PRACTIONIST: implementing PRACTIcal reasONIng sySTems

Vito Morreale*, Susanna Bonura*, Fabio Centineo*,
Alessandro Rossi*, Massimo Cossentino† and Salvatore Gaglio†‡
*R&D Laboratory - ENGINEERING Ingegneria Informatica S.p.A.
†ICAR-Italian National Research Council
‡DINFO-University of Palermo

***Abstract*— One of the best known approaches to the development of rational agents is the BDI (Belief-Desire-Intention) architecture. In this paper we propose a new framework, PRACTIONIST (PRACTIcal reasONIng sySTem), to support the development of BDI agents in Java (using JADE) with a Prolog belief base.**

In PRACTIONIST we adopt a goal-oriented approach with a clear separation between the deliberation and the means-ends reasoning, and then between the states of affairs to pursue and the way to do it. Besides, PRACTIONIST allows developers to implement agents that are able to reason about their beliefs and the other agents' beliefs, expressed by modal logic formulas.

Our approach also includes a specific tool that provides the developer with the possibility to effectively monitor the components involved in the execution cycle of an agent.

I. INTRODUCTION

The Belief-Desire-Intention (BDI) architecture [1] derives from the philosophical tradition of practical reasoning first developed by Bratman [2], which states that agents decide, moment by moment, which actions to perform in order to pursue their goals. Practical reasoning involves two processes: (1) *deliberation*, to decide what states of affairs to achieve; and (2) *means-ends reasoning*, to decide how to achieve these states of affairs. Besides, in such a theory intentions are important, as they influence the selection of the actions to perform.

In the context of rational agents, the BDI model appears very attractive, because the abstractions of belief, desire and intention are quite intuitive. Moreover the model provides a clear functional decomposition that indicates what sort of subsystems might be required to build an agent. Nevertheless, the development of this abstract architecture involves several issues in efficiently implementing the deliberation process and the means-ends reasoning [3].

Moreover, since the BDI agent model suggests a declarative approach to represent the internal states, the debugging of BDI agents, the effective observation of their mental attitudes and execution flow are critical and difficult activities. Thus, having some development and debugging tools is crucial when implementing BDI agents, especially in real case scenarios or complex application domains.

Actually, several concrete implementations of the most known BDI agent architecture, the Procedural Reasoning System (PRS) developed by Georgeff and Lansky [4], have been proposed in the literature. Among them, it is worth mentioning dMARS [5] developed at the Australian AI Institute, the UM-PRS implemented in C++ at the University of Michigan [6], and JAM [7], a Java version of PRS.

In order to enable the testing of BDI agents, the 3APL platform [8], an experimental multiagent platform, provides a graphical interface by which designers can develop, execute, and monitor the agents. JADEX, an add-on to the JADE platform [9] that supports the development of BDI agents, provides two tools as a support of the JADE introspector agent: the *debugger*, which allows the visualization and reconfiguration of the internal BDI concepts, and the *logger agent*, which allows developers to detect the agent's sequence of outputs [10]. Finally, the JACK software [11], a commercial suite of tools with a programming language that extends the Java language with BDI features, provides an agent debugging environment, which allows inspection of messages and the internal execution states.

In several PRS-related BDI implementations, mental states, deliberation, and means-ends reasoning, when actually implemented, somewhat differ from their original meaning. As an example, often executing plans are considered as intentions. But intentions should be related to ends, while plans should be related to means to achieve such ends.

PRACTIONIST (PRACTIcal reasONIng sySTem) is a new framework we have been developing, which adopts a goal-oriented approach and stresses the separation between the deliberation process and the means-ends reasoning. Indeed, the abstraction of goal is used to formally define both desires and intentions during the deliberation phase. Unlike some of existing BDI implementations, in our approach we actually adopt the plans as recipes to achieve the intentions. Besides, PRACTIONIST allows developers to implement agents that are able to reason about their beliefs and the other agents' (including humans') beliefs, since beliefs are not simple grounded literals or data structures but modal logic formulas [12].

Finally, our framework provides the developer with the PRACTIONIST Agent Introspection Tool, to entirely and easily monitor the components involved in the execution cycle of an agent. Throughout this paper we show how PRACTIONIST agents actually work by means of snapshots of our monitoring tool and through the well-known *blocks world* example. In this simple case study, we developed a *blocks world agent*, which,

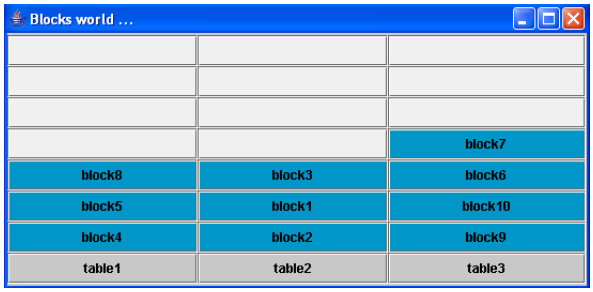

Fig. 1. An initial situation for the blocks world problem.

starting from an initial situation (see figure 1), is requested (through an ACL message) to order some numbered blocks.

