

Multi-dimensional Logic Programming

JOÃO A. LEITE^{*†‡}, JOSÉ J. ALFERES^{†‡} and LUÍS MONIZ PEREIRA^{†‡}

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

Abstract

According to the paradigm 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 time (as in updates), specificity (as in taxonomies), strength of the updating instance (as in the legislative domain), hierarchical position of knowledge source (as in organizations), etc. The mutual relationships extant between different states are used to determine the semantics of the combined theory composed of all individual theories. Although suitable to encode one dimension (e.g. time, hierarchies, domains,...), *Dynamic Logic Programming* cannot deal with more than one such dimension simultaneously because it is only defined for a linear sequence of states. To overcome this limitation, we introduce, in this paper, the notion of *Multi-dimensional Dynamic Logic Programming*, which generalizes *Dynamic Logic Programming* by allowing collections of states represented by arbitrary acyclic digraphs. In this setting, *Multi-dimensional Dynamic Logic Programming* assigns semantics to sets and subsets of logic programs, depending on how they stand in relation to one another, these relations being defined by the acyclic digraph that represents the states. By dint of such natural generalization, *Multi-dimensional Dynamic Logic Programming* affords extra expressiveness, thereby enlarging the latitude of logic programming applications unifiable under a single framework. The generality and flexibility provided by the acyclic digraphs ensures a wide scope and variety of possibilities, some of which will be expounded in this paper.

1 Introduction and Motivation

Till recently, most of the work accomplished in the field of theory update (Winslett, 1988; Katsuno & Mendelzon, 1991; Marek & Truszczyński, 1994; Przymusiński & Turner, 1997), i.e. the study of changing worlds and how to incorporate new knowledge about them, was based on the idea of reducing the problem of finding an update of a knowledge base DB by another knowledge base U to the problem of finding the updates of its individual interpretations or models. The update of models is governed by the update rules specified in U , and also by inertia applied to the literals of the models that were not directly affected by the update program. This approach, while adequate for the purpose of updating theories in classical

* Partially supported by PRAXIS XXI scholarship no. BD/13514/97.

† Partially supported by project ACROPOLE (PRAXIS XXI PBIC/C/TIT/2519/95).

‡ Partially supported by project MENTAL (PRAXIS XXI 2/2.1/TIT/1593/95).

propositional logic (for which it was developed), when applied to non-monotonic theories suffers from a major drawback, as pointed out in (Leite & Pereira, 1997): it leads to counterintuitive results when the intensional part of the knowledge base (i.e. the set of rules) changes, as shown by the following example:

Example 1

Consider the logic program:

$$\begin{aligned} free &\leftarrow not\ jail \\ jail &\leftarrow jail_for_eutanasia \\ jail_for_eutanasia &\leftarrow eutanasia \end{aligned}$$

whose only stable model is $M = \{free\}$. Suppose now that the update U states that **eutanasia** becomes true, i.e. $U = \{eutanasia \leftarrow\}$. According to the interpretation approach to updating, we would obtain $\{free, eutanasia\}$ as the only update of M by U . However, by inspecting the initial program and the update, we are likely to conclude that, since **free** was true because **jail** could be assumed false, which was the case because **eutanasia** was false, now that **eutanasia** became true **jail_for_eutanasia** and **jail** should also have become true, and **free** should be removed from the conclusions.

Suppose now that the law changes, so that eutanasia no longer implies going to jail. That could be described by the new (update) program:

$$U_2 = \{not\ jail_for_eutanasia \leftarrow eutanasia\}$$

We should now expect **jail** to become false and so **free** to become true (again).

