$\mathcal{MINERVA}$ - A Dynamic Logic Programming Agent Architecture

João Alexandre Leite, José Júlio Alferes and Luís Moniz Pereira

Centro de Inteligência Artificial (CENTRIA)
Universidade Nova de Lisboa, 2825-114 Caparica, Portugal
{jleite | jja | lmp}@di.fct.unl.pt

Abstract. The agent paradigm, commonly implemented by means of imperative languages mainly for reasons of efficiency, has recently increased its influence in the research and development of computational logic based systems. Since efficiency is not always the crucial issue, but clear specification and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought back into the spotlight. To this accrues the recent significant improvements in the efficiency of *Logic Programming* implementations for *Non-monotonic Reasoning*.
This paper presents an overall description of $\mathcal{MINERVA}$, an agent architecture and system designed with the intention of providing a common agent framework based on the unique strengths of *Logic Programming*, to allow for the combination of several non-monotonic knowledge representation and reasoning mechanisms developed in recent years. In [7], the semantics of the multi-dimensional structure and combination of the evolving societal knowledge of agents in described and discussed in detail.

1 Introduction

The *Logic Programming (LP)* paradigm provides a well-defined, general, integrative, encompassing, and rigorous framework for systematically studying computation, be it syntax, semantics, procedures, or attending implementations, environments, tools, and standards. *LP* approaches problems, and provides solutions, at a sufficient level of abstraction so that they generalize from problem domain to problem domain. This is afforded by the nature of its very foundation in logic, both in substance and method, and constitutes one of its major assets. To this accrues the recent significant improvements in the efficiency of *Logic Programming* implementations for *Non-monotonic Reasoning* [11, 8, 12]. Besides allowing for a unified declarative and procedural semantics, eliminating the traditional wide gap between theory and practice, the use of several and quite powerful results in the field of non-monotonic extensions to *LP*, such as belief revision, inductive learning, argumentation, preferences, abduction, etc.[10, 9] can represent an important composite added value to the design of rational agents. These results, together with the improvement in efficiency, allow the referred mustering of *Logic Programming* and *Non-monotonic Reasoning* to accomplish a feliticious combination between reactive and rational behaviours of agents, the *Holy Grail* of modern *Artificial Intelligence*, whilst preserving clear and precise specification enjoyed by declarative languages.

Until recently, *LP* could be seen as a good representation language for static knowledge. If we are to move to a more open and dynamic environment, typical of the agency paradigm, we need to consider ways of representing and integrating knowledge from different sources which may evolve in time. To solve this limitation, the authors, with others, first introduced *Dynamic Logic Programming (DLP)* [1]. There, they studied and defined the declarative and operational semantics of sequences of logic programs (or dynamic logic programs). Each program in the sequence contains knowledge about some given state, where different states may, for example, represent different time periods or different sets of priorities. In [6] the *DLP* paradigm was then generalized in order to allow, not only for sequences of logic programs, but for collections of programs structured by acyclic digraphs (DAGs). By dint of such generalization, *Multi-dimensional Dynamic Logic Programming* ($\mathcal{MDLP}$) affords extra expressiveness, thereby enlarging the latitude of logic programming applications unifiable under a single framework. The generality and flexibility provided by a *DAG* ensures a wide scope and variety of new possibilities. By virtue of the newly added characteristics of multiplicity and composition, $\mathcal{MDLP}$ provides a "societal" viewpoint in Logic Programming, important in these web and agent days, for combining knowledge in general. In [7] these new possibilities are explored.

Moreover, an agent not only comprises knowledge about each state, but also knowledge about the transitions between states. The latter can represent the agent's knowledge about the environment's evolution, and its own behaviour and evolution. Since logic programs describe knowledge states, it's only fit that logic programs be utilized to describe transitions of knowledge states as well, by associating with each state a set of transition rules to obtain the next one. In [3], the authors, with others, introduce the language LUPS – "Language for dynamic updates" – designed for specifying changes to logic programs. Given an initial knowledge base (as a logic program) LUPS provides a way for sequentially updating it. The declarative meaning of a sequence of sets of update commands in LUPS is defined by the semantics of the dynamic logic program generated by those commands.

Based on the strengths of $\mathcal{MDLP}$ as a framework capable of simultaneously representing several aspects of a system in a dynamic fashion, and on those of *LUPS* as a powerful language to specify the evolution of such representations via state transitions, we have launched into the design of an agent architecture, $\mathcal{MINERVA}$. Named after the Goddess of Wisdom, this architecture is conceived with the intention of providing, on a sound theoretical basis, a common agent framework based on the strengths of Logic Programming, so as to allow the combination of several non-monotonic knowledge representation and reasoning mechanisms developed in recent years. Rational agents, in our opinion, will require an admixture of any number of those reasoning mechanisms to carrying out their tasks. To this end, a $\mathcal{MINERVA}$ agent hinges on a modular design, whereby a common knowledge base, containing all knowledge shared by more that one sub-agent, is concurrently manipulated by specialized sub-agents.