This paper is organized as follows: in section II we give a brief overview of the PRACTIONIST framework and the agent model; then in sections III to V we provide a brief description of the main agent components; finally in section VI we describe the execution flow of PRACTIONIST agents referring to their previously described components.

II. The PRACTIONIST Framework: an Overview

The PRACTIONIST framework aims at supporting the programmer in developing agents *(i)* endowed with a symbolic representation about their internal states and environment, *(ii)* able to plan their activities in order to pursue some objectives, and *(iii)* provided with both proactive and reactive behaviours.

PRACTIONIST has been designed on top of JADE, a widespread platform that implements the FIPA specifications [13] and provides some core services, such as a communication support, interaction protocols, life-cycle management, and so forth. Therefore, the PRACTIONIST agents are executed within JADE containers and the main cycle is implemented by means of a JADE cyclic behaviour.

In PRACTIONIST, an agent is a software component with the following elements: *(i)* a set of *perceptors* able to listen to some relevant perceptions; *(ii)* a set of *beliefs*, which represents the information the agent has got about both its internal state and the external world; *(iii)* a set of *goals*, which are some objectives related to some states of affairs to bring about or actions to perform; *(iv)* a set of *plans* that are the means to achieve its intentions; *(v)* a set of *actions* the agent can perform to act over its environment; *(vi)* and a set of *effectors* that actually support the agent in performing its actions. The main components of PRACTIONIST agents are described in more details in the following sections.

The framework also provides developers with the PRACTIONIST Agent Introspection Tool (PAIT), a visual integrated monitoring and debugging tool, which supports the analysis of the agent's state during its execution. In particular, the PAIT can be suitable to display, test and debug the agents' relevant entities and execution flow. Each of these components can be observed at run-time through a set of specific tabs (see figure 2); the content of each tab can be also displayed in an independent window.

All the information showed at run-time could be saved in a file, providing the programmer with the possibility to perform an off-line analysis. Moreover, the PAIT provides a dedicated area for log messages inserted in the agent source code, according to the Log4j approach [14]. The usage of this console and the advantages it provides are described in more details in the following sections along with the agent's components.

III. Beliefs

In general, the BDI model refers to beliefs instead of knowledge, as agents' information about the world is usually incomplete or incorrect, due to uncertainty and problems with perceptions and communication in their dynamic and possibly unpredictable environment [1]. Indeed, *beliefs* are not necessarily true, while *knowledge* usually refers to something that is definitely true [12]. According to this, an agent may believe true something that is false from another agent's and/or the designer's point of view, but in the BDI model the idea is just to provide the agents with a subjective window onto the world.

The PRACTIONIST framework adopts the common approach of modeling agents' beliefs by the *doxastic* modal logic, which is based on the axioms K, D, 4, and 5 (see [12] for more details). Thus, in our framework beliefs are expressed through the modal operator $Bel(\alpha, \varphi)$, whose arguments are the agent α (the believer) and what it believes (φ, the fact). Each fact φ may be believed true or false by a PRACTIONIST agent α, i.e.:

- $Bel(\alpha, \varphi)$: the agent α believes that φ is true;
- $Bel(\alpha, \neg\varphi)$: the agent α believes that φ is false.

Moreover, in order to assert that the agent α does not have any belief about φ, we defined the following operator:

$$Ubif(\alpha, \varphi) \Leftrightarrow \neg Bel(\alpha, \varphi) \wedge \neg Bel(\alpha, \neg\varphi)$$

Each fact can either be a closed formula of the classical modal logic or a belief of any agent. In other words, an agent may believe something (e.g. 'it is possible that it is raining in Rome'), or have *nested beliefs*, that is it may believe that some agent (even itself) believes something (e.g. 'the agent Jim believes that it is raining in Rome').

Finally, in PRACTIONIST agents it is possible to link an agent's beliefs to others' beliefs or other elements, obtaining new entailed beliefs, through the *belief formulas* (BFs). An example of belief formula follows:

$$Bel(tom, Bel(john, raining)) \wedge Bel(tom, trust(who : john)) \Rightarrow Bel(tom, raining)$$

Therefore BFs define relationships among two or more beliefs, allowing to infer new beliefs not explicitly asserted.

In regards to the architectural aspects, each PRACTIONIST agent is endowed with a Prolog belief base, where beliefs are asserted, removed, or entailed through the inference on the basis of KD45 rules and user-defined belief formulas. Therefore, in any moment the agent's belief set (BS) is composed of the beliefs that have been both directly asserted