This example illustrates that, when updating non-monotonic knowledge bases, it is not sufficient to just consider the truth values of literals figuring in the heads of its rules because the truth value of their rule bodies may also be affected by updates on other literals. In other words, it suggests that the *principle of inertia* should be applied not just to the individual literals in an interpretation but rather to the *entire rules of the knowledge base*. And all the more so in Logic Programming because rules encode directional information concerning the entailment: Contrapositives are not warranted as in classical logic, but must be stated on their own if so desired. The import of directionality is even more patent when updating is involved: The final update rule of the example states that if you assist in eutanasia you will not be in jail for it, but it does not purport to assert the contrapositive, i.e. that if you are in jail for eutanasia then you will not have assisted it. In the sequel, use of negated heads interpreted as deletion will be elaborated upon.

Newton's first law, also known as the law of inertia, states that: "*every body remains at rest or moves with constant velocity in a straight line, unless it is compelled to change that state by an unbalanced force acting upon it*" (adapted from (Newtono, 1726)). One often tends to interpret this law in a commonsensical way, as things keeping as they are unless some kind of force is applied on them. This is true but does not exhaust the meaning of the law. It is the result of all applied forces that governs the outcome. Take a body to which several forces are applied,

and which is in a state of equilibrium due to all those forces canceling out. Later one or more forces are removed and the body starts to move.

In (Leite & Pereira, 1997), the authors suggested that the same kind of behaviour would be present when updating programs. Before ascertaining the truth value, by simple inertia, of those elements not directly affected by the update program, one should verify whether the truth of such elements is not indirectly affected by the direct updating of other elements. That is, the body of a rule may act as a force sustaining the truth of its head, but this force may be withdrawn if and when the body becomes false. Another way to view program updating, and envisage in particular the role of inertia, is to say that the rules themselves of the initial program carry over to the updated program due to inertia, instead of simply the truth of their conclusion literals. Just in case, of course, the rules are not actually overruled by the update program. On the basis of this reading of the *inertia principle*, a declarative and transformational semantics for logic program updates was then proposed. These were later refined and generalized, and gave rise to the paradigm of *Dynamic Logic Programming* (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a).

According to *Dynamic Logic Programming*, itself a generalization of the notion of the update of a logic program P by another one U , knowledge is given by a series of theories (encoded as generalized logic programs) representing distinct supervenient states of the world. Different states, sequentially ordered, can represent different time periods, different hierarchical instances, or even different domains of knowledge. Consequently, individual theories may comprise mutually contradictory as well as overlapping information. The role of *Dynamic Logic Programming* is to employ the mutual relationships extant among different states to precisely determine the declarative as well as the procedural semantics for the combined theory comprised of all individual theories at each state. Intuitively, one can add, at the end of the sequence, newer or more specific rules (arising from new, newly acquired, or more specific knowledge) leaving to *Dynamic Logic Programming* the task of ensuring that these added rules are in force, and that previous or less specific rules are still valid (by inertia) only so far as possible, i.e. that they are kept for as long as they are not in conflict with newly added ones, which always prevail.

In (Alferes *et al.*, 1999a), the authors proposed a language (*LUPS*) designed for specifying changes (updates) to logic programs. A source, integrative review, and synthesis of some of the recent developments concerning the evolution of logic programs by means of updates can be found in (Alferes & Pereira, 2000a) to induct the reader to the topic.

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 (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a). 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 (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a);
- combine rules learnt by a diversity of agents (Lamma *et al.*, 2000);

- reason about updates of agents' beliefs (Dell'Acqua & Pereira, 1999);
- model agent interaction (Quaresma & Pereira, 1998; Quaresma & Rodrigues, 1999);
- model and reason about actions (Alferes *et al.*, 2000b);
- resolve inconsistencies in metaphorical reasoning (Leite *et al.*, 2000);

The common property 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,...). This is so inasmuch *DLP* is defined for linear sequences of states alone.

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 such 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 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. *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.

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). Some of these examples will be further elaborated upon in a subsequent section. For this to be possible, *DLP* needs to be extended to allow for a more general structure of states.