Its description here is by no means exhaustive, a task that would be impossible to accomplish within this article alone. Rather, it aims at providing an overview highlighting the role of computational logic in its several components.

2 Dynamic Logic Programming

In this section we briefly review Dynamic Logic Programming [1], its generalization $\mathcal{MDLP}$ [6], and the update command language *LUPS* [3].

The idea of dynamic updates is simple and quite fundamental. Suppose that we are given a set of generalized logic program modules P_s (possibly with negation in rule heads), structured by a DAG (or simply index by a sequence number, in [1]). Each program P_s contains some knowledge that is supposed to be true at the state s. Different states may represent different time periods or different sets of priorities or perhaps even different viewpoints. Consequently, the individual program modules may contain mutually contradictory as well as overlapping information. The role of the dynamic program update is to use the mutual relationships existing between different states (and specified by the relation in the DAG) to precisely determine, at any given state s, the *declarative* as well as the *procedural* semantics of the combined program, composed of all modules. The declarative semantics at some state is determined by the stable models of the program that consists in all those rules that are "valid" in that state. Intuitively a rule is "valid" in a state if either it belong to the state or it belong to some previous state in the DAG and is not rejected (i.e. it is inherited by a form of inertia). A rule is rejected from a less prioritary state is rejected if there is another conflicting rule in a more prioritary state with a true body. A transformational semantics, that directly provides a means for $\mathcal{MDLP}$ implementation, has also been defined[1]. For details on Dynamic Logic Programming and $\mathcal{MDLP}$ the reader is referred to [1, 6]. The paper [7], contains a more detailed review of this subject, as well as an explanation of how $\mathcal{MDLP}$ can be employed to model the combination of inter- and intra-agent societal viewpoints.

LUPS [3] is a logic programming command language for specifying updates. In can be viewed as a language that declaratively specifies how to construct a sequence of logic programs. A sentence U in *LUPS* is a set of simultaneous update commands (or actions) that, given a pre-existing sequence of logic programs (or a $\mathcal{MDLP}$), whose semantics corresponds to our knowledge at a given state (or program), produces a new $\mathcal{MDLP}$ with one more program, corresponding to the knowledge that results from the previous sequence after performing all the simultaneous commands. Different possibilities on how to connect the newly produced program with the previously extant ones give rise to different ways of combining the knowledge of the various agents. In [7] these possibilities are examined, and it is shown how to attain: equal role among agents, time prevailing and hierarchy prevailing representations. In this paper, if not otherwise stated, simply assume that the new program is concatenated immediately after the program where the command is evaluated.

A program in *LUPS* is a sequence of such sentences, and its semantics is defined by means of a dynamic logic program generated by the sequence of commands. In [3], a translation of a LUPS program into a generalized logic program is presented, where stable models exactly correspond to the semantics of the original LUPS program. This translation directly provides an implementation of LUPS (available at the URL given above).

[1] The implementation can be found in `centria.di.fct.unl.pt/~jja/updates/`

LUPS update commands specify assertions or retractions to the current program (a predefined in the $\mathcal{MDLP}$, usually the one resulting from the last update performed). In LUPS a simple assertion is represented as the command:

$$\textbf{assert } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{1}$$

meaning that if $L_{k+1}, \ldots, L_m$ holds in the current program, the rule $L \leftarrow L_1, \ldots, L_k$ is added to the new program (and persists by inertia, until possibly retracted or overridden by some future update command). To represent rules and facts that do not persist by inertia, i.e. that are one-state events, LUPS includes the modified form of assertion:

$$\textbf{assert event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{2}$$

The retraction of rules is performed with the two update commands:

$$\textbf{retract } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{3}$$

$$\textbf{retract event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{4}$$

meaning that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current program) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by **event**).

Normally assertions represent newly incoming information. Although its effects remain by inertia (until contravened or retracted), the assert command itself does not persist. However, some update commands may desirably persist in the successive consecutive updates. This is especially the case of laws which, subject to some preconditions, are always valid, or of rules describing the effects of an action. In the former case, the update command must be added to all sets of updates, to guarantee that the rule remains indeed valid. In the latter case, the specification of the effects must be added to all sets of updates, to guarantee that, whenever the action takes place, its effects are enforced. To specify such persistent update commands, *LUPS* introduces:

$$\textbf{always } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{5}$$

$$\textbf{always event } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{6}$$

$$\textbf{cancel } L \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m \tag{7}$$

The first two statements signify that, in addition to any new set of arriving update commands, the persistent update command keeps executing along with them too. The first case without, and the second case with the **event** keyword. The third statement cancels execution of this persistent update, once the conditions for cancellation are met.