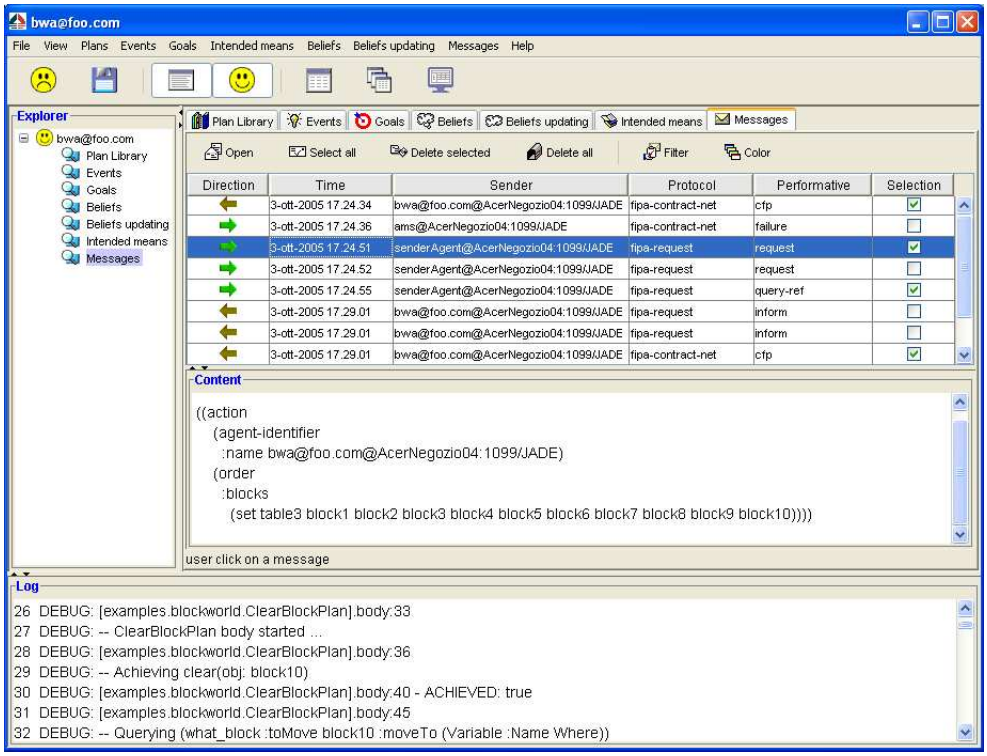

Fig. 2. The PRACTIONIST Agent Introspection Tool (PAIT).

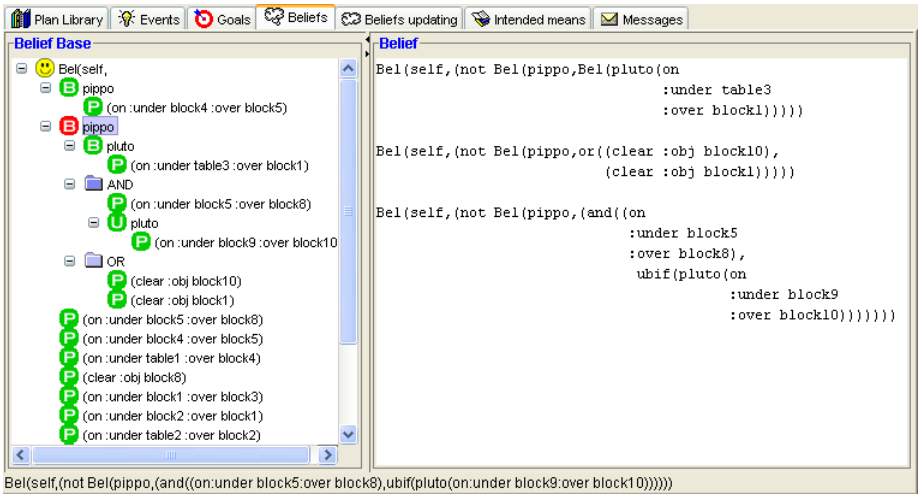

Fig. 3. The belief base view of PAIT.

and inferred by means of the BFs and the other built-in theorems.

Referring to the blocks world example, the initial set up of the blocks shown in figure 1 is graphically represented in PAIT in terms of *Bel*s and *Ubif*s (figure 3). As it can be noticed, the structure of the belief base is represented as a tree, whose root refers to the current agent (i.e. *self*) and nodes refer to the believed facts. This structure lets the developer easily explore the belief base and select groups of beliefs ranging from the whole belief set (by selecting the root) to a specific belief (by selecting the corresponding leaf). Between them the developer can choose an intermediate node in order to select the corresponding set of beliefs that share the same prefix.

Moreover, the icon of nodes represents the type of believed facts, such as *predicate* or *modal logic formula* (P), *Bel* (B), *Ubif* (U), while the color represents their truth value, that is *true* (green) and *false* (red).

As an example, selecting the node related to the agent *pippo*, the beliefs with

$$Bel(self, not(bel(pippo,$$

as prefix will be detailed on the right frame, as shown in figure

3. On the other hand, if the developer selects the leaf related to the agent *pluto* within the above-mentioned node, PAIT will show only the belief

$$Bel(self, not(Bel(pippo, Bel(pluto, on(under : table1, over : block3)))))$$

It should be also noticed that this feature of the console is very useful when developing and testing agents, as it provides the user with a real-time snapshot of the agent's information attitudes.

