Dynamic Logic Programming with Multiple Dimensions

João Alexandre Leite, José Júlio Alferes and Luís Moniz Pereira

Abstract

According to the recently introduced notion of *Dynamic Logic Programming*, knowledge is given by a set of theories (encoded as logic programs) representing different states of the world. Different states may represent different time periods, different hierarchical instances, or even different domains. The mutual relationships between different states are used to determine the semantics of the combined theory composed of all individual theories. Although suitable to encode one of the possible representational dimensions (e.g. time, hierarchies, domains,...), *Dynamic Logic Programming* cannot deal with more than one such dimensions simultaneously because it is only defined for linear sequences of states. In this paper, we extend *Dynamic Logic Programming* to allow for collections of states represented by arbitrary acyclic digraphs, introducing the notion of *Multi-dimensional Dynamic Logic Programming*. Within this more general theory, we will also show how some particular state structures can be quite useful to model and reason about dynamic multi-agent systems.

1 Introduction and Motivation

During the last few years, the notion of agency has virtually invaded every sub-field of computer science. Although commonly implemented by means of imperative languages, mainly for reasons of efficiency, the agent paradigm has recently increased its influence in the research and development of computational logic based systems. Since efficiency is not always a real issue, but clear specification and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought (back) to the spot-light. To this accrues the recent significant improvements in the efficiency of *Logic Programming* implementations for *Non-monotonic Reasoning* [8, 19, 23]. Besides allowing for a unified declarative and procedural semantics, eliminating the traditional high gap between theory and practice, the use of several and quite powerful results in the field of non-monotonic extensions to *Logic Programming (LP)* can represent an important added value to the design of rational agents. For a better

All Authors are with Centro de Inteligência Artificial (CENTRIA), Departamento de Informática, Universidade Nova de Lisboa, 2825-114 Caparica, Portugal. E-mail: `jleite|jja|lmp@di.fct.unl.pt`

All Authors are partially supported by project MENTAL (PRAXIS XXI 2/2.1/TIT/1593/95).
First Author is partially supported by PRAXIS XXI scholarship no. BD/13514/97.

understanding and a thorough exposition of how *Logic Programming* can contribute to agent-based computing, the reader is referred to [22] and [6]. Embedding agent rationality in the LP paradigm affords us with a number of tools and formalisms captured in that paradigm, such as belief revision, inductive learning, argumentation, preferences, etc. [13, 18]

In [1], the paradigm of *Dynamic Logic Programming (DLP)* was introduced, following the eschewing of performing updates on a model basis, but rather as a process of logic programming rule updates [15]. According to *DLP*, knowledge is given by a set of theories (encoded as generalized logic programs) representing different states of the world. Different states may represent different time periods, different hierarchical instances, or even different domains. Consequently, the individual theories may contain mutually contradictory as well as overlapping information. The role of *Dynamic Logic Programming* is to use the mutual relationships extant between different states to precisely determine the declarative as well as the procedural semantics of the combined theory composed of all individual theories.

Since its introduction, *DLP* has been employed to represent several different aspects of a system, namely as a way to:

- represent and reason about the evolution of knowledge in time [1, 2];
- combine rules learned by agents [17];
- reason about updates of agents' beliefs [9];
- model agent interaction [20, 21];
- model and reason about actions [3];
- solve inconsistencies in metaphorical reasoning [16];

The common aspect among these *DLP* applications is that the states associated with the given set of theories only encode one of the several possible representational dimensions (e.g. time, hierarchies, domains,...). This is so because *DLP* is only defined for linear sequences of states.

For example, *DLP* can be used to model the relationship of a hierarchical related 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 a group of agents over time. An instance of such a multi-dimensional scenario can be found in legal reasoning, where the legislative agency is divided according 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. *DLP* can be used to model each of these principles individually, by using the sequence of states to represent either the hierarchy or time, but is unable to cope with both at once when they interact. For this to be possible, *DLP* needs to be extended to allow a more general structure of states. An example based on these legal principles will be further explored in a subsequent section.

In this paper we introduce the notion of *Multi-dimensional Dynamic Logic Programming (MDLP)* which generalizes *DLP* to allow for collections of states represented by arbitrary acyclic digraphs. Within this more general theory, we will also show how some particular state structures can be quite useful to model and reason about dynamic multi-agent systems.

The remainder of the paper is structured as follows: In Sect.2 we briefly recap *DLP* and some other definitions used throughout the paper. In Sect.3 we define *MDLP*. In Sect.4 we relate *MDLP* to multi-agent systems and illustrate with an example. In Sect.5 we motivate further developments and conclude.

2 Background Definitions

In this section we begin with a brief review of the so called generalized logic programs, their stable model semantics, and the notion of dynamic logic programming, as set forth in [1]. For the purpose of self containment, we will also recap some definitions related to graph theory.

