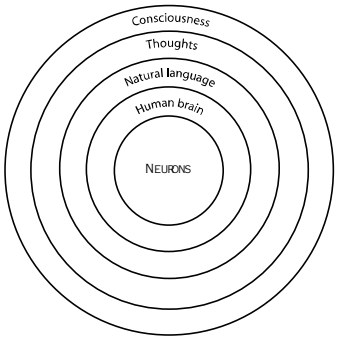
This last point is a mere conjecture that deserves an explanation. The concept of *theory lifting* was introduced by J. McCarthy in his attempt at formalizing contexts [12]. An executable account of this concept was given in [2]. We now suspect, and will try and formalize the hypothesis, that theory lifting can be used to model process coalition. As an example of a "Gedanken" experiment [8] that may be attempted, "a white square" should be recognized as "a sail" or "a hanging bed sheet" by lifting contextual knowledge associated with the surrounding landscape.

7 Conclusions

The explicit modeling of conscience threads, introduced in section 3, together with the possible reflection on one's own actions presented in section 4, opens the door to more complex consciousness models that should go beyond the simple monitoring of beliefs illustrated in this paper.

From a technical point of view, it is interesting to note that the *conscious* operator was defined using a *call(r,φ)/return(s,ψ)* pair of communication primitives, where $\varphi\theta=\psi\theta$ and $l^r \vdash \psi\theta$, with l^r referring to the agent's local state holding his beliefs. In contrast, the *caught* operator was defined using a *ask(s,ψ)/tell(r,φ)* pair involving a simplified $\varphi \vdash \psi\theta$ operation (because the agent's local state cannot be of any use in this case). Somehow, this a posteriori justifies the choice of communication primitives we made in [3].

We are well aware of the formal inconsistencies that arise when trying to model self referential sentences using a single uniform language, as discovered by Montague [13]. To escape from these pitfalls, we adopted a *constructivist* point of view i.e., similarly to nature itself, we grounded our concept of consciousness on successive, distinct operational layers, as summarized in the following pictures:



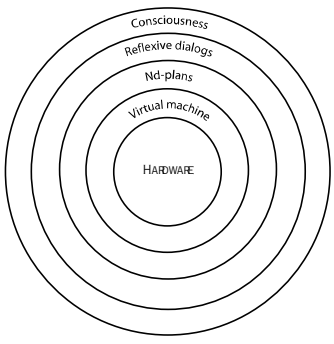


Fig. 4. Natural consciousness v/s simulated consciousness

Whereas inner layers interact with neighbors only (i.e., the virtual machine executes on the hardware, nd-plans are interpreted by this machine, dialogs are compiled into nd-plans, etc.), outer consciousness does rely on a direct access to a deeper underlying layer i.e., the virtual machine represented by the pseudo thread *react*. We are tempted to postulate that similar things happen in the working of natural consciousness.

Acknowledgment

We are indebted to an anonymous referee for his careful reading of this paper.

References

1. B.J. Baars, *A Cognitive Theory of Consciousness*, Cambridge University Press (1988)
2. P. Bonzon, A Reflexive Proof System for Reasoning in Context, *Proc. 14th Nat'l Conf. On AI , AAAI97* (1997)
3. 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)
4. P. Bonzon, Compiling Dynamic Agent Conversations, in:M.Jarke,J.Koehler&G.Lakemeyer (eds), *Advances in Artificial Intelligence*, LNAI vol. 2479, Springer Verlag (2002)
5. J. Cunningham, Towards an Axiomatic Theory of Consciousness, Logic Journal of the IGPL, vol. 9, no 2 (2001)
6. D.C. Dennett, *Consciousness Explained*, Little Brown (1991)
7. O. Flanagan, *Consciousness Reconsidered*, MIT Press (1992)
8. S. Franklin & A. Graesser, A Software Agent Model of Consciousness, *Consciousness and Cognition*, vol. 8 (1999)
9. S. Franklin, Modeling Consciousness and Cognition in Software Agents, in: N. Taatgen, (ed.), *Proceedings of the International Conference on Cognitive Modeling*, Groeningen (2000)
10. G. de Giacomo, Y.Lespérance and H. Levesque, ConGolog, a Concurrent Programming Language Based on the Situation Calculus, *Artificial Intelligence*, vol. 121 (2000)
11. 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)
12. J. McCarthy, Notes on Formalizing Context, *Proc.13th Joint Conf. on Artificial Intelligence IJCAI93* (1993)
13. R. Montague, Syntactical treatment of modalities, with corollaries on reflexion principles and finite axiomatizability, Acta Philosophica Fennica, vol. 16 (1963)
14. E. Rich, *Artificial Intelligence*, McGraw-Hill (1983)
15 A. Sloman & R. Chrisley, Virtual Machines and Consciousness, submitted (2002)
16. M.Wooldridge, Intelligent Agents, in:G.Weiss (ed.), *Multiagent Systems*, MIT Press (1999)