IV. PLANS

In the PRACTIONIST framework plans represent an important container by which developers define the actual behaviours of agents. Therefore, each agent may own a declared set of plans (the *plan library*), which specify the activities the agent should undertake in order to pursue its intentions, or to handle incoming perceptions, or to react to changes of its beliefs.

Though the structure of plans can be defined by Java classes, the preferred approach relies on a declarative *plan description*, which specifies the complete set of information (the *plan slots*) used when actually executing the plans, as described in section VI. The complete list of such slots is reported in table I.

Thus, a plan represents a possible recipe to manage the trigger event; it may be related to a goal, an external event, or an event which notifies a change of the belief set. How to actually handle a certain event is reported within the body, which is an *activity*, that is a set of *acts*, such as *desiring* to bring about some states of affairs or to perform some action, *adding* or *removing* beliefs, *sending* ACL messages, *doing* an action and so forth.

It should be noticed that acts and actions are different, as one of the possible acts concerns with doing an action that lets the agent influence the environment, while other acts may produce internal effects only.

Regarding to the blocks world running example, one of the developed plans is the *TopLevelPlan*, which provides the agent with the capability of receiving and handling the request for ordering the blocks. In section VI we illustrate how the plans and their components are used during the execution flow and how the execution of plans affects the overall behaviour of PRACTIONIST agents.

V. ACTIONS

In PRACTIONIST, actions are described by tuples of four elements: (1) *arguments*, which are the objects each action acts over; (2) *results*, which are some kind of direct responses received from the environment; (3) *preconditions*, which should be satisfied before executing the action; and (4) *effects* (for both successfully and failing action execution), which are the state of affairs that will be true or false after executing the action (as long as preconditions are satisfied). It should be noticed that arguments and results are objects, while preconditions and effects are modal logic closed formulas.

TABLE I
THE STRUCTURE OF PRACTIONIST PLANS.

Identifier	Unambiguous (within each agent) identifier of plans
Trigger event	If this event matches the selected event, this plan can be activated. In this case the plan is defined as *practical*
Context	A modal logic formula that, when believed true by the agent, makes *applicable* a practical plan, so that the agent can select it to pursue its objectives
Success condition	When the agent believes that this condition holds, the plan ends with success, regardless its execution state
Cancel condition	When the agent believes that this condition holds, the plan ends with failure, regardless its execution state
Body	Set of acts that are performed during the execution of the plan. The body defines the actual behaviour of the plan
Invariant	Condition that must remain true during the execution of the plan. As soon as it becomes false (at least according to the agent's point of view), it will try to restore it
Belief updates in case of success	Effects of this plan, in terms of belief updates in case the plan ends with success
Belief updates in case of failure	Effects of this plan, in terms of belief updates in case the plan ends with failure

Actions are implemented in PRACTIONIST through Prolog structures or Java classes that include the above-mentioned elements. An example of action description from the blocks world agent follows:

```
action(move(block: Block, to: To),
  inputs: [Block, To],
  outputs: [],
  preconditions:
   [ on(over: Block, under: From),
     clear(obj: To), clear(obj: Block) ],
  success:
   [ -clear(obj: To),
     -on(over: Block, under: From),
      +clear(obj: From),
      +on(over: Block, under: To) ],
  failure: [])
```

It states that the action $move$ takes two arguments as inputs (respectively the block to move and the block where it is moved over). Moreover, preconditions $on(over : Block, under : From)$, $clear(obj : To)$, and $clear(obj : Block)$ must be satisfied before performing the action in order to have a proper execution. Finally, once the action has been executed, in case of success the agent will believe that both $clear(obj : To)$ and $on(over : Block, under : From)$ are false, while it will believe that both $clear(obj : From)$ and

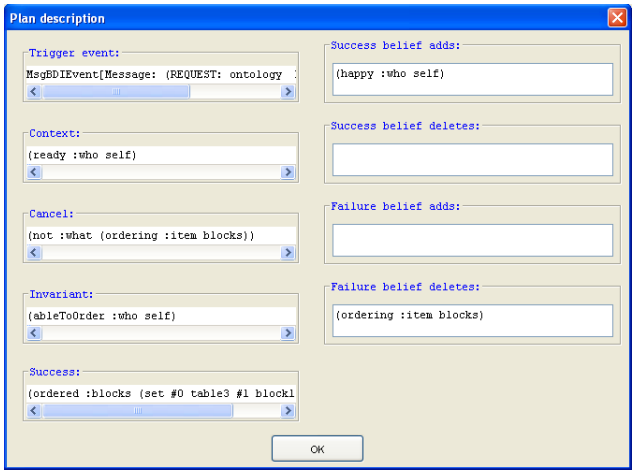

Fig. 4. An example of plan description: the *TopLevePlan*.

$on(over : Block, under : To)$ are true. Otherwise, in case of failure in executing the action, no update of the agent's beliefs has to be done.

Planning attitudes and the decision making of PRACTIONIST agents will rely on action description information, especially preconditions and effects.