2.1 Generalized Logic Programs and their Stable Models

To represent negative information in logic programs and their updates, we need more general logic programs that allow default negation $not\, A$ not only in rule premises but also in their heads. It is worth noting why, in the update setting, generalizing the language to allow default negation in rule heads (thus defining "*generalized logic programs*") is more adequate than introducing explicit negation in programs (both in heads and bodies). Suppose we are given a rule stating that A is true whenever some condition $Cond$ is met. This is naturally represented by the rule $A \leftarrow Cond$. Now suppose we want to say, as an update, that A should no longer be the case (i.e. should be deleted or retracted), if some condition $Cond'$ is met. How to represent this new knowledge? By using extended logic programming (with explicit negation) this could be represented by $\neg A \leftarrow Cond'$. But this rule says more than we want to. It states that A is false upon $Cond'$, and we only want to go as far as to say that the truth of A is to be deleted in that case. All we want is to say that, if $Cond'$ is true, then $not\, A$ should be the case, i.e. $not\, A \leftarrow Cond'$. As argued in [11], the difference between explicit and default negation is fundamental whenever the information about some atom A cannot be assumed to be complete. Under these circumstances, the former means that there is evidence for A being false, while the latter means that there is no evidence for A being true. In the deletion example, we desire the latter case. Note, however, that the adequacy of generalized logic programs for this purpose is in facilitating the intuitive writing of updates. Indeed, as proven in [7], generalized logic programs and extended logic programs have the same expressive power. In fact, every generalized program can be transformed into an extended program, in such a way that the stable models of the original correspond to the answer sets of the transformed one, and vice-versa. We refer to [7] for these transformations' rendering.

For the definition of the semantics it will be convenient to *syntactically* represent generalized logic programs as *propositional Horn theories*. In particular, we will represent default negation $not\, A$ as a standard propositional variable (atom). Suppose that $\mathcal{K}$ is an arbitrary set of propositional variables whose names do not begin with a "*not*". By the propositional language $\mathcal{L}_{\mathcal{K}}$ *generated* by the set $\mathcal{K}$ we mean the language $\mathcal{L}$ whose set of propositional variables consists of $\{A : A \in \mathcal{K}\} \cup \{not\, A : A \in \mathcal{K}\}$. Atoms $A \in \mathcal{K}$, are called *objective atoms* while the atoms $not\, A$ are called *default atoms*. From the definition it follows that the two sets are disjoint. By a *generalized logic program* P in the language $\mathcal{L}_{\mathcal{K}}$ we mean a finite or infinite set of propositional Horn clauses r of the form $L \leftarrow L_1, \ldots, L_n$where L and L_i are atoms from $\mathcal{L}_{\mathcal{K}}$. By $head(r)$ we mean L. If $head(r) = A$ (resp. $head(r) = not\, A$) then $not\, head(r) = not\, A$ (resp. $not\, head(r) = A$). If all the atoms L appearing in heads of clauses of P are objective atoms, then we say that the logic program P is *normal*. Consequently, from a syntactic standpoint, a logic program is simply viewed as a propositional Horn theory. However, its *semantics* significantly differs from the semantics of classical propositional theories and is determined by the class of stable models defined below.

By a (2-valued) *interpretation* M of $\mathcal{L}_{\mathcal{K}}$ we mean any set of atoms from $\mathcal{L}_{\mathcal{K}}$ that satisfies the condition that for any A in $\mathcal{K}$, precisely one of the atoms A or $not\, A$ belongs to M. Given an interpretation M we define:

$$
\begin{aligned}
&M^{+} = \{A \in \mathcal{K} : A \in M\} \\
&M^{-} = \{not\, A : not\, A \in M\} \;= \{\; not\, A : A \notin M\}.
\end{aligned}
$$

Definition 1 ***(Stable models of generalized logic programs)*** *We say that a (2-valued) interpretation M of $\mathcal{L}_{\mathcal{K}}$ is a stable model of a generalized logic program P if M is the least model of the Horn theory $P \cup M^{-}$: $M =$Least$(P \cup M^{-})$.* $\diamond$

Following an established tradition, from now on we will often be omitting the default (negative) atoms when describing interpretations and models.

2.2 Dynamic Logic Programming

Here, we recap the notion of *dynamic program update* $\bigoplus\{\; P_s : s \in S\}$ over a totally ordered set $\mathcal{P} = \{\; P_s : s \in S\}$ of logic programs. The idea of dynamic updates, inspired by [14], is simple and quite fundamental. Suppose that we are given a set of program modules P_s, indexed by different states of the world s. 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 $\bigoplus\; \{P_s : s \in S\}$ is to use the mutual relationships existing between different states (and specified in the form of the ordering relation) to precisely determine, at any given state s, the *declarative* as well as the *procedural* semantics of the combined program, composed of all modules. Consequently, the notion of a dynamic program update supports the important paradigm of *dynamic logic programming*. Given individual and largely *independent* program modules P_s describing our knowledge at different states of the world (for example, the knowledge acquired at different times), the dynamic program update $\bigoplus\{P_s \;:\; s \;\in S\;\}$ specifies the exact meaning of the union of these programs.

Suppose that $\mathcal{P} = \{P_s : s \in S\}$ is a finite or infinite sequence of generalized logic programs in the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$, indexed by the finite or infinite set $S =$

$\{1, 2, \ldots, n, \ldots\}$ of natural numbers. We will call elements s of the set $S \cup \{0\}$ *states* and we will refer to 0 as the *initial state.*

By $\overline{\mathcal{K}}$ we denote the following superset of the set $\mathcal{K}$ of propositional variables:

$$\overline{\mathcal{K}} = \mathcal{K} \cup \left\{ A^-, A_s,\ A_s^-, A_{P_s},\ A_{P_s}^- : A \in \mathcal{K},\ s \in S \cup \{0\} \right\}$$

We denote by $\overline{\mathcal{L}} = \mathcal{L}_{\overline{\mathcal{K}}}$ the extension of the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$ generated by $\overline{\mathcal{K}}$.

