$\mathcal{MINERVA}$ - Combining Societal Agents Knowledge

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. This paper explores the applicability of the new paradigm of *Multi-dimensional Dynamic Logic Programming* to represent an agent's view of the combination of societal knowledge dynamics. The representation of a dynamic society of agents is the core 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*, and allows the combination of several non-monotonic knowledge representation and reasoning mechanisms developed in recent years. An overall description of $\mathcal{MINERVA}$ can be found in this paper's sequel [11].

1 Introduction

Over recent years, the notion of agency has claimed a major role in defning the trends of modern research. Infuencing a broad spectrum of disciplines such as Sociology, Psychology, among others, the agent paradigm virtually invaded every sub-feld of Computer Science [6, 9, 17]. Although commonly implemented by means of imperative languages, mainly for reasons of effciency, the agent concept has recently increased its infuence in the research and development of computational logic based systems. Since effciency is not always the crucial issue, but clear specifcation and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought back into the spotlight.

The *Logic Programming* paradigm provides a well-defned, 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 suffcient 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 signifcant improvements in the effciency of *Logic Programming* implementations for *Non-monotonic Reasoning* [15, 18]. Besides allowing for a unifed declarative and procedural semantics, eliminating the traditional wide gap between theory and practice, the use of several and quite powerful results in the feld of non-monotonic extensions to *Logic Programming (LP)*, such as belief revision, inductive learning, argumentation, preferences, abduction, etc.[17] can represent an important composite added value to the design of rational agents.

Until recently, *Logic Programming* 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. Moreover, an agent not only comprises knowledge about each states, but also some form of knowledge about the transitions between states. This knowledge about state transitions can represent the agent's knowledge about the environment's evolution, as well as its own behaviour and evolution. Since logic programs describe knowledge states, it's only £t that logic programs describe transitions of knowledge states as well. It is natural to associate with each state a set of transition rules to obtain the next state. Recent developments have opened *Logic Programming* to these otherwise unreachable dynamic worlds.

In [2], the authors, with others, introduced *Dynamic Logic Programming*. There, they studied and de£ned 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. The introduction of *Dynamic Logic Programming* has extended Logic Programming, making possible for a logic program to undergo a sequence of modi£cations, opening up the possibility of incremental design and evolution of logic programs, therefore signi£cantly facilitating *modularization* of logic programming and, thus, modularization of non-monotonic reasoning as a whole.

In [5], the authors, with others, introduced 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, unifying states and state transitions into a single declarative logic based framework.

Even though the main motivation behind the introduction of *Dynamic Logic Programming* was to represent the evolution of knowledge in time, the relationship between the different states can encode other aspects of a system, as explored in [2, 10, 7, 16, 3, 13, 4]. Although *Dynamic Logic Programming* can represent several states in one evolving dimension or aspect of a system, no more than one such aspectual evolution can be encoded and combined simultaneously. This is so because *Dynamic Logic Programming* is de£ned only for linear sequences of states. *Multi-dimensional Dynamic Logic Programming* ($\mathcal{MDLP}$) [12] was introduced to generalize *DLP* to allow for collections of states represented by arbitrary acyclic digraphs (*DAG*), not just sequences of states. $\mathcal{MDLP}$ assigns semantics to sets and subsets of logic programs, depending on how they stand in relation to one another, as de£ned by the *DAG* that represents the states and their con£guration. By dint of such natural generalization, $\mathcal{MDLP}$ affords extra expressiveness, thereby enlarging the latitude of logic programming applications uni£able under a single framework. The ¤exibility 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 this paper we explore the application of the new paradigm of *Multi-dimensional Dynamic Logic Programming* to represent an agent's view of the combination of societal knowledge dynamics, i.e. the agent's view of the evolution of its knowledge as a result of knowledge evolution in the community of agents. In [11], we describe its role in the $\mathcal{MINERVA}$ architecture, together with the use of *LUPS* as a language to specify the agent's behaviour.

We begin with a brief overview of DLP in Sect. 2. In Sect. 3, we present $\mathcal{MDLP}$. In Sect. 4 we explore the application of $\mathcal{MDLP}$ to represent inter and intra-agent relationships and their views of a multi-agent system. We then conclude in Sect. 6.

2   Background

We start with an overview of the syntax and semantics of generalized logic programs, followed by a short recap of the paradigm of *Dynamic Logic Programming*.