VI. Execution Flow

In this section, we present the execution flow of PRACTIONIST agents in terms of relationships among the abstractions described above. Referring to figure 5, an agent cyclically performs the following steps:

1) it searches for stimuli from the environment through the perceptors that transform perceptions into *events* (we call them *external events*), which in turn are put into the *Event Queue* (in figure 6 a snapshot of the events tab of PAIT is shown), along with *internal* events (those generated by belief updates or goal commitments).
   In the PRACTIONIST console, the user could examine the current, the suspended and the handled events. Each event is tagged by some information about its arrival time and when it has been actually handled.
   Referring to the blocks world example, the agent will receive an ACL message, by which another agent requests it for putting the blocks in a specified order (see the selected message in figure 2). This component of the PAIT deals with the interactions established with other agents: all incoming and outgoing messages are registered in a single table, even though a complete description of messages can be shown in a different dialog box whose structure reflects the well-known FIPA ACL message. Besides, the messages can also be ordered (e.g. by their arrival time, direction, etc.) or filtered according to the relative performative;
2) it selects and extracts an event from the queue (*Event Selection*). In the blocks world example, let us suppose that the event corresponding to the above-mentioned received message is extracted from the queue and then handled as follows;
3) it selects *practical* plans from the *Plan Library*, which are those plans whose trigger event matches the selected event (*Options* in figure 5); if the selected event is related to a goal and no plan has been triggered, an automatic planning is performed on the basis of available action descriptions in order to build a new dynamically-generated plan, which is able to pursue that goal (*Planning* in figure 5). If the planner is not able to figure out any plan, the calling plan will fail, then the selected event will have not been successfully managed. As stated, in the blocks world example we have defined a plan (i.e. $TopLevelPlan$) that will be activated by the above-mentioned message, as it has the following trigger event:

```
msg(request
     (action
       (agent-identifier :name bwa@foo.com)
       (order(blocks: BlockList))))
```

   Once the plan is triggered, the following substitution is made:

```
BlockList = [ table3, block1, block2,
              block3, ..., block8,
              block9, block10]
```

4) among practical plans, the agent detects the *applicable* ones, which are those plans whose context is believed true by the agent. Then the agent builds the intended means (*Build Intended Means*), which represents the means the agent has just chosen and committed to in order to satisfy a goal, or to react to a perception or a change in its beliefs. Therefore, the intended means will contain the main plan and the other alternative practical plans (see figure 7).
   If the event selected at the step (2) concerns with a goal, this means that some executing intended means has generated it after the deliberation phase (see below). Thus, the selected plan is put on top of such an intended means (figure 7a). On the other hand, in case of external events or belief update events, a new intented means stack is created (figure 7b).
   The set of intended means stacks is monitored by means of a corresponding tab in PAIT (figure 8), which shows nested intended means through a tree data structure, used as an explorer to select the ones to trace. In particular, once an intended means is selected, its log messages (managed inside the plan's source code by the developer) will be shown in the frame on the right, along with the logs of nested intended means. The PRACTIONIST console makes the process of building intended means stacks easy to follow, providing the users with the opportunity to examine the entire means-ends reasoning;
5) all intended means stacks are concurrently executed in separate threads. In other words, the main plan at the top of each stack is extracted and then executed (see *Execute intended means* in figure 5). In section VI-A we provide more details on the execution of plans in PRACTIONIST agents, in terms of the several kind of

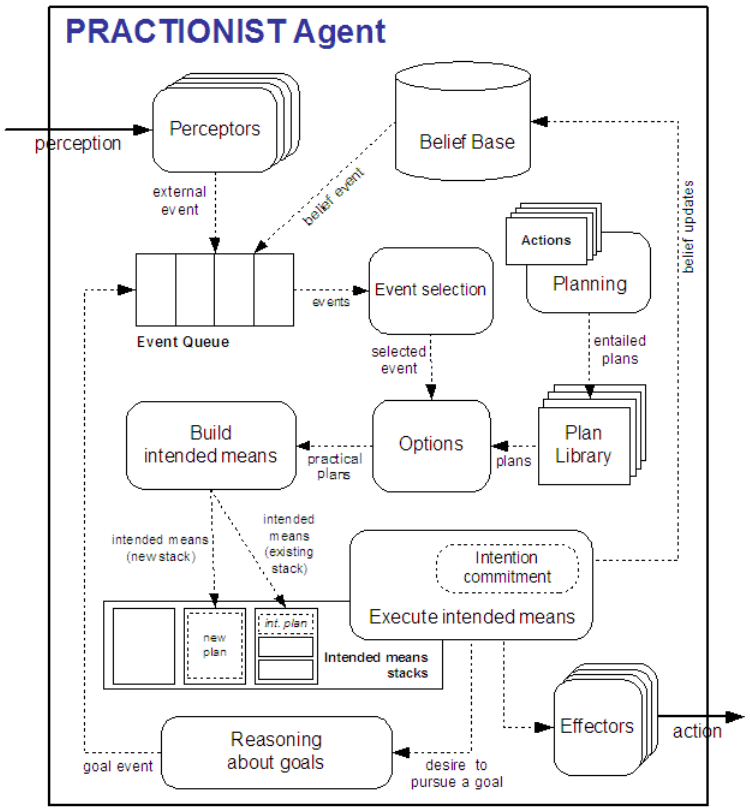

Fig. 5. PRACTIONIST agent architecture.