In this paper we introduce for the first time the notion of *Multi-dimensional Dynamic Logic Programming (MDLP)*, which generalizes *DLP* to allow for collections of states represented by arbitrary acyclic digraphs. In this setting, *MDLP* assigns semantics to sets and subsets of logic programs, depending on how they stand in relation to one another, these relations being defined by the acyclic digraph (*DAG*) that represents the states. By dint of such natural generalization, *MDLP* affords extra expressiveness, thereby enlarging the latitude of logic programming applications unifiable under a single framework. The generality and flexibility provided by the *DAGs* ensures a wide scope and variety of possibilities, of which some will be expounded in this paper.

It is our opinion that, by virtue of these newly added characteristics of multiplicity and composition, *MDLP* provides a “societal” viewpoint in *Logic Programming*, important in these web and agent days, for combining knowledge in general.

The remainder of this paper is structured as follows: In Section 2 we recapitulate some notions used throughout the paper, namely those concerning generalized logic programs, dynamic logic programming and graphs; In Section 3 we introduce *Multi-dimensional Dynamic Logic Programming*, propose a declarative semantics and set forth some basic properties; In Section 4 we present an equivalent semantics based on a syntactical transformation proven sound and complete wrt. the declarative semantics. Also in Section 4 we establish *DLP* to be a special case of *MDLP*; The transformation of Section 4 is the basis for an existing implementation which is briefly described in Section 5; In Section 6 we explore a few application domains by means of examples; In Sect 7 we conclude and open doors for future developments.

2 Background

In this section, with the purpose of self containment, we will recap the main notions used throughout the paper. These include the syntax and semantics of *generalized logic programs*, the notion of *dynamic logic programming* and some basic definitions of *graph theory*.

2.1 Generalized Logic Programs and their Stable Models

In order to represent *negative* information in logic programs and in their updates, we need more general logic programs that allow default negation *not A* not only in premises of their clauses but also in their heads. We call such programs *generalized logic programs* and in this section we recall the semantics of such programs, as defined in (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a).

It is worth noting why, in the update setting, generalizing the language to allow default negation in rule heads is more adequate than introducing explicit negation in programs (both in heads and bodies). In updates a *not A* head means atom *A* is deleted if the body holds. Deleting *A* means that *A* is no longer true, not necessarily that it is false. In fact, in situations where closed world assumption is not desired for some atoms (i.e. where so atom are not to be assumed by default), when deleting *A* one does not necessarily want *A* do become false. Using explicitly negated atoms in rule heads to cater for deletions would necessarily imply that the deletion of *A* makes *A* false.

The class of generalized logic programs can be viewed as a special case of yet broader classes of programs, introduced earlier in (Lifschitz & Woo, 1992). As shown in (Alferes *et al.*, 1998b), their semantics coincides with the stable models semantics (Gelfond & Lifschitz, 1988) for the special case of normal programs. Moreover, the semantics also coincides with the one in (Lifschitz & Woo, 1992) when the latter is restricted to the language of generalized programs. Note however that, unlike single generalized programs (cf. (Inoue & Sakama, 1998)), in updates the head *not s* cannot be moved freely into the body, to obtain simple denials: there is inescapable pragmatic information in specifying exactly which *not* literal figures in the head, namely the one being deleted when the body holds true, and thus it is not indifferent that any other (positive) body literal in the denial could be moved to the head. For

example, in an update setting, the rule $\text{not } a \leftarrow b$ should be different from the rule $\text{not } b \leftarrow a$: the first states that a is to be deleted whenever b is true while the latter states that b is to be deleted whenever a is true. In single generalized programs, as proven in (Inoue & Sakama, 1998), these two rules are equivalent (and are also equivalent to the denial $\leftarrow a, b$).