Generalized Logic Programs and their Stable Models To represent *negative* information in logic programs and in their updates, since we need to allow default negation $not\,A$ not only in premises of their clauses but also in their heads, we use *generalized logic programs* as de£ned in [2][1]. By a *generalized logic program P* in a language $\mathcal{L}$ we mean a £nite or in£nite set of propositional clauses of the form $L_0 \leftarrow L_1, \ldots, L_n$ where each L_i is a literal (i.e. an atom A or the default negation of an atom $not\,A$). If r is a clause (or rule), by $H(r)$ we mean L, and by $B(r)$ we mean $L_1, \ldots, L_n$. If $H(r) = A$ (resp. $H(r) = not\,A$) then $not\,H(r) = not\,A$ (resp. $not\,H(r) = A$). By a (2-valued) *interpretation* M of $\mathcal{L}$ we mean any set of literals from $\mathcal{L}$ that sat-is£es the condition that for any A, *precisely one* of the literals A or $not\,A$ belongs to M. Given an interpretation M we de£ne $M^+ = \{A : A$ is an atom, $A \in M\}$ and $M^- = \{not\,A : A$ is an atom, $not\,A \in M\}$. Following established tradition, wherever convenient we omit the default (negative) atoms when describing interpretations and models. We say that a (2-valued) interpretation M of $\mathcal{L}$ is a stable model of a generalized logic program P if $r(M) = least\,(r(P) \cup r(M^-))$, where $r(.)$ univocally renames every default literal $not\,A$ in a program or model into new atoms, say not_A. The class of generalized logic programs can be viewed as a special case of yet broader classes of programs, introduced earlier in [14], and, for the special case of normal programs, their semantics coincides with the stable models semantics [8].

Dynamic Logic Programming Next we recall the semantics of *dynamic logic programming* [2]. A dynamic logic program is a sequence $P_0 \oplus \ldots \oplus P_n \oplus \ldots$ (also denoted by $\bigoplus \mathcal{P}$, where $\mathcal{P} = \{P_s : s \in S\}$ is a £nite or in£nite sequence of LPs, indexed by the £nite or in£nite set $S = \{1, 2, \ldots, n, \ldots\}$. Such sequence may be viewed as the outcome of updating P_0 with P_1, ..., updating it with P_n,... As we will see in the following sections, each P_i is determined by the ith state transition. The role of dynamic logic programming is to ensure that these newly added rules are in force, and that previous rules are still valid (by inertia) for as long as they do not con¤ict with more recent ones, whenever the latter remain in force themselves. The notion of dynamic logic program at state s, denoted by $\bigoplus_s \mathcal{P}$, characterizes the meaning of the dynamic logic program when queried at state s, by means of its stable models, de£ned as follows:

---

[1] In [5] the reader can £nd the motivation for the usage of generalized logic programs, instead of using simple denials as a result of freely moving the head not s into the body.

Definition 1 (Stable Models of DLP). *Let $\bigoplus \mathcal{P} = \bigoplus \{\, P_s : s \in S \}$ be a dynamic logic program, let $s \in S$. An interpretation M_s is a* stable model *of $\bigoplus \mathcal{P}$ at state s iff M_s is a stable model of $\mathcal{P}_s - Reject(s, M_s)$ where:*

$\mathcal{P}_s = \bigcup_{i \leq s} P_i$
$Reject(s, M_s) = \{ r \in P_i : \exists r' \in P_j, i < j \leq s, H(r) = not\ H(r') \land M_s \vDash B(r') \}$

3   Multi-dimensional Dynamic Logic Programming

Even though the main motivation behind the introduction of *DLP* was to represent the evolution of knowledge in time, the relationship between the different states can encode other aspects of a system, as pointed out in [2]. In fact, since its introduction, *DLP* (and *LUPS*) has been employed to represent a stock of features of a system, namely as a means to: represent and reason about the evolution of knowledge in time [2]; combine rules learnt by a diversity of agents [10]; reason about updates of agents' beliefs [7]; model agent interaction [16]; model and reason about actions [3]; resolve inconsistencies in metaphorical reasoning [13]; integrate updates and preferences [4].

The common feature among these applications of *DLP* is that the states associated with the given set of theories encode only one of several possible representational dimensions (e.g. time, hierarchies, domains,...), inasmuch *DLP* is defined for linear sequences of states alone. For example, *DLP* can be used to model the relationship of a strict hierarchy group of agents, and *DLP* can be used to model the evolution of a single agent over time. But *DLP*, as it stands, cannot deal with both settings at once, and model the evolution of one such group of agents over time.