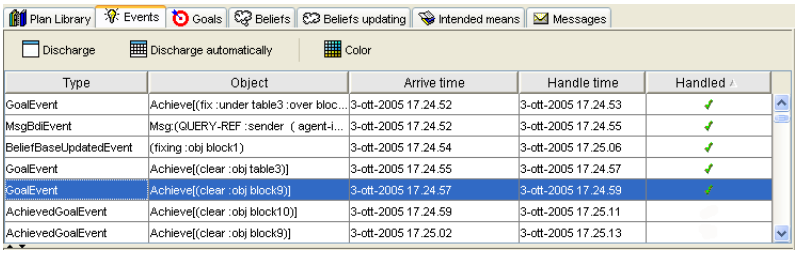

Fig. 6. The event queue view of PAIT.

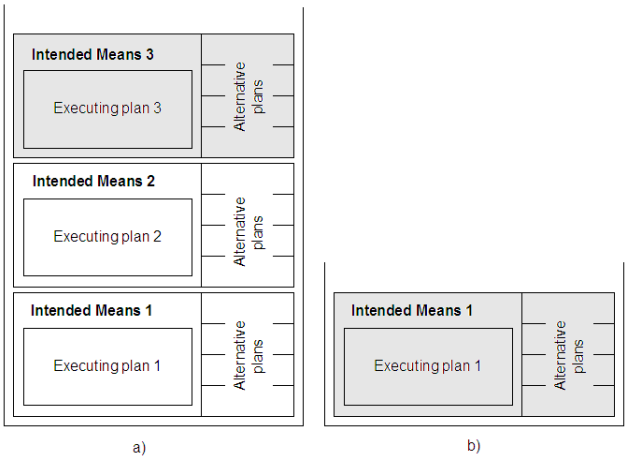

Fig. 7. Building of intended means stacks: a) the new intended means is put on top of executing stack; b) the new intended means is put in a new stack.

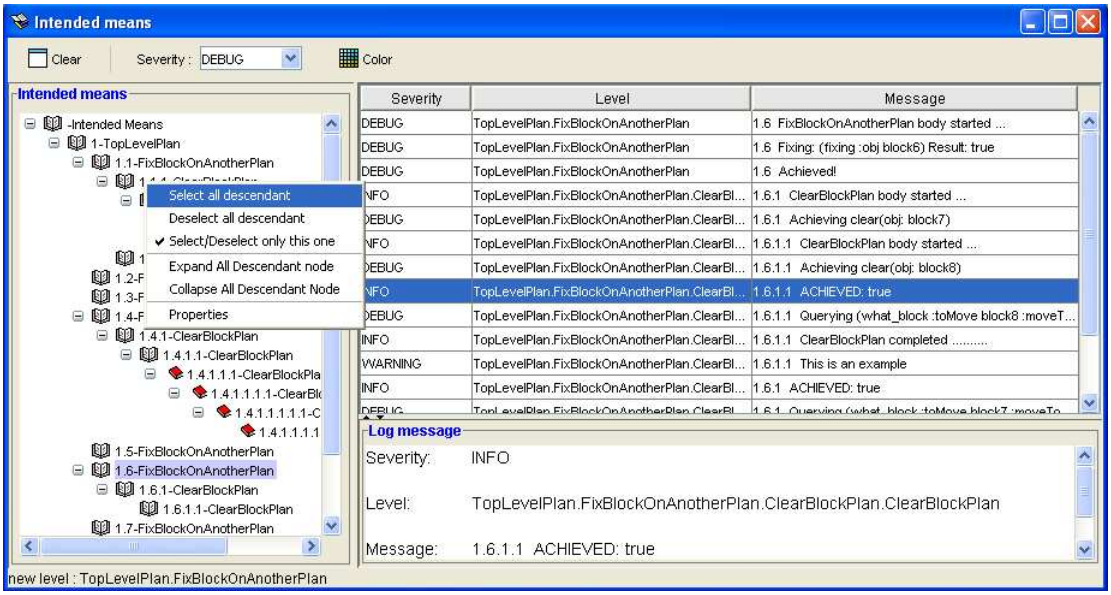

Fig. 8. Intended Means View.

acts the framework provides.
During the execution of the main plan, if the agent believes that its success condition (see table I) holds, that plan ends with success regardless its execution point and state. On the other hand, if the agent believes that the cancel condition is true, the plan ends with failure. Thus, referring to the *blocks world agent* and the *TopLevelPlan* described in figure 4, if during the execution of such a plan the required final order of blocks is achieved (for example due to some external reasons, e.g. other agents do the work of ordering the blocks), the agent will stop executing the plan, being successful in achieving its goal. Likewise, if the agent suddenly believes that the predicate $ordering$ is false, it will stop executing the plan, but in this case failing in pursuing its objectives.
Moreover, while executing the plan the agent checks the condition to be maintained (e.g. $ableToOrder(who : self)$ in figure 4): if it is believed false, the agent will try to bring it about, by desiring and possibly intending to achieve it. If the agent succeeds in doing that, the plan will continue executing, otherwise it will fail. When the plan has completed its execution, the agent will update some beliefs, according to whether the plan has succeeded or not.