For our purposes, it will be convenient to *syntactically* represent generalized logic programs as *propositional Horn theories*. In particular, we will represent default negation $\text{not } A$ 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:

$$\{A : A \in \mathcal{K}\} \cup \{\text{not } A : A \in \mathcal{K}\}. \quad (1)$$

Atoms $A \in \mathcal{K}$, are called *objective atoms* while the atoms $\text{not } A$ 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 of the form:

$$L \leftarrow L_1, \dots, L_n \quad (2)$$

where L and L_i are atoms from $\mathcal{L}_{\mathcal{K}}$. If r is a clause (or rule), by $\text{head}(r)$ we mean L . If $\text{head}(r) = A$ (resp. $\text{head}(r) = \text{not } A$) then $\text{not head}(r) = \text{not } A$ (resp. $\text{not head}(r) = A$). 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 $\text{not } A$ belongs to M . Given an interpretation M we define:

$$\begin{aligned} M^+ &= \{A \in \mathcal{K} : A \in M\} \\ M^- &= \{\text{not } A : A \in \mathcal{K}, \text{not } A \in M\} = \{\text{not } A : A \in \mathcal{K}, A \notin M\}. \end{aligned} \quad (3)$$

By a (2-valued) *model* M of a generalized logic program P we mean a (2-valued) interpretation of P that satisfies all of its clauses. A program is called *consistent* if it has a model. A model M is considered *smaller* than a model N if the set of *objective* atoms of M is properly contained in the set of objective atoms of N . A model of P is called *least* if it is the smallest model of P . Every consistent program P has the least model $M = \{A : A \text{ is an atom and } P \vdash A\}$.

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 $P \cup M^-$:

$$M = \text{least}(P \cup M^-), \quad (4)$$

or, equivalently, if $M = \{A : A \text{ is an atom and } P \cup M^- \vdash A\}$.

Example 2

Consider the program:

$$\begin{array}{llll} a & \leftarrow & \text{not } b & \quad c \leftarrow b \quad e \leftarrow \text{not } d \\ \text{not } d & \leftarrow & \text{not } c, a & \quad d \leftarrow \text{not } e \end{array} \quad (5)$$

and let $\mathcal{K} = \{a, b, c, d, e\}$. This program has precisely one stable model $M = \{a, e, \text{not } b, \text{not } c, \text{not } d\}$. To see that M is stable we simply observe that:

$$M = \text{least}(P \cup \{\text{not } b, \text{not } c, \text{not } d\}). \quad (6)$$

On the other hand, the interpretation $N = \{\text{not } a, \text{not } e, b, c, d\}$ is not a stable model because:

$$N \neq \text{least}(P \cup \{\text{not } e, \text{not } a\}) = \{d, \text{not } a, \text{not } e\}. \quad (7)$$

Following an established tradition, whenever convenient we will be omitting the default (negative) atoms when describing interpretations and models. Thus the above model M will be simply listed as $M = \{a, e\}$.

2.2 Dynamic Logic Programming

Here, we will recap the notion of *dynamic program update* $\bigoplus \{P_s : s \in S\}$ over a set of logic programs $\mathcal{P} = \{P_s : s \in S\}$ ordered by a linear sequence of states $S = \{1, 2, \dots, n, \dots\}$ (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a). The idea of dynamic updates, inspired by (Leite, 1997), 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. Intuitively such a sequence may be viewed as the result of, starting with program P_1 , updating it with program P_2 , \dots , and updating it with program P_n . This notion represents a generalization of the update of a logic program P , by another logic program U , $P \oplus U$, whose rationale it is to apply the *principle of inertia* to the rules of the initial program P , so as long as they do not conflict with the newly added rules of U (Leite & Pereira, 1997). 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 $S =$

$\{1, 2, \dots, n, \dots\}$ 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 $\bar{\mathcal{K}}$ we denote the following superset of the set \mathcal{K} of propositional variables:

$$\bar{\mathcal{K}} = \mathcal{K} \cup \{A^-, A_s, A_s^-, A_{P_s}, A_{P_s}^- : A \in \mathcal{K}, s \in S \cup \{0\}\} \quad (8)$$

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

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 $\uplus\mathcal{P}$, which consists of the following clauses in the extended language $\bar{\mathcal{L}}$:

(RP) Rewritten Program Clauses

$$A_{P_s} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad (9)$$

$$A_{P_s}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad (10)$$

for any clause

$$A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (11)$$

respectively, for any clause

$$\text{not } A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (12)$$

in the program P_s , where $s \in S$.