Definition 2 ***(Dynamic Program Update)*** *By the dynamic program update over the sequence of updating programs $\mathcal{P} = \{P_s : s \in S\}$ we mean the logic program $\biguplus \mathcal{P}$, which consists of the following clauses in the extended language $\overline{\mathcal{L}}$:*

- $A_{P_s} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$ *for any clause* $A \leftarrow B_1, \ldots, B_m, \ not\, C_1, \ldots, \ not\, C_n$ *in the program P_s, where $s \in S$.*
- $A_{P_s}^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$ *for any clause* $not\, A \leftarrow B_1, \ldots, B_m, \ not\, C_1, \ldots, \ not\, C_n$ *in the program P_s, where $s \in S$.*
- $A_s \leftarrow A_{s-1}, not\, A_{P_s}^-$ *and* $A_s^- \leftarrow A_{s-1}^-, not A_{P_s}$ *and* $A_s \leftarrow A_{P_s}$ *and* $A_s^- \leftarrow A_{P_s}^-$ *for all objective atoms $A \in \mathcal{K}$ and for all $s \in S$.*
- A_0^-*,for all objective atoms $A \in \mathcal{K}$.* ◇

Observe that the dynamic program update $\biguplus \mathcal{P}$ is a normal logic program, i.e., it does not contain default negation in heads of its clauses. Moreover, only the inheritance rules contain default negation in their bodies. Also note that the program $\biguplus \mathcal{P}$ does not contain the atoms A or A^-, where $A \in \mathcal{K}$, in heads of its clauses. These atoms appear only in the bodies of rewritten program clauses. The notion of the dynamic program update $\bigoplus_s \mathcal{P}$ at a given state $s \in S$ changes that:

Definition 3 ***(Dynamic Program Update at a Given State)*** *Given a fixed state $s \in S$, by the dynamic program update at the state s, denoted by $\bigoplus_s \mathcal{P}$, we mean the dynamic program update $\biguplus \mathcal{P}$ augmented with the following* Current State Rules*:*

- $A \leftarrow A_s$ *and* $A^- \leftarrow A_s^-$ *and* $not\, A \leftarrow A_s^-$ *for all objective atoms $A \in \mathcal{K}$.* ◇

2.3 Graphs

A *directed graph*, or *digraph*, $D = (V, E, \delta)$ is a composite notion of two sets $V = V_D$ of *vertices* and $E = E_D$ of (*directed*) *edges* and a mapping $\delta : E \to V \times V$. If $\delta(e) = (v, w)$ then v is called the *initial vertex* and w the *final vertex* of the edge e. A *directed edge sequence from v_0 to v_n* in a digraph is a sequence of edges $e_1, e_2, ..., e_n$ such that $\delta(e_i) = (v_{i-1}, v_i)$ 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_i to v_i, for all vertices v_i 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 $v_i < v_j$ if there is a directed path from v_i to v_j and that $v_i \leq v_j$ if $v_i < v_j$ or $i = j$.

For simplicity, we will omit the explicit representation of the mapping δ of a graph, and represent its edges $e \in E$ by their corresponding pairs of vertices (v, w)

such that $(v, w) = \delta(e)$. Therefore, a graph D will be represented by the pair (V, E) where V is a set of vertices and E is a set of pairs of vertices.

Follows the notion of relevancy DAG, D_v, of a DAG D, wrt a vertex v of D.

Definition 4 ***(Relevancy DAG)*** *Let $D = (V, E)$ be an acyclic digraph. Let v be a vertex of D, i.e. $v \in V$. The relevancy DAG of D wrt v is $D_v = (V_v, E_v)$ where:*

$$V_v = \{v_i : v_i \in V \text{ and } v_i \leq v\}$$
$$E_v = \{(v_i, v_j) : (v_i, v_j) \in E \quad \text{and } v_i, v_j \in V_v \ \} \diamond$$

Intuitively the relevancy DAG of D wrt v is the subgraph of D consisting of all vertices and edges contained in all directed paths to v.

3 Multi-dimensional Dynamic Logic Programming

As noted in the introduction, allowing the individual theories of a dynamic program update to be related by a linear sequence of states only, limits the use of DLP to represent and reason about a single dimension, where by dimension we mean an aspect that is encoded by the states (e.g. time, hierarchy,...). In this section we generalize DLP to allow for states to be represented by the vertices of an acyclic digraph (DAG) and their relations by the corresponding edges, thus enabling to concurrently represent, depending on the choice of a particular DAG, several dimensions of a representational updatable system. In particular, the DAG can represent not only a system with n independent dimensions, but also the inclusion of inter-dimension dependencies. We start by defining the framework consisting of the generalized logic programs indexed by a DAG:

Definition 5 ***(Multi-dimensional Dynamic Logic Program)*** *Let $\mathcal{L}_\mathcal{K}$ be a propositional language as described before. A* Multi-dimensional Dynamic Logic Program, *$\mathcal{P}$, is a pair $(\mathcal{P}_D, D)$ where $D = (V, E)$ is an acyclic digraph and $\mathcal{P}_D = \{P_v : v \in V\}$ is a set of generalized logic programs in the language $\mathcal{L}_\mathcal{K}$, indexed by the vertices $v \in V$ of D. We call such vertices of D states. For simplicity, we will often leave the language $\mathcal{L}_\mathcal{K}$ implicit.* $\diamond$

We want to characterize the models of $\mathcal{P}$ at any given state. For this purpose, we will maintain 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 overriding by program U. With the introduction of a *DAG* to index the programs, it is no longer the case that a given program has a single ancestor. This has to be dealt with, the 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. In the example depicted in Fig.1, rules of P_i can be used to reject rules from P_c but not to reject rules from P_f.