In effect, knowledge updating is not to be simply envisaged as taking place in the time dimension alone. Several updating dimensions may combine simultaneously, with or without the temporal one, such as specificity (as in taxonomies), strength of the updating instance (as in the legislative domain), hierarchical position of knowledge source (as in organizations), credibility of the source (as in uncertain, mined, or learnt knowledge), or opinion precedence (as in a society of agents). For this to be possible, *DLP* needs to be extended to allow for a more general structure of states.

In this section we present the notion of *Multi-dimensional Dynamic Logic Programming ($\mathcal{MDLP}$)* (introduced in [12]) which generalizes *DLP* to allow for collections of states represented by arbitrary acyclic digraphs. In this setting, $\mathcal{MDLP}$ assigns semantics to sets and subsets of logic programs, depending on how they relate to one another, these relations being defined by the acyclic digraph representing the states.

Graphs   A *directed graph*, or *digraph*, $D = (V, E)$ is a pair of two finite or infinite sets $V = V_D$ of *vertices* and $E = E_D$ of pairs of vertices or *(directed) edges*. A *directed edge sequence from v_0 to v_n* in a digraph is a sequence of edges $e_1, e_2, ..., e_n \in E_D$ such that $e_i = (v_{i-1}, v_i)$ for $i = 1, ..., n$. A *directed path* is a directed edge sequence in which all the edges are distinct. A *directed acyclic graph*, or *acyclic digraph (DAG)*, is a digraph D such that there are no directed edge sequences from v to v, for all vertices v of D. A *source* is a vertex with in-valency 0 (number of edges for which it is a final vertex) and a *sink* is a vertex with out-valency 0 (number of edges for

which it is an initial vertex). We say that $v < w$ if there is a directed path from v to w and that $v \leq w$ if $v < w$ or $v = w$. The relevancy DAG of a DAG D wrt a vertex v of D is $D_v = (V_v, E_v)$ where $V_v = \{v_i : v_i \in V \text{ and } v_i \leq v\}$ and $E_v = \{(v_i, v_j) : (v_i, v_j) \in E \text{ and } v_i, v_j \in V_v\}$. The relevancy DAG of a DAG D wrt a set of vertices S of D is $D_S = (V_S, E_S)$ where $V_S = \bigcup_{v \in S} V_v$ and $E_S = \bigcup_{v \in S} E_v$, where $D_v = (V_v, E_v)$ is the relevancy DAG of D wrt v.

3.1 Declarative Semantics

We start by defining the framework consisting of the generalized logic programs indexed by a DAG. Throughout this paper, we will restrict ourselves to DAG's such that for every vertex v of the DAG, any path ending in v is finite.

Definition 2 (Multi-dimensional Dynamic Logic Program). *Let $\mathcal{L}$ be a propositional language. A* Multi-dimensional Dynamic Logic Program (MDLP), $\mathcal{P}$, *is a pair* $(\mathcal{P}_D, D)$ *where* $D = (V, E)$ *is a DAG and* $\mathcal{P}_D = \{P_v : v \in V\}$ *is a set of generalized logic programs in the language $\mathcal{L}$, indexed by the vertices $v \in V$ of D. We call* states *such vertices of D. For simplicity, we often leave the language $\mathcal{L}$ implicit.*

To characterize the models of $\mathcal{P}$ at any given state we will keep to the basic intuition of logic program updates, whereby an interpretation is a stable model of the update of a program P by a program U iff it is a stable model of a program consisting of the rules of U together with a subset of the rules of P comprised by those that are not rejected (do not carry over by inertia) due to their being overridden by program U. With the introduction of a DAG to index programs, a program may have more than a single ancestor. This has to be dealt with, the desired intuition being that a program $P_v \in \mathcal{P}_D$ can be used to reject rules of any program $P_u \in \mathcal{P}_D$ if there is a directed path from u to v. Moreover, if some atom is not defined in the update nor in any of its ancestor, its negation is assumed by default. Formally, the stable models of the *MDLP* are:

Definition 3 (Stable Models at state s). *Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a* MDLP, *where* $\mathcal{P}_D = \{P_v : v \in V\}$ *and* $D = (V, E)$. *An interpretation M_s is a* stable model of $\mathcal{P}$ at state $s \in V$, *iff* $M_s = least([\mathcal{P}_s - Reject(s, M_s)] \cup Default(\mathcal{P}_s, M_s))$ *where:*

$$\mathcal{P}_s = \bigcup_{i \leq s} P_i$$
$$Reject(s, M_s) = \{r \in P_i \mid \exists r' \in P_j, i < j \leq s, H(r) = not\, H(r') \wedge M_s \vDash B(r')\}$$
$$Default(\mathcal{P}_s, M_s) = \{not\, A \mid \nexists r \in \mathcal{P}_s : (H(r) = A) \wedge M_s \vDash B(r)\}$$

Intuitively, the set $Reject(s, M_s)$ contains those rules belonging to a program indexed by a state i that are overridden by the head of another rule with true body in state j along a path to state s. $\mathcal{P}_s$ contains all rules of all programs that are indexed by a state along all paths to state s, i.e. all rules that are potentially relevant to determine the semantics at state s. The set $Default(\mathcal{P}_s, M_s)$ contains default negations $not\, A$ of all unsupported atoms A, i.e., those atoms A for which there is no rule in $\mathcal{P}_s$ whose body is true in M_s.