3 Overall Architecture

In the $\mathcal{MINERVA}$ agent architecture, an agent consists of several specialized, possibly concurrent, sub-agents performing various tasks while reading and manipulating a common knowledge base. A schematic view of the $\mathcal{MINERVA}$ architecture is depicted in Fig. 1. The common knowledge base contains knowledge about the self and the agent community, and is conceptually divided into the following components: *Capabilities*, *Intentions*, *Goals*, *Plans*, *Reactions*, *Object Knowledge Base* and *Internal*

Behaviour Rules. There is also an internal clock. Although conceptually divided in such components, all these modules will share a common representation mechanism based on $\mathcal{MDLP}$ and *LUPS*, the former to represent knowledge at each state and *LUPS* to represent the state transitions, i.e. the common part of the agent's behaviour. Every agent is composed of specialized function related subagents, that execute their various specialized tasks. Examples of such subagents are those implementing the reactive, planning, scheduling, belief revision, goal management, learning, dialogue management, information gathering, preference evaluation, strategy, and diagnosis functionalities. These sub-agents contain a *LUPS* program encoding its behaviour, interfacing with the *Common Knowledge Base*. Whilst some of those sub-agent's functionalities are fully specifiable in *LUPS*, others will require private specialized procedures where *LUPS* serves as an interface language.

In all *LUPS* commands, both in the common knowledge base and in the sub-agents, by default the **when**-clauses are evaluated in the *Object Knowledge Base*, and the rules are added to the *Object Knowledge Base* also. More precisely, the **when**-condition of a sub-agent α_n *LUPS* command is evaluated at a special sink program of the *Object Knowledge Base*[2]. If the condition is verified, the rule is asserted (resp. retracted), by default, at the state $\alpha_{n_{c+1}}$, i.e. the sub-agent's state corresponding to the next time state. In effect, at each time state c, each sub-agent's LUPS interpreter evaluates its commands at state α' (dynamically and therefore differently at each time state) and produces its corresponding next state, $c + 1$. Whenever some literal L in a **when**-condition is to be evaluated in a program different from the default one, we denote that by $L@ProgramName$. This @ notation will also be used to denote addition of rules to programs other than the default one.

4 Common Knowledge Base

Object Knowledge Base The *Object Knowledge Base* is the main component containing knowledge about the world, i.e. knowledge at the object level and info about the society where the agent is situated.

It is represented by an evolving *MDLP*. In it there is a sequence of nodes for each sub-agent of the agent α, representing its evolution in time. There is also a sequence of nodes for every other agent in the multi-agent system, representing α's view of the evolution of those agents in time. As will be seen in the following Section, at each time state each sub-agent manipulates its corresponding state node. There is a *Dialoguer* sub-agent dealing with the interactions with other agents, and manipulating the state nodes corresponding to the agents it communicates with. Several methods are possible for combining these several sequences to form the *MDLP*'s DAG. Each corresponds to a different way of combining inter- and intra-agent social viewpoints. In [7] we show how this can be done.

Capabilities This block contains a description of the actions, and their effects, capable of being performed by the agent. *LUPS*, by allowing to declaratively specify both

[2] This special program is a sink of the DAG, and its semantics corresponds to the semantics of all the *Object Knowledge Base*, cf. [7, 6].