(IR) Inheritance Rules

$$A_s \leftarrow A_{s-1}, \text{ not } A_{P_s}^- \quad (13)$$

$$A_s^- \leftarrow A_{s-1}^-, \text{ not } A_{P_s} \quad (14)$$

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

(UR) Update Rules

$$A_s \leftarrow A_{P_s} \quad (15)$$

$$A_s^- \leftarrow A_{P_s}^- \quad (16)$$

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

(DR) Default Rules

$$A_0^- \quad (17)$$

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

The dynamic program update $\uplus\mathcal{P}$ 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 $\uplus\mathcal{P}$ 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 $\oplus_s\mathcal{P}$ 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 $\oplus_s \mathcal{P}$, we mean the dynamic program update $\uplus \mathcal{P}$ augmented with the following *Current State Rules*:

(CS_s) Current State Rules

$$A \leftarrow A_s \quad (18)$$

$$A^- \leftarrow A_s^- \quad (19)$$

$$\text{not } A \leftarrow A_s^- \quad (20)$$

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 finite or infinite sets $V = V_D$ of *vertices* and $E = E_D$ of (*directed*) *edges* and a mapping $\delta : E \rightarrow V \times V$. 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, \dots, e_n such that $\delta(e_i) = (v_{i-1}, v_i)$ for $i = 1, \dots, n$. 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 $v = w$.

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. The *transitive closure* of a graph D is a graph $D^+ = (V, E^+)$ such that for all $v, w \in V$ there is an edge (v, w) in E^+ if and only if $v < w$ in D .

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

Definition 4 (Relevancy DAG wrt a vertex)

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:

$$V_v = \{v_i : v_i \in V \text{ and } v_i \leq v\} \quad (21)$$

$$E_v = \{(v_i, v_j) : (v_i, v_j) \in E \text{ and } v_i, v_j \in V_v\} \quad (22)$$

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 . We will also need the notion of Relevancy DAG wrt a set of vertices:

Definition 5 (Relevancy DAG wrt a set of vertices)

Let $D = (V, E)$ be an acyclic digraph. Let S be a set of vertices of D , i.e. $S \subseteq V$,

and for every $v \in S$ let $D_v = (V_v, E_v)$ be the relevancy *DAG* of D wrt v . The relevancy *DAG* of D wrt S is $D_S = (V_S, E_S)$ where:

$$V_S = \bigcup_{v \in S} V_v \quad (23)$$

$$E_S = \bigcup_{v \in S} E_v \quad (24)$$

Intuitively the relevancy *DAG* of D wrt S is the subgraph of D consisting of the union of the relevancy *DAG*s wrt all vertices in S .

3 Multi-dimensional Dynamic Logic Programming

As noted in the introduction, allowing the individual theories of a dynamic program update to only relate via a linear sequence of states, limits the use of *DLP* to represent and reason about a single evolving aspect of a system (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 pairwise relations by the corresponding graph edges, enabling thus to represent concurrently, depending on the particular choice of *DAG*, several interrelated dimensions of a representational updatable system. In particular, the *DAG* can represent not only a system with n independent dimensions, but also encompass inter-dimensional dependencies. In this setting, *MDLP* assigns semantics to sets and to subsets of logic programs, depending on how they stand in relation to one another.

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*, the relevancy *DAG* wrt v has a finite number of vertices.