According to [12], to determine the models of a *MDLP* at state s, we need only consider the part of the *MDLP* corresponding to the relevancy graph wrt state s.

We might have a situation where we desire to determine the semantics jointly at more than one state. If all these states belong to the relevancy graph of one of them, we simply determine the models at that state. But this might not be the case. Formally, the semantics of a *MDLP* at an arbitrary set of its states is determined by the definition:

Definition 4 (Stable Models at a set of states S). *Let* $\mathcal{P} = (\mathcal{P}_D, D)$ *be a* MDLP, *where* $\mathcal{P}_D = \{P_v : v \in V\}$ *and* $D = (V, E)$. *Let* S *be a set of states such that* $S \subseteq V$. *An interpretation* M_S *is a* stable model of $\mathcal{P}$ at the set of states S iff $M_S = least\left([\mathcal{P}_S - Reject(S, M_S)] \cup Default\left(\mathcal{P}_S, M_S\right)\right)$ *where:*

$$\mathcal{P}_S = \bigcup_{s \in S}\left(\bigcup_{i \leq s} P_i\right)$$
$$Reject(S, M_S) = \left\{ \begin{array}{c} r \in P_i \mid \exists s \in S, \exists r' \in P_j, i < j \leq s, \\ H(r) = not\, H(r') \wedge M_S \vDash B(r') \end{array} \right\}$$
$$Default\left(\mathcal{P}_S, M_S\right) = \{not\, A \mid \nexists r \in \mathcal{P}_S : (H(r) = A) \wedge M_S \vDash B(r)\}$$

This is equivalent to the addition of a new vertex α to the DAG, and connecting to α, by addition of edges, all states we wish to consider. Furthermore, the program indexed by α is empty. We then determine the stable models of this new *MDLP* at state α. Note the addition of state α does not affect the stable models at other states. Indeed, α and the newly introduced edges do not belong to the relevancy DAG wrt any other state. A particular case of the above definition is when $S = V$, corresponding to the semantics of the whole $MDLP$. In [12], we have presented an alternative definition, based on a purely syntactical transformation that, given a *MDLP*, produces a generalized logic program whose stable models are in a one-to-one equivalence relation with the stable models of the *MDLP* previously characterized. The computation of the stable models at some state s reduces to the computation of the transformation followed by the computation of the stable models of the transformed program. This directly provides for an implementation of $\mathcal{MDLP}$, publicly available at `centria.di.fct.unl.pt/~jja/updates`.

4 Inter- and Intra-Agent Social Viewpoints

The previous Section contains the definition of the notion of *Multi-dimensional Dynamic Logic Programming*, $\mathcal{MDLP}$, as an extension of *DLP* to allow for states to be related by an arbitrary DAG. The stable models of $\mathcal{MDLP}$ have been characterized but nothing has been yet explained as how to use such DAGs to represent real problems. In particular, we have not shown how DAGs allow for the combination of more than one representational dimension, the very motivation to introduce $\mathcal{MDLP}$.

In this section we explore some particular classes of DAGs suitable in the context of multi-agent systems.

Agents are situated and therefore need to represent and reason about information they obtain directly by sensing the environment or communicated by other agents. These agents, as well as the environment, evolve in time, i.e. the incoming information is to be used as an update over existing knowledge. Moreover these agents do not have the same credibility, this being represented via a hierarchy of predominance.
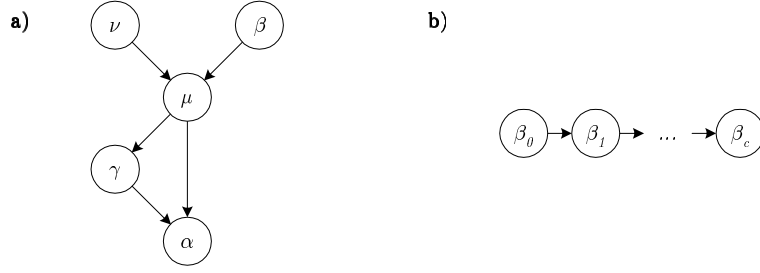
Fig. 1.