Formally, the models of the Multi-dimensional Dynamic Logic Program are characterized according to the definition:

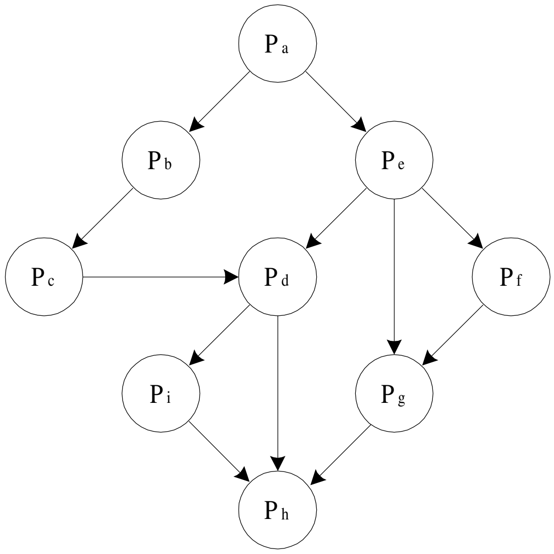

Figure 1:

Definition 6 ***(Stable Models at state*** s***)*** *Let* $\mathcal{P} = (\mathcal{P}_D, D)$ *be a Multi-dimensional Dynamic Logic Program, where* $\mathcal{P}_D = \{P_v : v \in V\}$ *and* $D = (V, E)$. *An interpretation* M_s *is a* stable model of the multi-dimensional update at state $s \in V$, *iff:*

$$M_s = least\left([\mathcal{P}_s - Reject(s, M_s)] \cup Default\left(\mathcal{P}_s, M_s\right)\right)$$

where

$$\mathcal{P}_s = \bigcup_{i \leq s} P_i$$

$$Reject(s, M_s) = \left\{ \begin{array}{c} r \in P_i \mid \exists r' \in P_j, i < j \leq s, \\ head(r) = not\, head(r') \wedge M_s \vDash body(r') \end{array} \right\}$$

$$Default\left(\mathcal{P}_s, M_s\right) = \{not\, A \mid \nexists r \in \mathcal{P}_s : (head(r) = A) \wedge M_s \vDash body(r)\} \diamond$$

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 $Default\left(\mathcal{P}_s, M_s\right)$ 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.

Example 7 *Consider the following programs:* $P_t = \{a \leftarrow not\, b\}$, $P_u = \{c \leftarrow\}$, $P_v = \{not\, a \leftarrow c\}$, $P_w = \{\}$ *indexed by the graph* $D = (V, E)$ *such that* $V = \{t, u, v, w\}$ *and* $E = \{(t, u), (t, v), (u, w), (v, w)\}$, *as depicted in Fig.2. The only stable model at state* w *is* $M_w = \{not\, a, not\, b, c\}$. *To confirm, we have that:*

$$Reject(w, M_w) = \{a \leftarrow not\, b\} \qquad Default\left(\mathcal{P}_w, M_w\right) = \{not\, b\}$$

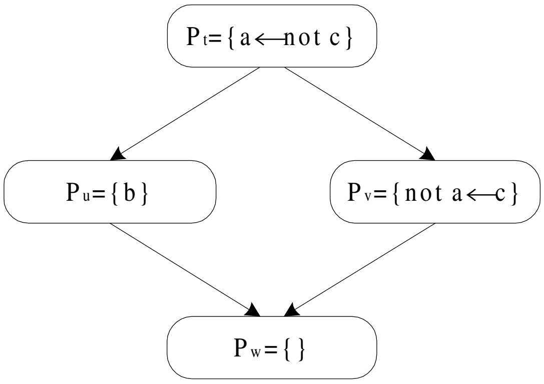

Figure 2:

and, finally,

$$[\mathcal{P}_w - Reject(s, M_w)] \cup Default\,(\mathcal{P}_w, M_w) = \{not\, a \leftarrow c; c \leftarrow; not\, b\}$$

whose least model is M_w. Note that at state v the only stable model is $M_v = \{a, not\, b, not\, c\}$ because the rule $not\, a \leftarrow c$ only rejects the rule $a \leftarrow not\, b$ at state w, that is, when both the rules $not\, a \leftarrow c$ and $c \leftarrow$ are present. $\diamond$

The following theorem establishes that *Multi-dimensional Dynamic Logic Programming* is a generalization of *Dynamic Logic Programming.*

Theorem 8 ***(Generalization of DLP)*** *Let $\mathcal{P}_D = \{P_s : s \in S\}$ be a sequence of generalized logic programs in the language $\mathcal{L} = \mathcal{L}_\mathcal{K}$, indexed by set of natural numbers $S = \{1, 2, 3, \ \ldots\ , n\}$. Let $\mathcal{P} = (\mathcal{P}_D, D)$ be the* Multi-dimensional Dynamic Logic Program, *where $D = (S, E)$ is the acyclic digraph such that $E = \{(1,2), (2,3), ..., (n-1, n)\}$. Then, an interpretation N of the language $\overline{\mathcal{L}} = \mathcal{L}_{\overline{\mathcal{K}}}$ is a stable model of the dynamic program update at state s, $\bigoplus_s \mathcal{P}_D$, if and only if N is the extension $N = \overline{M}$ of an interpretation M such that M is a stable model of the multi-dimensional update at state s.* $\diamond$

3.1 Transformational Semantics

The previous definition (Def.6) establishes the semantics of *Multi-dimensional Dynamic Logic Programming* by characterizing its stable models at each state. Next we present an alternative definition, based on a purely syntactical transformation that, given a *Multi-dimensional Dynamic Logic Program,* $\mathcal{P} = (\mathcal{P}_D, D)$, produces a generalized logic program whose stable models are in a one-to-one relation with the stable models of the multi-dimensional update previously characterized. This transformation also provides an approach towards a mechanism for implementing *Multi-dimensional Dynamic Logic Programming*: with a pre-processor performing the transformation query answering is reduced to that over generalized logic programs.