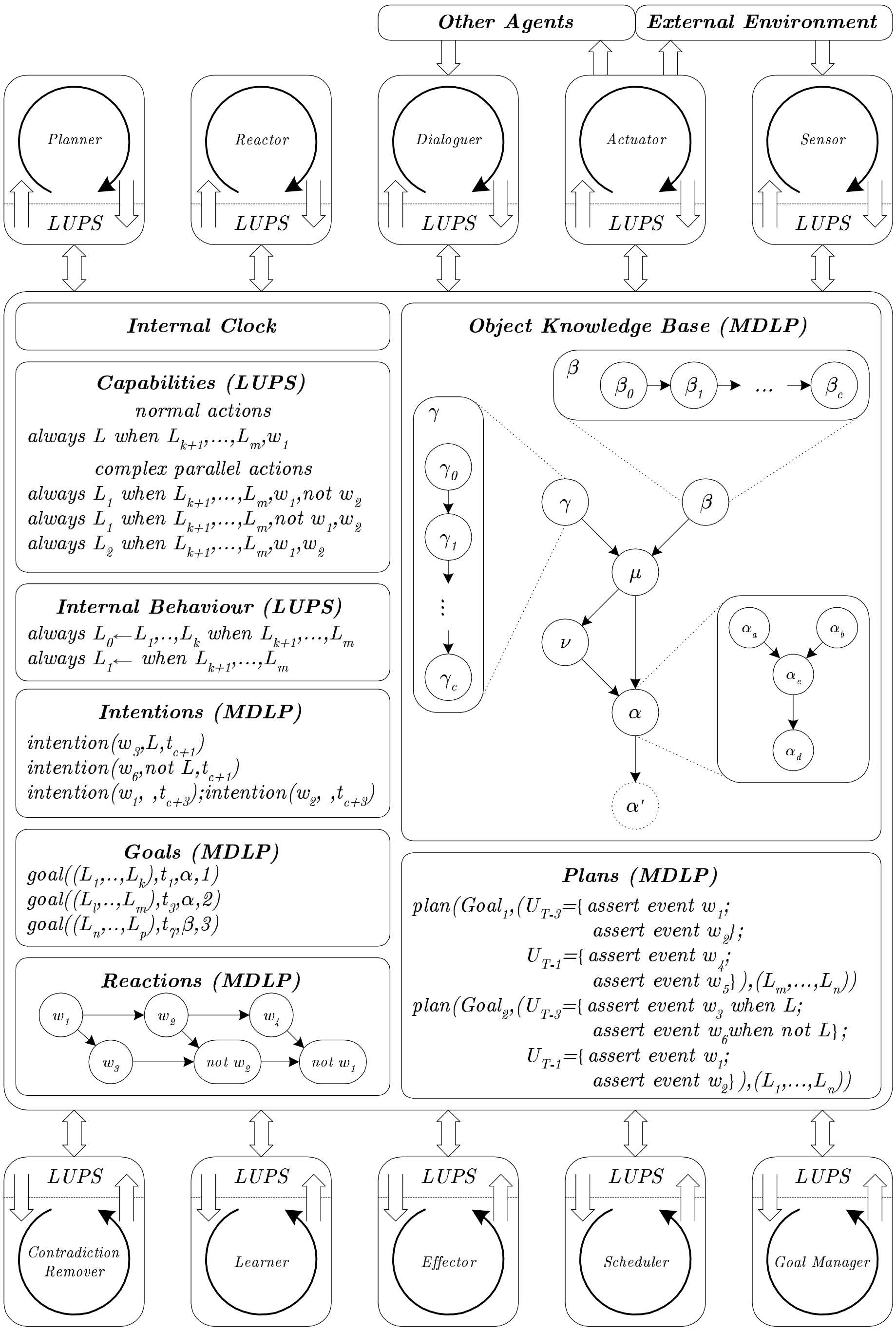

Fig. 1. The $\mathcal{MINERVA}$ agent architecture

knowledge states and their transitions, can be used as a powerful representation language for specifying actions and their effects. Its diversity of update commands can model from the simplest condition-effect action to parallel actions and their coupled indirect effects. Typically, for every *action* ω, there will be one (or more) *LUPS* commands, where the action name ω appears in the '$\mathbf{when}$' clause. For example, in

$$\mathbf{always}\ L \leftarrow L_1, \ldots, L_k\ \mathbf{when}\ L_{k+1}, \ldots, L_m, \omega$$

we have, intuitively, that ω is an action whose preconditions are $L_{k+1}, \ldots, L_m$ and whose effect is an update that, according to its type, can model different kinds of actions, all in one unified framework. Examples of kinds of actions are:

– actions of the form "ω $\mathbf{causes}$ F $\mathbf{if}$ $F_1, \ldots, F_k$", where $F, F_1, \ldots, F_k$ are fluents (such as in the language $\mathcal{A}$ of [5]) translates into the update command

$$\mathbf{always}\ F\ \mathbf{when}\ F_1, \ldots, F_k, \omega$$

– actions whose epistemic effect is a rule update of the form "ω $\mathbf{updates\ with}$ $L \leftarrow L_1, \ldots, L_k$ $\mathbf{if}$ $L_{k+1}, \ldots, L_m$" translates into the update command

$$\mathbf{always}\ L \leftarrow L_1, \ldots, L_k\ \mathbf{when}\ L, \ldots, L_m, \omega$$

– actions that, when performed in parallel, have different outcomes, of the form "ω_a $\mathbf{or}$ ω_b cause L_1 $\mathbf{if}$ $L_{k+1}, \ldots, L_m$" and "ω_a $\mathbf{and}$ ω_b $\mathbf{in\ parallel\ cause}$ L_2 $\mathbf{if}$ $L_{k+1}, \ldots, L_m$" translate into the three update commands:

$$\begin{array}{l}\mathbf{always}\ L_1\ \mathbf{when}\ L_{k+1}, \ldots, L_m, \omega_a, \mathit{not}\, \omega_b \\ \mathbf{always}\ L_1\ \mathbf{when}\ L_{k+1}, \ldots, L_m, \mathit{not}\, \omega_a, \omega_b \\ \mathbf{always}\ L_2\ \mathbf{when}\ L_{k+1}, \ldots, L_m, \omega_a, \omega_b\end{array}$$

Other types of actions (e.g. action with non-deterministic effects) can also be translated into LUPS commands. For lack of space these are not shown here.