In this section we explore DAGs that provide a way to represent the evolution in time of knowledge with provenance in a community of hierarchically related agents.

We start with an agent α, situated in a community of agents represented by the greek letters β, γ, μ, ν. The multi-agent system is $\mathcal{A} = \{\alpha, \beta, \gamma, \mu, \nu\}$. According to agent's α hierarchical view of the world, and its position within the community, all agents are related according to the DAG $D_h = (\mathcal{A}, E_h)$ where $E_h = \{(\nu, \mu), (\beta, \mu), (\mu, \gamma), (\mu, \alpha), (\gamma, \alpha)\}$, depicted in Fig. 1 a).

According to this DAG, agent α's opinions prevail over those of every other agent. However this need not be so. If, for example, one of these agent's role was to coordinate the community, it would be natural to exist an edge connecting α to this agent.

In a static environment, this representation would be suf£cient to determine the semantics of α's view of the community. In such a situation, the rules asserted by each agent would constitute programs indexed by the DAG of Fig. 1 a), i.e. $P_\beta, P_\gamma, P_\mu, ...$

In a realistic scenario, where the dynamics of the system cannot be ignored, there is no single program representing each agent. Rather, there is a sequence of programs representing the knowledge of each agent at each time point. Suppose these time points were represented by the set $T = \{0, 1, ..., c\}$ (where by c we mean the current time state), then, for example, the knowledge of agent β would be represented by the set of programs $\{P_{\beta_0}, P_{\beta_1}, ..., P_{\beta_c}\}$, indexed according to the DAG $D_{\beta_t} = (B_t, E_t)$ where $B_t = \{\beta_t : t \in S\}$ and $E_t = \{(0, 1), ..., (c-1, c)\}$ as depicted in Fig. 1 b).

The full dynamic hierarchical scenario, comprising all agents, is then represented by the set of programs $\mathcal{P}_D = \{P_{a_t} : a \in \mathcal{A}, t \in T\}$ indexed by the DAG $D = (\mathcal{A}_T, E)$ where $\mathcal{A}_T = \{a_t : a \in \mathcal{A}, t \in T\}$.

There still remains to be de£ned the relationships between all these programs, i.e. the edges belonging to E. To this purpose, we will propose three basic ways to systematically relate these programs.

4.1   Equal Role Representation

The £rst approach to combining the hierarchical and temporal dimensions is accomplished by assigning equal roles to both precedence relations. In this scenario, we maintain the temporal precedence relation within each agent, and the hierarchical one within each time state, and we do not relate any two programs that fall outside this scope,
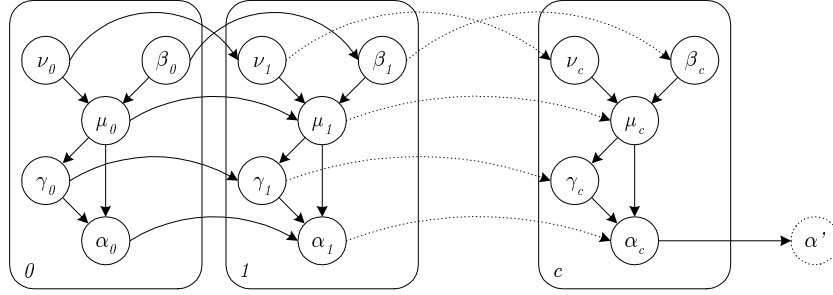
Fig. 2. Equal Role Representation

i.e. there is no precedence between a higher ranked older program and a lower ranked newer one. Accordingly, the set of edges E, of the DAG D contains the union of the following two sets of edges:

Time Dependence Edges (TDE) : $\{(a_{t_1}, a_{t_2}) : a \in \mathcal{A}, t_1, t_2 \in T, t_1 < t_2\}$.
Hierarchy Dependence Edges (HDE) : $\{(a_t, b_t) : a, b \in \mathcal{A}, t \in T, a < b\}$.

Intuitively, each rule can be used to reject any rule of a lower ranked agent indexed by a time state equal or lower than its own. This situation is depicted in Fig. 2.