Similar to DLP, and without loss of generality, we will extend the DAG D with an initial state (s_0) and a set of directed edges (s_0, s') connecting the initial state to all the sources of D. For our purposes, we will extend $\overline{\mathcal{K}}$ with a predicate $reject/1$. Therefore, from now on $\overline{\mathcal{K}}$ denotes the superset of the set $\mathcal{K}$ of propositional variables:

$$\overline{\mathcal{K}} = \mathcal{K} \cup \left\{A^-, A_s,\ A_s^-, A_{P_s},\ A_{P_s}^-, reject(A_s), reject(A_s^-) : A \in \mathcal{K},\ s \in V \cup \{s_0\}\right\}$$

Definition 9 ***(Multi-dimensional Dynamic Program Update)*** *Let $\mathcal{P}$ be a Multi-dimensional Dynamic Logic Program, where $\mathcal{P} = (\mathcal{P}_D, D)$, $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Given a fixed state $s \in V$, the multi-dimensional dynamic program update over $\mathcal{P}$ at state* **s** *is the generalized logic program $\boxplus_s \mathcal{P}$, which consists of the following clauses in the extended language $\overline{\mathcal{L}}$, where $D_s = (V_s, E_s)$ is relevancy DAG of D wrt s:*

(RP) Rewritten program clauses:

$$A_{P_v} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$
$$A_{P_v}^- \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

for any clause:

$$A \leftarrow B_1,\ \ldots,\ B_m,\ \ not\, C_1,\ \ldots,\ not\, C_n$$

respectively, for any clause:

$$not\, A \leftarrow B_1,\ \ldots,\ B_m,\ not\, C_1,\ \ldots,\ not\, C_n$$

in the program P_v, where $v \in V_s$. The rewritten clauses are obtained from the original ones by replacing atoms A (respectively, the atoms $not\, A$) occurring in their heads by the atoms A_{P_v} (respectively, $A_{P_v}^-$) and by replacing negative premises $not\, C$ by C^-.

(IR) Inheritance rules:

$$A_v \leftarrow A_u, not\, reject(A_u) \qquad A_v^- \leftarrow A_u^-, not\, reject(A_u^-)$$

for all objective atoms $A \in \mathcal{K}$ and all $(u, v) \in E_s$. The inheritance rules say that an atom A is true (respectively, false) in the state $v \in V_s$ if it is true (respectively, false) in any ancestor state u and it is not *rejected,* i.e., forced to be false (respectively, true).

(RR) Rejection Rules:

$$reject(A_u^-) \leftarrow A_{P_v} \qquad reject(A_u) \leftarrow A_{P_v}^-$$

for all objective atoms $A \in \mathcal{K}$ and all $u, v \in V_s$ where $u < v$. The rejection rules say that if an atom A is true (respectively, false) in the program P_v, then it rejects inheritance of any false (respectively, true) atoms of any ancestor.

(UR) Update rules:

$$A_v \leftarrow A_{P_v} \qquad A_v^- \leftarrow A_{P_v}^-$$

for all objective atoms $A \in \mathcal{K}$ and all $v \in V_s$. The update rules state that an atom A must be true (respectively, false) in the state $v \in V_s$ if it is true (respectively, false) in the program P_v.

(DR) Default Rules:

$$A^-_{s_0}$$

for all objective atoms $A \in \mathcal{K}$. Default rules describe the initial state s_0 by making all objective atoms initially false.

($\mathbf{CS}_s$) Current State Rules:

$$A \leftarrow A_s \qquad A^- \leftarrow A^-_s \qquad not\, A \leftarrow A^-_s$$

for all objective atoms $A \in \mathcal{K}$. Current state rules specify the current state s in which the program is being evaluated and determine the values of the atoms A, A^- and $not\, A$. $\diamond$

As mentioned before, the generalized stable models of the program obtained by the previous transformation coincide with those characterized in Def.6, as expressed in the following theorem:

Theorem 10 *Given a Multi-dimensional Dynamic Logic Program $\mathcal{P} = (\mathcal{P}_D, D)$, the generalized stable models of $\boxplus_s\mathcal{P}$, restricted to $\mathcal{L}$, coincide with the generalized stable models of the multi-dimensional update at state s according to Def.6.* $\diamond$

The transformation specified in Def.9 depends on the prior determination of the relevancy graph wrt the given state. Our choice to make this so was based on criteria of clarity and readability. Nevertheless this needs not be so: one can also declaratively specify, by means of a logic program, the notion of relevancy graph, giving rise to the following transformation as the basis of an implementation:

Definition 11 ***(Multi-dimensional Dynamic Program Update - Alternative Transformation)*** *Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a Multi-dimensional Dynamic Logic Program, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Given a fixed state $s \in V$, the multi-dimensional dynamic program update over $\mathcal{P}$ at state* **s** *is the generalized logic program $\boxplus_s\mathcal{P}$, which consists of the following clauses in the extended language*

$$\overline{\mathcal{L}} \cup \{rel\_edge(u,v), rel\_path(u,v), edge(u,v) : u, v \in V\}$$

(RP) Rewritten program clauses:

$$A_{P_v} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$
$$A^-_{P_v} \leftarrow B_1, \ldots, B_m, C_1^-, \ldots, C_n^-$$

for any clause:

$$A \;\leftarrow B_1,\; \ldots,\; B_m,\;\; not\, C_1,\; \ldots,\; not\, C_n$$

respectively, for any clause:

$$not\, A \;\;\leftarrow B_1,\; \ldots,\; B_m,\; not\, C_1,\; \ldots,\; not\, C_n$$

in the program P_v, where $v \in V$.