Definition 6 (Multi-dimensional Dynamic Logic Program)

Let $\mathcal{L}_{\mathcal{K}}$ be a propositional language as described before. A *Multi-dimensional Dynamic Logic Program (MDLP)*, \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 finite or infinite set of generalized logic programs in the language $\mathcal{L}_{\mathcal{K}}$, indexed by the vertices $v \in V$ of D . We call *states* such vertices of D . For simplicity, we will often leave the language $\mathcal{L}_{\mathcal{K}}$ implicit.

3.1 Declarative Semantics

We want to characterize the models of \mathcal{P} at any given state. For this purpose, 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 the subset of rules of P comprised by those that are not rejected, i.e. do not carry over by inertia due to their being overridden by update 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 or a single descendent. 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 program $P_u \in \mathcal{P}_D$, at

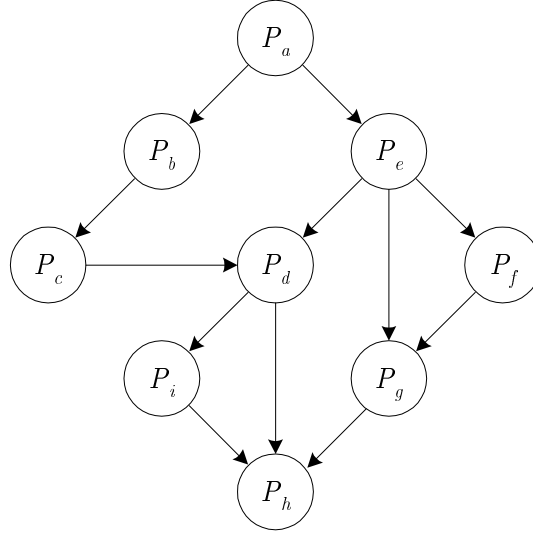


Fig. 1

v or some descendant of v , 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 this definition:

Definition 7 (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 \mathcal{P} at state $s \in V$* , iff:

$$M_s = \text{least}([\mathcal{P}_s - \text{Reject}(s, M_s)] \cup \text{Default}(\mathcal{P}_s, M_s)) \quad (25)$$

where

$$\mathcal{P}_s = \bigcup_{i \leq s} P_i \quad (26)$$

$$\text{Reject}(s, M_s) = \left\{ \begin{array}{l} r \in P_i \mid \exists r' \in P_j, i < j \leq s, \\ \text{head}(r) = \text{not head}(r') \wedge M_s \models \text{body}(r') \end{array} \right\} \quad (27)$$

$$\text{Default}(\mathcal{P}_s, M_s) = \{\text{not } A \mid \nexists r \in \mathcal{P}_s : (\text{head}(r) = A) \wedge M_s \models \text{body}(r)\} \quad (28)$$

Intuitively, the set $\text{Reject}(s, M_s)$ 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 any path to state s . \mathcal{P}_s contains all rules of all programs that are indexed by a state along all paths leading to state s , i.e. all rules that are potentially relevant to determine the semantics at state s . The set $\text{Default}(\mathcal{P}_s, M_s)$ 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 .

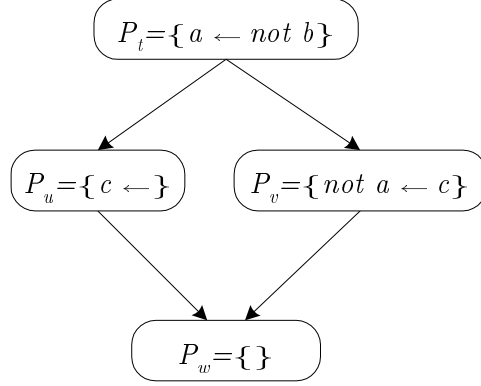


Fig. 2

Example 3

Consider the MDLP $\mathcal{P} = (\mathcal{P}_D, D)$ such that $\mathcal{P}_D = \{P_t, P_u, P_v, P_w\}$ where:

$$\begin{array}{ll} P_t = \{a \leftarrow \text{not } b\} & P_u = \{c \leftarrow\} \\ P_v = \{\text{not } a \leftarrow c\} & P_w = \{\} \end{array} \quad (29)$$