Internal Behaviour Rules The *Internal Behaviour Rules* are *LUPS* commands that specify the agent's reactive internal epistemic state transitions. They are of the form:

$$\mathbf{assert}\ L \leftarrow L_1, \ldots, L_k\ \mathbf{when}\ L_{k+1}, \ldots, L_m$$

whose effect is the immediate asserting of rule $L \leftarrow L_1, \ldots, L_k$ in the *Object Knowledge Base* if $L_{k+1}, \ldots, L_m$ holds now. These commands will be executed by one of the sub-agents (the *Reactor*), and they are quite common since they may be used by more than one of the sub-agents. E.g. the *Planner* sub-agent must be aware of these rules when planning for goals. An example of internal behaviour rules would be:

$$\begin{array}{l}\mathbf{assert}\ \mathit{jail}(X) \leftarrow \mathit{abortion}(X)\ \mathbf{when}\ \mathit{government}(\mathit{republican}) \\ \mathbf{assert}\ \mathit{not}\, \mathit{jail}(X) \leftarrow \mathit{abortion}(X)\ \mathbf{when}\ \mathit{government}(\mathit{democrat})\end{array}$$

stating that whenever the government is republican, an abortion implies going to jail, and whenever the government is democrat, an abortion does not imply going to jail.

Goals The *Goals* structure is a DLP where at each state the program contains facts of the form *goal* $(Goal, Time, Agent, Priority)$ representing the goals that are to be achieved by the agent. $Goal$ is a conjunction of literals, $Time$ refers to a time state, $Agent$ represents the origin of the goal and $Priority$ contains the priority of the goal. *Goals* is a dynamic structure in the sense that new goals can be asserted onto this structure by any sub-agent, some of them originating in another one. They will be manipulated by a *Goal Manager* sub-agent that can assert, retract and change a goal's priority.

Plans As shown in [2], *LUPS*, together with abduction, can be used to represent and solve planning problems. There, the notion of *action update*, U^{ω}, is defined as a set of update commands of the form $U^{\omega} = \{\textbf{assert event } \omega\}$ where ω is an action name. Intuitively, each command of the form "$\textbf{assert event } \omega$" represents the performing of action ω. Note that performing an action itself is something that does not persist by inertia. Thus, according to the description of *LUPS*, the assertion must be of an event. By asserting $\textbf{event } \omega$, the effect of the action will be enforced if the preconditions are met. Each *action update* represents a set of simultaneous actions. Plans are *LUPS* programs composed of action updates. An example of a plan to achieve a $Goal_1$ at time T would be:

$$U_{T-3} = \{\textbf{assert event } \omega_1; \textbf{assert event } \omega_2\}$$
$$U_{T-1} = \{\textbf{assert event } \omega_4; \textbf{assert event } \omega_5\}$$

meaning that in order to achieve $Goal_1$ at time T, the agent must execute actions ω_1 and ω_2 at time $T-3$ and actions ω_4 and ω_5 at time $T-1$. Also, associated with each plan, there is a set of preconditions ($Conditions$) that must be met at the beginning of the plan. Each plan ($Plan$) for goal ($Goal$) is generated by a planner sub-agent, which asserts them into the *Common Knowledge Base* in the form $plan\ (Goal, Plan, Conditions)$ where they are kept for future reuse.

By permiting plans to be represented in the *LUPS* language, we allow the use of more complex planners, namely conditional planners. Conditional action updates are to be represented by *LUPS* commands of the form:

$$\textbf{assert event } \omega \textbf{ when } L_{k+1}, \ldots, L_m$$

An example of such a conditional plan, for $Goal_2$ would be:

$$U_{T-3} = \{\textbf{assert event } \omega_3 \textbf{ when } L; \textbf{assert event } \omega_6 \textbf{ when } \textit{not } L\}$$
$$U_{T-2} = \{\textbf{assert event } \omega_1; \textbf{assert event } \omega_2\}$$

meaning that in order to achieve $Goal_2$ at time T, the agent must execute, at time $T-3$, the command $\textbf{assert event } \omega_3$ if L holds at time $T-3$, otherwise, if *not* L holds, it must execute the command $\textbf{assert event } \omega_6$. Then, it should execute the commands $\textbf{assert event } \omega_1$ and $\textbf{assert event } \omega_2$ at time $T-2$. These conditions, in the $\textbf{when}$ statement of these commands, can also be employed to include the verification of the success of previous actions.

The plans stored in the common knowledge are going to be processed, together with the reactions, by a specialized *Scheduler* sub-agent that will avail itself of them to determine the intentions of the agent.