Remark 1. Throughout this Section, we have chosen a simpli£ed representation of the DAGs to make their interpretation easier. For this purpose, we introduce new nodes (meta-nodes) encapsulating part of the DAG (detail). To obtain the complete DAG from this simpli£cation one needs to replace the meta-node with the detail while replacing the edges entering the meta-node with a set of edges entering each source node of the detail. Similarly, one needs to replace each node departing from the meta-node with a set of edges departing from each sink of the detail. In every DAG, we have added a new node labelled α', which becomes its single sink, and an empty program associated with it, indicating where the semantics corresponding to agent $\alpha's$ view of the overall system at time state c can be determined. Also, since the semantics of MDLP is invariant wrt the transitive closure of the DAG, we will often be omitting some edges that do not affect such transitive closure.

Such a scenario can be found in legal reasoning, where the legislative agency is divided conforming to a hierarchy of power, governed by the principle *Lex Superior (Lex Superior Derogat Legi Inferiori)* according to which the rule issued by a higher hierarchical authority overrides the one issued by a lower one, and the evolution of law in time is governed by the principle *Lex Posterior (Lex Posterior Derogat Legi Priori)* according to which the rule enacted at a later point in time overrides the earlier one. *Lex Superior* is encoded by the Hierarchy Dependence Edges and *Lex Posterior* is encoded the Time Dependence Edges.

Allowing rejection governed by time and hierarchy alone, potentiates contradiction inasmuch as there are many pairs of programs not related according to this graph. If the
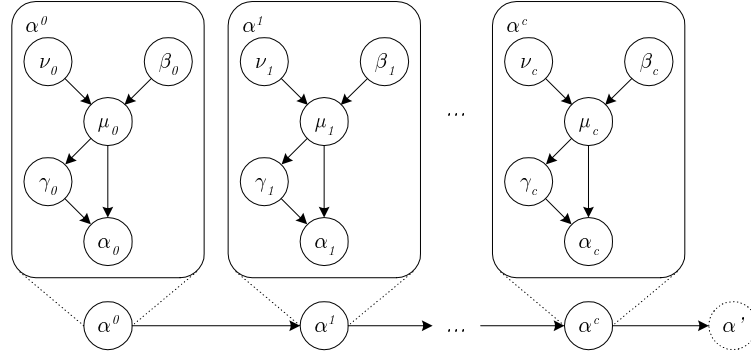
Fig. 3. Time Prevailing Representation

purpose of our agency system were to perform some sort of paraconsistent reasoning, such as in an agent based negotiation system trying to reach an agreement, this would be the ideal scenario: contradiction would generate messages to the responsible agents to possibly review their positions. But often this is not the case and we may want to reduce the amount of contradiction, namely by establishing a skewed relation between the temporal and hierarchical dimensions. Two approaches will be explored in the following subsections.

4.2 Time Prevailing Representation

According to this representation, the DAG D contains, besides the Time and Hierarchy Dependence Edges, the following edges:

Time Prevailing Edges (TPE) : $\{(a_{t_1}, b_{t_2}) : a, b \in \mathcal{A}, t_1, t_2 \in T, t_1 < t_2\}$.

The intuitive reading is that any rule indexed by a more recent time state overrides any older rule, independently of which agents these rules belong to. This situation is depicted in Fig. 3.

This representation is particularly useful in very dynamic situations where competence is distributed, i.e. when knowledge changes rapidly and different agents will typically provide rules about different literals. This is so mainly because any newer rule always overrides any older one. It means that if a situation is completely defined by the rules issued by the community at a given time state, one can simply ignore older rules.

The main drawback of this representation relates to the trustfulness of agents in the community. It requires all agents to be fully trusted because, in allowing all new rules to override all old ones, irrespective of their hierarchical position, any untrustworthy lower ranked agent can override any higher ranked agent just by issuing a rule at a later time state. This leads us to the next, alternative, representation.
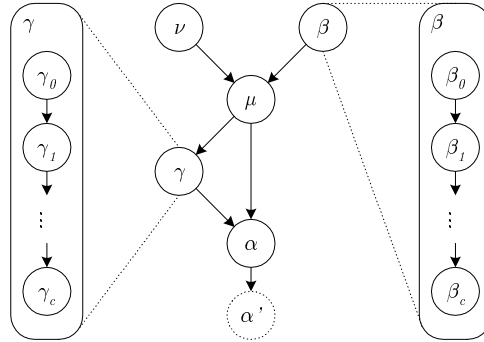
Fig. 4. Hierarchy Prevailing Representation

4.3 Hierarchy Prevailing Representation

According to this representation, the DAG D contains, besides the Time and Hierarchy Dependence Edges, the following edges:

Hierarchy Prevailing Edges (HPE) $: \{(a_{t_1}, b_{t_2}) : a, b \in \mathcal{A}, t_1, t_2 \in T, a < b\}.$

The intuitive reading is that any rule indexed by a higher ranked agent overrides any lower ranked agent's rule, independently of the time state it is indexed by. This situation is depicted in Fig. 4.