(IR) Inheritance rules:

$$A_v \leftarrow A_u, not\, reject(A_u), rel\_edge(u,v)$$
$$A_v^- \leftarrow A_u^-, not\, reject(A_u^-), rel\_edge(u,v)$$

for all objective atoms $A \in \mathcal{K}$ and all $u, v \in V$.

(RR) Rejection Rules:

$$reject(A_u^-) \leftarrow A_{P_v}, rel\_path(u,v) \qquad reject(A_u) \leftarrow A_{P_v}^-, rel\_path(u,v)$$

for all objective atoms $A \in \mathcal{K}$ and all $u, v \in V$.

(UR) Update rules:

$$A_v \leftarrow A_{P_v} \qquad A_v^- \leftarrow A_{P_v}^-$$

for all objective atoms $A \in \mathcal{K}$ and all $v \in V$.

(DR) Default Rules:

$$A_{s_0}^-$$

for all objective atoms $A \in \mathcal{K}$.

($\mathbf{CS}_s$) Current State Rules:

$$A \leftarrow A_s \qquad A^- \leftarrow A_s^- \qquad not\, A \leftarrow A_s^-$$

for all objective atoms $A \in \mathcal{K}$.

(ER) Edge Rules

$$edge(u,v)$$

for all $(u,v) \in E$. Edge rules describe the edges in graph D.

($\mathbf{RER}_s$) Current State Relevancy Edge Rules

$$rel\_edge(X,s) \leftarrow edge(X,s)$$
$$rel\_edge(X,Y) \leftarrow edge(X,Y), rel\_path(Y,s)$$

Current State Relevancy Edge Rules define which edges are in the relevancy graph wrt s.

(RPR) Relevancy Path Rules

$$rel\_path(X,Y) \leftarrow rel\_edge(X,Y)$$
$$rel\_path(X,Z) \leftarrow rel\_edge(X,Y), rel\_path(Y,Z)$$

Relevancy Path rules define the notion of relevancy path in a graph. $\diamond$

Based on this transformation, an implementation running on XSB-Prolog exists and can be found at `http://centria.di.fct.unl.pt/~jja/updates/`.

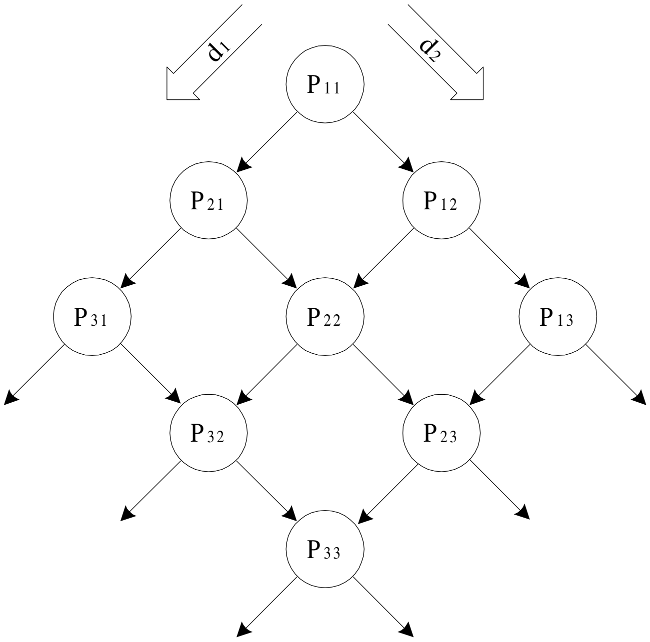

Figure 3:

4 MDLP and Multi-agent Systems

Some particular acyclic digraphs are especially useful to model several aspects of multi-agent systems. One of these is a n-dimensional lattice, where each dimension represents one particular characteristic to be modeled. Suppose a linear hierarchically related society of agents situated in a dynamic environment. Fig.3 represents the lattice that encodes this situation. It has two dimensions, one representing the linearly arranged agents, and the other representing time. If d_1 represents time and d_2 represents the hierarchy, P_{11} contains the new knowledge of agent 1 at time 1. P_{32} contains the knowledge of agent 2 (who is hierarchically superior to agent 1) at time 3, and so on... The overall semantics of a system consisting of n agents at time t is given by $\boxplus_{t,n}\mathcal{P}$. Let us consider the following simple example:

Example 12 *In February 97, the President of Brazil passed a law determining that, in order to guarantee the safety aboard public transportation airplanes, all weapons are forbidden. Furthermore, all exceptional situations that, due to public interest, require an armed law enforcement or military agent are to be the subject of specific regulation by the Military and Justice Ministries. We will refer to this as rule 1. At the time of this event, there was in force an internal norm of the Department of Civil Aviation stating that "Armed Forces Officials and Police Officers can board with their weapons if their destination is a national airport". We will refer to this as rule 2. Restricting ourselves to the essential parts of these regulations, they can be encoded by the following generalized logic program clauses:*

$$
\begin{aligned}
rule1: &\quad not\, carry\_weapon \leftarrow not\, exception \\
rule2: &\quad carry\_weapon \leftarrow armed\_officer
\end{aligned}
$$