Reactions The *Reactions* block at the *Common Knowledge Base* is represented by a *very simple MDLP* whose object rules are just facts denoting actions ω or negation of actions names $not\,\omega$. The *MDLP* contains a sequence of nodes for every sub-agent capable of reacting, and a set of edges representing a possible hierarchy of reactions. Those sub-agents will contain a set of *LUPS* commands of the form

$$\mathbf{assert}\ \omega\ \mathbf{when}\ L_{k+1},\ldots,L_m$$

Whenever $L_{k+1},\ldots,L_m$ is true, action ω should be reactively executed. This corresponds to the assertion of ω in the corresponding node of the *Reactions* module. These, together with the plans, will be processed by the *Scheduler* sub-agent to transform them into intentions. The richness of the *LUPS* language enables us to express action blockage, i.e. under some conditions, preventing some action ω to be executed by the agent, possibly proposed by another sub-agent. This can be expressed by the command:

$$\mathbf{assert}\ not\,\omega\ \mathbf{when}\ L_{k+1},\ldots,L_m$$

whose effect would be the assertion of $not\,\omega$. This assertion would block any ω proposed by a lower ranked reactive sub-agent. By admitting the buffering of reactions, i.e. they are first proposed and then scheduled, this architecture allows for some conceptual control over them, namely by preventing some undesirable reactions being performed.

Intentions *Intentions* are the actions the agent has committed to. A *Scheduler* sub-agent compiles the plans and reactions to be executed into the intentions, according to the resources of the agent. Control over the level of reactivity and deliberation can be achieved by this process. The *Intentions* structure is a *DLP* where the object language rules are simply facts of the form:

$$intention\,(Action, Conditions, Time)$$

where $Action \in K_\omega$, $Conditions$ is a conjunction of literals from L, and $Time \in T$. They can be interpreted as: execute $Action$, at time T, if $Conditions$ is true. It is left to the Actuator sub-agent to execute the intentions. For example, to achieve $Goal_2$ and $Goal_1$, the two plans previously exemplified can be compiled into the set of intentions (where c is the current time state):

$$\begin{array}{lll} intention\,(\omega_3, L, c+1)\,; & intention\,(\omega_1, \_, c+3)\,; & intention\,(\omega_4, \_, c+5) \\ intention\,(\omega_6, not\,L, c+1)\,; & intention\,(\omega_2, \_, c+3)\,; & intention\,(\omega_5, \_, c+5) \end{array}$$

Although typically intentions are internal commitments, nothing prevents us from having a specialized sub-agent manage (and possibly retract from) the set of intentions.

5 Sub-Agents

The sub-agents constitute the labour part of the $\mathcal{MINERVA}$ architecture. Their tasks reside in the evaluation and manipulation of the *Common Knowledge Base* and, in some cases, in the interface with the environment in which the agent is situated. They differ essentially in their specialities, inasmuch as there are planners, reactors, schedulers, learners, dialoguers, etc. Conceiving the architecture based on the notion of, to some extent, independent sub-agents attain a degree of modularity, appropriate in what concerns the adaptability of the architecture to different situations.

Each of these sub-agents contains a *LUPS* program encoding its behaviour, interfacing with the *Common Knowledge Base* and possibly with private specialized procedures. Conceptually, each of these sub-agents also comprises a *LUPS* meta-interpreter that executes its *LUPS* program and produces a sequence of states in the structures of the Common Knowledge Base. The collection of all such sequences of states, produced by all the sub-agents will come to constitute the states of the *Object Knowledge Base*.

To permit private procedure calls, we extend LUPS by allowing those to be called in the **when** statement of a command:

$$\textbf{assert } X \leftarrow L_1, \ldots, L_k \textbf{ when } L_{k+1}, \ldots, L_m, ProcCall(L_{m+1}, \ldots, L_n, X)$$

In this case, if $L_{k+1}, \ldots, L_m$ is true at state α', then $ProcCall(L_{m+1}, \ldots, L_n, X)$ is executed and, if successful, X is instantiated and rule $X \leftarrow L_1, \ldots, L_k$ is asserted into the sub-agent's state corresponding to the next time state. These procedure calls can read from the *Common Knowledge Base*, but cannot change it. All changes to the Common Knowledge Base are performed via *LUPS* commands, ensuring a clear and precise semantics of the overall system.

It is important to stress that *LUPS* constitutes a language to specify state transitions, namely by determining the construction of the programs that define each state, whose meaning is assigned by the semantics of $\mathcal{MDLP}$. Some of the tasks to be performed by the sub-agents need to resort to procedure calls, but others are fully specifiable with *LUPS* commands that simply consult the Common Knowledge Base.

Sensor The *Sensor* sub-agent receives input from the environment through procedure calls of the form $SensorProc(Rule)$. The information, in the form of logic program rules, is asserted in the object knowledge base. This behaviour is accomplished with *LUPS* commands of the form:

$$\textbf{assert } Rule \textbf{ when } SensorProc(Rule)$$

The **when** statement can also be used as a filter, possibly specifying conditions for the acceptance of the input. Mark that since there is no explicit reference as where to execute the command, the assertions are, by default, performed at the *Sensor*'s node at the *Object Knowledge Base*.