This situation is useful, in contrast with the previous one, when some of the agents are untrustworthy because a lower ranked agent rule, to be used, may not be contradicted by any (even if older) higher ranked agent rule. The main drawback is that one has to consider the entire history of all higher ranked agents in order to accept/reject a rule provided by a lower ranked agent. Elsewhere [1], a number of techniques to reduce the size of a dynamic logic program have been developed, useful for simplifying the time sequence of programs of each individual agent.

Again in the context of Legal Reasoning, this scenario corresponds to the one used in many Legislatures, where collisions between rules are governed by the principle *Lex Superior Priori Derogat Legi Inferiori Posterior*, i.e. the rule issued by a higher hierarchical authority at an earlier point overrides the one issued by a lower hierarchical authority at a later point.

4.4 Representing Inter- and Intra-Agent Relationships

The representations set forth in the previous sub-sections refer to a community of agents. Nevertheless, they can be used at different levels of abstraction to represent macro and micro aspects of a multi-agent system, in a unified manner. Let us suppose that agent α is composed of several sub-agents concurrently performing dedicated tasks while reading and writing onto a common knowledge structure. In the next Section we
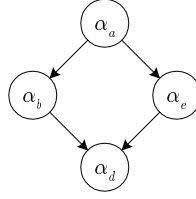
Fig. 5. Sub-agent Hierarchy

describe the overall architecture based on such notions. According to this view, agent α can now be seen as a community of sub-agents $\mathcal{A}_\alpha = \{\alpha_a, \alpha_b, \alpha_d, \alpha_e\}$, related, for example, according to the DAG $D_\alpha = (\mathcal{A}_\alpha, E_\alpha)$ where $E_\alpha = \{(\alpha_a, \alpha_b), (\alpha_a, \alpha_e),$ $(\alpha_b, \alpha_d), (\alpha_e, \alpha_d)\}$ as in Fig. 5. The overall dynamic system, comprising all agents and sub-agents, is now represented by the set of programs $\mathcal{P}_D = \{P_{a_t} : a_t \in \mathcal{A}_T\}$ indexed by the DAG $D = (\mathcal{A}_T, E)$ where $\mathcal{A}_T = \{a_t : a \in \mathcal{A} \setminus \{\alpha\}, t \in T\} \cup \{a_t : a \in \mathcal{A}_\alpha, t \in T\}$.

As for the relations between the programs, we propose a combination of the time and hierarchy prevailing representations to relate the sub-agents and agents respectively. As mentioned before, the time prevailing representation is the most ef£cient but requires all agents to be trusted. One would expect an agent to trust its component sub-agents. As for the representation of other agents, we will opt for the hierarchy prevailing relation. Formally, the set of edges in the DAG contains:

Time Prevailing Edges (TPE) : $\{(a_{t_1}, b_{t_2}) : a, b \in \mathcal{A}_\alpha, t_1, t_2 \in T, t_1 < t_2\}$, to model the relationships between the sub-agents of α.

Hierarchy Prevailing Edges (HPE) : $\{(a_{t_1}, b_{t_2}) : a, b \in \mathcal{A}, t_1, t_2 \in T, a < b\}$, to model the relationships between the agents of the system. Note that each edge entering (resp. departing from) α_t should be interpreted as a set of edges entering (resp. departing from) each of $\{\alpha_{a_t}, \alpha_{b_t}, \alpha_{d_t}, \alpha_{e_t}\}$.

This situation is depicted in Fig. 6. Note however that this is just one proposal of the many possible existing combinations to represent such relations.

5 Conclusions

In this paper we have explored Multi-dimensional Dynamic Logic Programming as a means to combine knowledge provenient from different agents, into a single knowledge base point of view, with a precise a declarative semantics. Depending on the situation and the relations amongst the agents, we have envisaged several classes of acyclic digraphs suitable for its encoding.