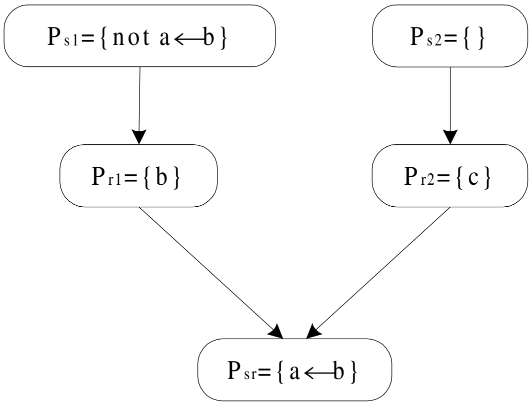

Figure 4:

Let us consider 2 distinct dimensions corresponding to the two principles that govern this situation: Lex Posterior (d_1) *and* Lex Superior (d_2). *Agent 1 represents the Department of Civil Aviation and Agent 2 represents the President of Brazil. We will consider two time points representing the time when the two regulations were enacted. We have then a graph whose vertices are* $\{(1,1),(1,2),(2,1),(2,2)\}$ *(in the form (agent,time)) and whose edges are the obvious subset of those depicted in Fig.3. We have that* $P_{1,1}$ *contains rule 2,* $P_{2,2}$ *contains rule 1 and the other two programs are empty. Let us further assume that there is an armed_officer represented by a fact in* $P_{1,1}$. *Applying Def.6, for* $M_{2,2} = \{not\ carry\_weapon,\ not\ exception,\ armed\_officer\}$ *at state* $(2,2)$ *we have that:*

$$Reject((2,2), M_{2,2}) = \{carry\_weapon \leftarrow armed\_officer\}$$
$$Default\,(\mathcal{P}_{2,2}, M_{2,2}) = \{not\,exception\}$$

it is trivial to see that

$$M_{2,2} = least\,([\mathcal{P}_{2,2} - Reject((2,2), M_{2,2})] \cup Default\,(\mathcal{P}_{2,2}, M_{2,2}))$$

which means that in spite of rule 2, since the exceptions have not been regulated yet, rule 1 prevails for all situations and no one can carry a weapon aboard an airplane. This would correspond to the only stable model of $\boxplus_{2,2}\mathcal{P}$. $\diamond$

In what concerns the usability of MDLP in a multi-agent context, it is not limited to the assignment of a single semantics to the overall system, i.e., the multi-agent system does not have to be described by a single DAG. Instead we could determine each agent's view of the world by associating with each agent a DAG representing its view of the relationships with other agents and amongst themselves.

Example 13 *Consider a society of agents representing a hierarchically structured research group. We have the Senior Researcher (A_{sr}), two Researchers (A_{r1} and A_{r2}) and two students (A_{s1} and A_{s2}) supervised by the two Researchers. The hierarchy is depicted in Fig.4, which also represents the view of the Senior Researcher. Typically, students think they are always right and do not like hierarchies, so their view of the*

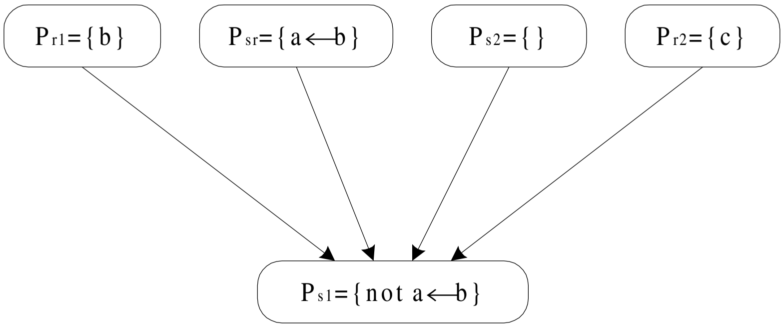

Figure 5:

community is quite different. Fig5 depicts one possible view by A_{s1}. In this scenario, we could use MDLP to determine and eventually compare A_{sr}'s view, given by $\boxplus_{sr}\mathcal{P}$ in Fig.4, with A_{s1}'s view, given by $\boxplus_{s1}\mathcal{P}$ in Fig.5. If we assign the following simple logic programs to the 5 agents:

$$P_{sr} = \{a \leftarrow b\} \qquad P_{r1} = \{b\} \qquad P_{r2} = \{c\} \qquad P_{s1} = \{not\, a \leftarrow c\} \qquad P_{s2} = \{\}$$

we have that $\boxplus_{sr}\mathcal{P}$ in Fig.4 has $M_{sr} = \{a, b, c\}$ as the only stable model, and $\boxplus_{s1}\mathcal{P}$ in Fig.5 has $M_{s1} = \{not\, a, b, c\}$ as the only stable model. That is, according to the student A_{s1}'s view of the world a is false, while according to the senior researcher A_{sr}'s view of the world a is true. ◇

5 Conclusions and Future Work

In this paper we introduced the notion of *Multi-dimensional Dynamic Logic Programming*. We have presented its semantics based on a characterization of its stable models, as well as by means of two syntactic transformations into a logic program which serve as the basis of an existing implementation.

Though the main motivation was to extend Dynamic Logic Programming to allow for the concurrent representation of several dimensions, be it within a single or a multi-agent system, MDLP turns out to be a useful practical framework to study the changes in behaviour of such multi-agent systems and how they hinge on the relationships amongst the agents i.e., on the current DAG that represents them. MDLP offers an important tool in the formal study of the social behaviour in multi-agent communities.