Dialoguer The Dialoguer sub-agent is similar to the *Sensor* one in what concerns receiving input from the environment. It differs in that it processes incoming messages from the other agents in the multi-agent system ($\beta, \gamma, \mu, \ldots$), and the *LUPS* commands are then executed by asserting the information into the corresponding agent's node. Moreover, it is this sub-agent's charge, according to the message, to generate new goals, issue replies, etc. Examples of *LUPS* commands typical of this sub-agent would be:

$$\begin{array}{l}
\textbf{assert } goal(Goal, Time, Agent, _)@goals \\
\qquad\qquad \textbf{when } MsgFrom(Agent, Goal, Time, Rule), cooperative(Agent). \\
\textbf{assert } Rule@Agent \textbf{ when } MsgFrom(Agent, Goal, Time, Rule). \\
\textbf{assert } msgTo(Agent, Goal, plan(Goal, Plan, Cond))@reactions \\
\qquad\qquad \textbf{when } goal(Goal, Time, Agent, _)@goals, Agent \neq \alpha, \\
\qquad\qquad\qquad\qquad plan(Goal, Plan, Cond)@plans.
\end{array}$$

These commands specify a behaviour for the acceptance of goals and beliefs from another agent, and for dealing with the request. The first command insists that a goal should be asserted into the *Goals* of the *Common Knowledge Base* whenever a message from a cooperative agent requests it. The second asserts the rules communicated by another agent into its corresponding node of the *Object Knowledge Base*. The third issues a communication action, by asserting it into the *Reactions*, to be scheduled, whenever there is a request from another agent to solve a goal, and there is a plan for such a goal.

Actuator The Actuator sub-agent interfaces with the environment by executing actions on it. At each cycle, the *Actuator* extracts the actions to be executed from the *Intentions* and performs them on the outside world. If successful, it event asserts the action name, ω, in the *Object Knowledge Base*. At each cycle, the *Actuator* executes LUPS commands of the form:

$$\begin{array}{l}\textbf{assert event}\ \omega\ \textbf{when}\ intention\,(Action, Cond, Time)\,@intentions, \\ \qquad\qquad\qquad\qquad Current(Time), Cond, ActionProc(\omega)\end{array}$$

corresponding to the execution of the *Intentions* scheduled for the current time state.

Effector The Effector, at each cycle, evaluates and executes the LUPS commands in the Capabilities and Common behaviour Rules. Note that, although the *Capabilities* and *Common Behaviour Rules* constitute the *LUPS* program of the *Effector* sub-agent, they belong to the *Common Knowledge Base* because they may be accessed by other sub-agents, namely by the Planner. The *Capabilities* require the prior successful execution of an action. This does not happen with the *Common Behaviour Rules*. These specify state transitions solely based on the internal state of the agent.

Reactor The *Reactor* will contain reactive rules that, if executed, produce an action to be performed. They are of the form:

$$\textbf{assert event}\ \omega@reactions\ \textbf{when}\ L_1, \ldots, L_k$$

An example of such a reaction would be

$$\textbf{assert event}\ runaway@reactions\ \textbf{when}\ danger$$

where $runaway$ is the name of an action. We can also reactively and temporarily block actions by using *LUPS* commands of the form:

$$\textbf{assert event}\ not\,\omega@reactions\ \textbf{when}\ L_1, \ldots, L_k$$

Planner In [2], it was shown how planning can be achieved by means of abduction in LUPS specified scenarios. Such an abduction based planner uses the *Object Knowledge Base* together with the *Intentions* (actions already set to be executed in the future), *Capabilities*, and the *Common Behaviour Rules*, to achieve a plan for a given goal, consistent with the current set of intentions. A *LUPS* command for the *Planner*, with such an abductive planner represented by the procedure call $AbductivePlan(Goal, Plan, Cond)$, would be:

$$\begin{array}{l}\textbf{assert}\ plan\,(Goal, Plan, Cond)\,@plans \\ \qquad \textbf{when}\ goal\,(Goal, T, \_, 1)\,@goals,\ AbductivePlan(Goal, Plan, Cond)\end{array}$$

Other planners, possibly more efficient and/or complex can be used, namely partial planners, with the appropriate *LUPS* commands to interface them.

Goal Manager Goals can be asserted by other sub-agents, some of them originating in other agents. These goals may conflict with one another. The task of the *Goal Manager* is to deal with such issues. It manipulates the *Goals* structure, being able to merge goals, set priorities, delete goals, etc. An example of a command for this sub-agent, specifying that if two goals are incompatible, the one with lower priority should be removed, is:

$$\begin{array}{l} \mathbf{retract}\ goal\,(G_1, T_1, A_1, P_1)\,@goals \\ \quad \mathbf{when}\ \ goal\,(G_1, T_1, A_1, P_1)\,@goals, goal\,(G_2, T_2, A_2, P_2)\,@goals, \\ \qquad\qquad incomp(G_1, G_2), P_1 < P_2 \end{array}$$