It should be noted that, when the executing plan fails, since a PRACTIONIST agent has a strong commitment to handling the selected event, it selects another plan from the above-mentioned alternative practical plans and checks whether it is applicable or not. Then, if some other applicable plan exists, the agent replaces the failed plan with it (which becomes the new main plan) in the executing intended means; otherwise, the whole intended means fails and in turn the plan below fails too.

A. Executing Plans

While executing its plans, the agent will have different behaviours according to the type of acts. The plans in turn will fail as soon as one of such acts fails. Some of the possible acts a PRACTIONIST agent can manage are:

- *do an action*: if its preconditions are satisfied (at least according to the agent's beliefs), the agent executes the action with the proper inputs and gets both the outputs and the general overcome of the action (that can be *true* or *false*). If the preconditions are not satisfied or the outcome is *false*, the action fails. Finally, the effects of such an action are applied to the belief base after the execution.
  It should be noted that actions are actually executed by some effectors. Thus, the agent will search for a proper effector that is able to execute an action and then delegate the actual execution to it;
- *add* or *remove* beliefs: as soon as it is executed, the corresponding *belief updated event* is generated. This event will be then handled in the following cycles;
- *query* over the belief base according to several criterion;
- *send an ACL message* to other agents;
- *desire* to bring about some state of affairs or perform an action, both expressed as goals. The agent processes such a desire in order to figure out whether it can be promoted to an intention or not, according to the processing described in section VI-B. If it can, a corresponding internal event is created, put in the event queue, and then considered at the step (2) in one of the following cycles. Then, the corresponding intended means is built and nested inside the one that contains the above-mentioned desire invocation.
  After executing the nested intended means, if the agent does not believe that the goal has succeeded, the desire

act will fail; otherwise, the goal will be satisfied and the act will succeed.
- *wait for external messages*, by specifying an abstract structure (a template) of them. Thus, as soon as a message is received by the agent, if the message matches the template, this intended means can continue its execution;
- *wait for successful goals*, which lets the agent synchronize its activities with other intended means, but at the goal level. In other words, there is not an explicit synchronization among plans or intended means. Instead, some intended means and their executing plans can be directly synchronized with some ends (the intentions).

Several other acts are provided in PRACTIONIST in order to support the developers in actually implementing effective plans.

B. The Deliberation Process

In the PRACTIONIST framework, a goal is an objective to pursue and we refer to it as an abstraction to make the distinction between the state of affairs to be achieved and the way to achieve it. Besides, we use goals as a mean to transform desires into intentions through the satisfaction of some properties during the deliberation phase. Thus, in PRACTIONIST two families of goals were defined, as follows:

- *state goals*, which refer to some states of affairs the agent desires/intends to bring about, or cease, or preserve, or avoid. We provided PRACTIONIST agents with the capability of managing and reasoning about the following state goals: *achieve*, which represents what kind of world state to bring about; *cease* (as opposed to achieve), which represents a world state an agent wants to stop; *maintain*, which has the purpose to observe some world state and continuously re-establish this state when it does not hold; *avoid* (as opposed to maintain), which has the purpose to observe some world state and continuously prevent it;
- *perform goals*, which are not related to some world states but to some actions the agent desires/intends to perform.

In our framework states of affairs are represented by closed formulas of the FOL. Moreover, for each type of state goals, PRACTIONIST defines a success condition formula which is, in turn, defined by a modal logic closed formula that depends on its state of affairs.

We also defined the properties of *inconsistency* and *entailment* for goals. More precisely, we define a goal G_1 as *inconsistent* with a goal G_2 if and only if when G_1 succeeds, then G_2 fails. On the other hand, a goal G_1 *entails* a goal G_2 (or equivalently G_2 *is entailed by* G_1) if and only if when G_1 succeeds, then also G_2 succeeds.

These properties of goals are used by PRACTIONIST agents when reasoning about goals during the deliberation phase.

Thus, as described in section VI-A, an agent α may have the *desire* to pursue a goal G. Before committing to it, the agent will check if the goal G is inconsistent with some of current *active goals*, which are those goals the agent is already committed to. Then, if G is inconsistent with at least one of those goals, G will remain just a desire and the agent will not "intend" it. On the other hand, if the goal G is not inconsistent with active goals, then it can be promoted to an *intention*.