Based on the strengths of $\mathcal{MDLP}$ as a framework capable of simultaneously represent several aspects of a system in a dynamic fashion, and of *LUPS* as a powerful language to specify the evolution of such representations by means of transitions, we have launched into the design of an agent architecture, $\mathcal{MINERVA}$. It aims at providing, on a sound theoretical basis, a common agent framework based on the strengths
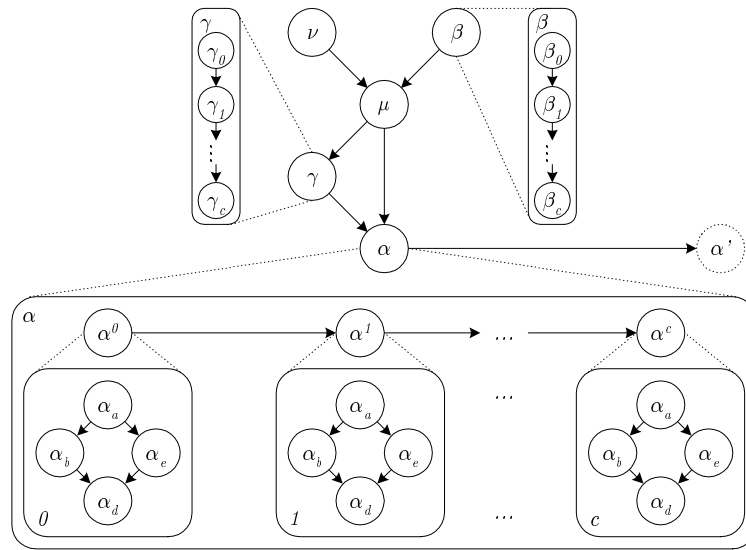
Fig. 6. Inter- and Intra-Agent Relationship Representation

of Logic Programming, to allow the combination of several non-monotonic knowledge representation and reasoning mechanisms developed in recent years.

The use of Logic Programming for the overall endeavour is justified on the ground of it providing a rigorous single encompassing theoretical basis for the aforesaid topics, as well as an implementation vehicle for parallel and distributed processing. Additionally, logic programming provides a formal high level flexible instrument for the rigorous specification and experimentation with computational designs, making it extremely useful for prototyping, even when other, possibly lower level, target implementation languages are envisaged.

Rational agents, in our opinion, will require an admixture of any number of those reasoning mechanisms mentioned in the introduction, to carry out their tasks. To this end, a $\mathcal{MINERVA}$ agent is based on a modular design where a common knowledge base is concurrently manipulated by specialized sub-agents. The common knowledge base contains all knowledge shared by more that one sub-agent. It is conceptually divided in 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 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 functionality related sub-agents, that execute 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 their behaviour, and interfacing with the *Common Knowledge Base*. Whilst some of those sub-agent's functionalities are fully speci£able in *LUPS*, others require private specialized procedures where *LUPS* serves as an interface language. In [11], this paper's sequel, we describe the $\mathcal{MINERVA}$ agent architecture

References

1. J. J. Alferes, J. A. Leite, and L. M. Pereira. Garbage collection in dynamic logic programming, 2000. Unpublished notes.
2. 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. Short version titled *Dynamic Logic Programming* appeared in Procs. of KR-98.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, and P. Quaresma. Planning as abductive updating. In *Procs. of AISB'00*. AISB, 2000.
4. J. J. Alferes and L. M. Pereira. Updates plus preferences. In *Procs. of (JELIA-00)*, volume 1919 of *LNAI*. Springer, 2000.
5. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Arti£cial Intelligence*. To appear. Short version appeared in Procs of LPNMR-99, LNAI-1730.
6. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic programming and multi-agent system: A synergic combination for applications and semantics. In *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999.
7. P. Dell'Acqua and L. M. Pereira. Updating agents. In *Procs of MAS-99, ICLP-99 Ws.*, 1999.
8. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Procs. of ICLP-88*. MIT Press, 1988.
9. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
10. E. Lamma, F. Riguzzi, and L. M. Pereira. Strategies in combined learning via logic programs. *Machine Learning*, 38(1/2):63–87, 2000.
11. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. To appear in Procs. of ATAL-01.
12. 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.
13. J. A. Leite, F. C. Pereira, A. Cardoso, and L. M. Pereira. Metaphorical mapping consistency via dynamic logic programming. In *Procs. of AISB'00*. AISB, 2000.
14. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *Procs. of KR-92*. Morgan-Kaufmann, 1992.
15. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *Procs. of (LPNMR-97)*, volume 1265 of *LNAI*. Springer, 1997.
16. P. Quaresma and I. P. Rodrigues. A collaborative legal information retrieval system using dynamic logic programming. In *Procs. of ICAIL-99*. ACM Press, 1999.
17. F. Sadri and F. Toni. Computational logic and multiagent systems: A roadmap, 1999. Available from `http://www.compulog.org`.
18. XSB-Prolog. The XSB logic programming system, version 2.0, 1999. Available at `http://www.cs.sunysb.edu/ sbprolog`.