Scheduler The *Scheduler* determines the Intentions of the agent, based on the current state of the agent. The Scheduler acts whenever there are pending reactions or goals and plans. More than one specialized scheduling procedure may exist. For example, if there were three procedures, depending on the need to combine reactions and plans or just to schedule one of them, the following LUPS commands would select their execution:

$$\begin{array}{ll} \mathbf{assert}\ \Pi @intentions\ \mathbf{when} & goal\,(G, \_, \_, \_)\,@goals, plan(G, \_, \_)@plans, \\ & not\ X\,@Reactions, SchedulePlans(\Pi). \\ \mathbf{assert}\ \Pi @intentions\ \mathbf{when} & not\ goal\,(\_, \_, \_, \_)\,@goals, X\,@reactions, \\ & ScheduleReactions(\Pi). \\ \mathbf{assert}\ \Pi @intentions\ \mathbf{when} & goal\,(G, \_, \_, \_)\,@goals, plan(G, \_, \_)@plans, \\ & X\,@reactions, CombineSchedule(\Pi). \end{array}$$

Other Sub-agents Other sub-agents can concurrently exist in the $\mathcal{MINERVA}$ architecture. Fig. 1 mentions a *Contradiction Remover* and a *Learner* sub-agent, but many others can be specified. Either fully specified by means of *LUPS* commands or with *LUPS* serving simply as an interface to procedure calls or legacy code, a great variety of sub-agents can be fully integrated into this architecture. If tight control is needed, *LUPS* also encompasses the specification of control sub-agents. For examples of the application of *LUPS* in several domains the reader is referred to [4].

6 Conclusions

We began this article by extolling the virtue and the promise of Computational Logic, most adeptly in the guise of Logic Programming, for adumbrating the issues of, and the solutions for, an integrated multi-agent architecture with a precisely defined declarative semantics. As we near the end of our proctrated exposition, we are confident to have brought the point home. To wit, our architectural scaffolding rests on a sound and wide basis, whose core are the evolving updatable agents and their attending, and multi-dimensionally configured, knowledge bases.

The basic architecture affords us too, we purport to have shown, with the elasticity and resilience to further support a spate of crucial ancillary functionalities, in the form of additional specialized agents, via compositional, communication, and procedural mechanisms. The circum-exploration of the territories under purview has hardly started, but the means of locomotion appear to be already with us.

Logic Programming is, without a doubt, the privileged melting pot for the articulate integration of functionalities and techniques, pertaining to the design and mechanization of complex systems, addressing ever more demanding and sophisticated computational abilities. For instance, consider again those reasoning abilities mentioned in previous sections. Forthcoming rational agents, to be realistic, will require an admixture of any number of them to carrying out their tasks. No other computational paradigm affords us with the wherewithal for their coherent conceptual integration. And, all the while, the very vehicle that enables testing its specification, when not outright its very implementation.

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000. A short version titled *Dynamic Logic Programming* appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, and P. Quaresma. Planning as abductive updating. In D. Kitchin, editor, *Proceedings of the AISB'00 Symposium on AI Planning and Intelligent Agents*, pages 1–8. AISB, 2000.
3. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 2001. To appear. A short version appeared in M. Gelfond, N. Leone and G. Pfeifer (eds.), LPNMR-99, LNAI 1730, Springer.
4. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. An exercise with dynamic logic programming. In L. Garcia and M. Chiara Meo, editors, *Proceedings of the 2000 Joint Conference on Declarative Programming (AGP-00)*, 2000.
5. M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and information Science*, 3(16), 1998.
6. J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic logic programming. In F. Sadri and K. Satoh, editors, *Proceedings of the CL-2000 Workshop on Computational Logic in Multi-Agent Systems (CLIMA'00)*, pages 17–26, 2000.
7. J. A. Leite, J. J. Alferes, and L. M. Pereira. MINERVA – combining societal agents knowledge. Technical report, Dept. Informática, Universidade Nova de Lisboa, 2001. Available at `http://centria.di.fct.unl.pt/˜jleite/`.
8. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In J. Dix, U. Furbach, and A. Nerode, editors, *LPNMR-97*, volume 1265 of *LNAI*, pages 420–429, Berlin, July 28–31 1997. Springer.
9. S. Rochefort, F. Sadri, and F. Toni, editors. *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*, Las Cruces, New Mexico, USA, 1999. Available from `http://www.cs.sfu.ca/conf/MAS99`.
10. F. Sadri and F. Toni. Computational logic and multiagent systems: A roadmap, 1999. Available from `http://www.compulog.org`.
11. D. De Schreye, M. Hermenegildo, and L. M. Pereira. Paving the roadmaps: Enabling and integration technologies, 2000. Available from `http://www.compulog.org/net/Forum/Supportdocs.html`.
12. XSB-Prolog. The XSB logic programming system, version 2.0, 1999. Available at `http://www.cs.sunysb.edu/ sbprolog`.