In what concerns future work, it can be discerned into development and integration. As for the former, we are currently exploring the dynamic updating of the indexing DAG, to reflect dynamic societies of agents. By specifying the edges of the graph by means of facts in the logic program, we already pave the way to allow them also to be described by rules, and therefore represent conditional edges. This set of logic program rules could, in turn, be subject to updates and represent evolving relationships between the states represented by the vertices. Note however that these updates would have to obey the constraint that the resulting graph be acyclic. We are also researching into the epistemic meaning and implications of dropping the acyclicity condition of the indexing digraph, and in particular to model computational steps and processes through synchronous activation of successive edges.

We are also exploring the relationship between MDLP and logic program composition [5].

Among the integration tasks, we foresee bringing into MDLP preferring facilities, such as described in [4] as well as abductive reasoning about updates.

To conclude, it is our opinion that *Multi-dimensional Dynamic Logic Programming* adds expressiveness and attractiveness to knowledge representation in logic programming and non-monotonic reasoning, and provides a scaffolding framework for inter-agent and intra-agent architectures.

References

[1] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Logic Programming.* In A. Cohn, L. Schubert and S. Shapiro (eds.), Procs. of KR'98. Morgan Kaufmann, 1998.

[2] J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Updates of Non-Monotonic Knowledge Bases.* The Journal of Logic Programming 45(1-3): 43-70, September/October 2000.

[3] J. J. Alferes, J. A. Leite, L. M. Pereira, P. Quaresma, *Planning as Abductive Updating*, In D. Kitchin (ed.), Procs. of AISB'00, 2000.

[4] J. J. Alferes and L. M. Pereira, *Updates plus Preferences*, M. O. Aciego, I. P. de Guzmán, G. Brewka and L. M. Pereira (eds.), Logics in Artificial Intelligence, Procs. of JELIA'00 European Workshop,Málaga, LNAI 1919, Springer, September-October 2000.

[5] A. Brogi, S. Contiero and F. Turini, Programming by combining general logic programs. Journal of Logic and Computation 9(1): 7-24, February 1999.

[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 K. Apt, V. Marek, M. Truszczynski and D. S. Warren (eds.), The Logic Programming Paradigm - A 25-Year Perspective, pages 5-32, Springer 1999.

[7] C. V. Damásio and L. M. Pereira. *Default negated conclusions: why not?* In R. Dyckhoff, H. Herre and P. Schroeder-Heister, editors, Procs. of ELP'96, LNAI 1050, pages 103-118. Springer-Verlag, 1996.

[8] D. De Schreye, M. Hermenegildo and L. M. Pereira, *Paving the Roadmaps: Enabling and Integration Technologies.* Available from `http://www.compulog.org/net/Forum/Supportdocs.html`

[9] P. Dell'Acqua, L. M. Pereira, *Updating Agents*, In S. Rochefort, F. Sadri and F. Toni (eds.), Procs. of the ICLP'99 Ws. on Multi-Agent Systems in Logic (MASL'99), 1999.

[10] P. Dell'Acqua, F. Sadri and F. Toni, *Combining Introspection and Communication with Rationality and Reactivity in Agents.* In J. Dix, F. Del Cerro and U. Furbach (eds.), Logics in Artificial Intelligence, LNCS 1489, Springer, 1998.

[11] M. Gelfond and V. Lifschitz. *Logic Programs with classical negation.* In Warren and Szeredi, editors, 7th Int. Conf. on LP, pages 579-597. MIT Press, 1990.

[12] N. Jennings, K. Sycara and M. Wooldridge. *A Roadmap of Agent Research and Development.* In Autonomous Agents and Multi-Agent Systems, 1, Kluwer, 1998.

[13] R. Kowalski and F. Sadri. *Towards a unified agent architecture that combines rationality with reactivity.* In D. Pedreschi and C. Zaniolo (eds), Procs. of LID'96, pages 137-149, Springer-Verlag, LNAI 1154, 1996.

[14] João A. Leite. *Logic Program Updates.* M.Sc. Dissertation, Universidade Nova de Lisboa, 1997.

[15] J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs.* In J.Dix, L.M. Pereira and T.C.Przymusinski (eds), Procs. of LPKR'97 Springer, LNAI 1471, 1998.

[16] J. A. Leite, F. C. Pereira, A. Cardoso and L. M. Pereira, *Metaphorical Mapping Consistency via Dynamic Logic Programming*, In G. Wiggins (ed.), Procs. of AISB, 2000.

[17] E. Lamma, F. Riguzzi and L. M. Pereira, *Strategies in Combined Learning via Logic Programs.* Machine Learning 38(1/2): 63-87, January 2000.

[18] 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`.

[19] I. Niemelä and P. Simons. *Smodels - an implementation of the stable model and well-founded semantics for normal logic programs.* In Procs. of the 4th LPNMR'97, Springer, July 1997.

[20] P. Quaresma and L. M. Pereira, *Modelling Agent Interaction in Logic Programming*, In O. Barenstein (ed.), Procs. of the INAP'98, Tokyo, Japan, September 1998.

[21] P. Quaresma, I. Rodrigues. *Using dynamic logic programming to model cooperative dialogues.* In AAAI'99 Fall Symposium on Modal and Temporal Logics based Planning for Open Networked Multimedia Systems. Cape Cod, USA. November 1999.

[22] F. Sadri and F. Toni. *Computational Logic and Multiagent Systems: a Roadmap*, 1999. Available from `http://www.compulog.org`.

[23] The XSB Group. *The XSB logic programming system, version 2.0*, 1999. Available from `http://www.cs.sunysb.edu/~sbprolog`.