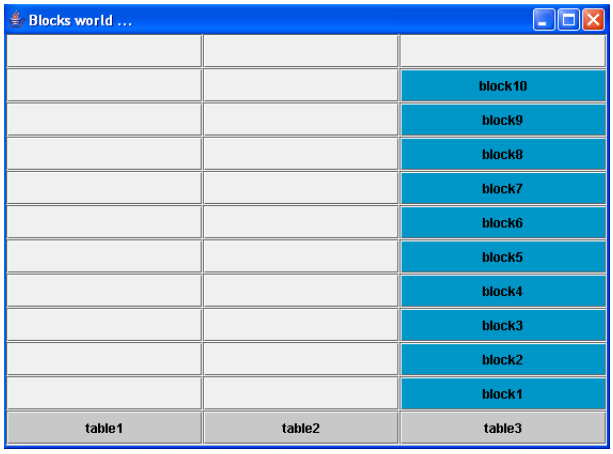

Fig. 9. The intended final situation in the blocks world problem.

But before moving to means-ends reasoning, the agent α will check if it believes that the goal G has already succeeded or if the goal G is entailed by some of current active goals. If so, there is no reason to really pursue the goal, that is the agent does not need to make any means-ends reasoning to figure out how to achieve such a goal, else the agent will try to figure out a set of plans to achieve this intention, and build the intended means, as explained in section VI.

VII. Conclusions and Future Work

In this paper we presented PRACTIONIST, a framework we have been developing for the implementation of agents according to the BDI model of agency.

PRACTIONIST implements the practical reasoning proposed by Bratmam, trying to provide developers with usable abstractions and processing capabilities. In this direction, we believe that with respect to some of existing BDI agent implementations, our approach provides a more clear separation between the ends the agent wishes/wants to bring about and the means to do it. As a matter of fact, referring to the blocks world example, the agent is mainly programmed in terms of states to achieve instead of actions to perform, according to the philosophy of our framework. The figure 9 shows the final situation of blocks (the ends) *intended* and pursued by the agent through its plans (the means).

Moreover, our framework provide a very expressive way to represent and reasoning about beliefs through modal logic formulas.

We also presented the PRACTIONIST Agent Introspection Tool (PAIT), which supports the developer in the debugging of agents according to our model. We argue that such a tool is important especially when defining BDI agents in real case scenarios and in complex environments, due to the intrinsic declarative nature of mental attitudes when compared to the adopted imperative programming languages.

However, some further work should be done with respect to the several issues that a BDI model involves. Among them, our intention is to improve the execution flow by adding some functionalities like timing, new acts, and so on, that could help in the successful application of our framework in real problems.

ACKNOWLEDGMENT

This work is partially supported by the Italian Ministry of Education, University and Research (MIUR) through the project PASAF.

REFERENCES

[1] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," in *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, J. Allen, R. Fikes, and E. Sandewall, Eds. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991, pp. 473–484. [Online]. Available: http://citeseer.nj.nec.com/rao91modeling.html

[2] M. E. Bratman, *Intention, Plans, and Practical Reason*. Cambridge, MA: Harvard University Press, 1987.

[3] G. Weiss, Ed., *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999. [Online]. Available: http://jmvidal.cse.sc.edu/library/WeissBook/

[4] M. P. Georgeff and A. L. Lansky, "Reactive reasoning and planning," in *Proc. of AAAI-87*, Seattle, WA, 1987, pp. 677–682.

[5] M. d'Inverno, D. Kinny, M. Luck, and M. Wooldridge, "A formal specification of dMARS," ser. LNAI, M. P. Singh, A. Rao, and M. J. Wooldridge, Eds., vol. 1365. Springer, 1997, pp. 155–176.

[6] J. Lee, M. J. Huber, P. G. Kenny, and E. H. Durfee, "UM-PRS: An Implementation of the Procedural Reasoning System for Multirobot Applications," in *Conference on Intelligent Robotics in Field, Factory, Service, and Space (CIRFFSS)*, Houston, Texas, 1994, pp. 842–849. [Online]. Available: citeseer.ist.psu.edu/lee94umprs.html

[7] M. J. Huber, "Jam: A bdi-theoretic mobile agent architecture." in *Agents*, 1999, pp. 236–243.

[8] M. Dastani, F. de Boer, F. Dignum, W. van der Hoek, M. Kroese, and J.-J. Meyer, "Implementing cognitive agents in 3APL," pp. 515–516.

[9] F. Bellifemine, A. Poggi, and G. Rimassa, "JADE - a FIPA-compliant agent framework," in *Proceedings of the Practical Applications of Intelligent Agents*, 1999. [Online]. Available: http://jmvidal.cse.sc.edu/library/jade.pdf

[10] L. Braubach, A. Pokahr, and W. Lamersdorf, "Jadex: A short overview," in *Main Conference Net.ObjectDays 2004*, 9 2004, pp. 195–207.

[11] P. Busetta, R. Rnnquist, A. Hodgson, and A. Lucas, "Jack intelligent agents - components for intelligent agents in java," 1999.

[12] B. F. Chellas, *Modal Logic: An Introduction*. Cambridge: Cambridge University Press, 1980.

[13] "FIPA Abstract Architecture Specification," http://www.fipa.org/specs/fipa00001/, August 2001.

[14] "Jakarta Log4J Homepage," http://jakarta.apache.org/log4j/. [Online]. Available: http://jakarta.apache.org/log4j/