and $D = (V, E)$ where

$$V = \{t, u, v, w\} \quad (30)$$

$$E = \{(t, u), (t, v), (u, w), (v, w)\} \quad (31)$$

as depicted in Fig.2. The only stable model at state w is $M_w = \{\text{not } a, \text{not } b, c\}$. To confirm, we have that:

$$\text{Reject}(w, M_w) = \{a \leftarrow \text{not } b\} \quad \text{Default}(\mathcal{P}_w, M_w) = \{\text{not } b\} \quad (32)$$

and, finally,

$$[\mathcal{P}_w - \text{Reject}(s, M_w)] \cup \text{Default}(\mathcal{P}_w, M_w) = \{\text{not } a \leftarrow c; c \leftarrow; \text{not } b\} \quad (33)$$

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

Example 4

Consider the MDLP $\mathcal{P} = (\mathcal{P}_D, D)$ such that $\mathcal{P}_D = \{P_t, P_u, P_v, P_w\}$ where

$$\begin{array}{ll} P_t = \{d \leftarrow\} & P_w = \{\text{not } a \leftarrow b\} \\ P_u = \{a \leftarrow \text{not } e\} & b \leftarrow \text{not } c \\ P_v = \{\text{not } a \leftarrow d\} & c \leftarrow \text{not } b \end{array} \quad (34)$$

and $D = (V, E)$ where

$$V = \{t, u, v, w\} \quad (35)$$

$$E = \{(t, u), (t, v), (u, w), (v, w)\} \quad (36)$$

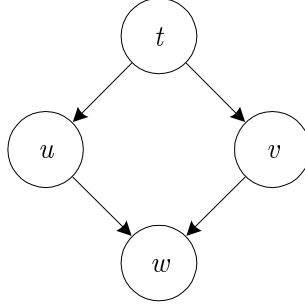


Fig. 3

as depicted in Fig. 3. The only stable model at state w is

$$M_w = \{\text{not } a, b, \text{not } c, d, \text{not } e\}$$

To confirm, we have that

$$\text{Reject}(w, M_w) = \{a \leftarrow \text{not } e\} \quad \text{Default}(\mathcal{P}_w, M_w) = \{\text{not } a, \text{not } c, \text{not } e\} \quad (37)$$

and, finally,

$$\begin{aligned} & [\mathcal{P}_w - \text{Reject}(s, M_w)] \cup \text{Default}(\mathcal{P}_w, M_w) = \\ & = \{d \leftarrow; \text{not } a \leftarrow d; \text{not } a \leftarrow b; b \leftarrow \text{not } c; c \leftarrow \text{not } b\} \cup \{\text{not } a, \text{not } c, \text{not } e\} \end{aligned} \quad (38)$$

whose least model is M_w . The reader can check that M_w is *the only* stable model at state w .

The following proposition establishes that when determining 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 .

Proposition 1

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Let s be a state in V . Let $\mathcal{P}' = (\mathcal{P}_{D_s}, D_s)$ be a *Multi-dimensional Dynamic Logic Program* where $D_s = (V_s, E_s)$ is the relevancy DAG of D wrt s , and $\mathcal{P}_{D_s} = \{P_v : v \in V_s\}$. M is a *stable model of \mathcal{P} at state s* iff M is a *stable model of \mathcal{P}' at state s* .

Proof

Simply recall that in Def. 7 the sets of rules \mathcal{P}_s , $\text{Reject}(s, M)$, and $\text{Default}(\mathcal{P}_s, M)$, that completely characterize the stable models at state s , only depend on the rules of the programs indexed by the relevancy DAG wrt s and on the relevancy DAG itself. Since they all coincide for \mathcal{P} and \mathcal{P}' , the proposition follows. \square

3.2 Multiple State Semantics

In the previous section we presented the semantics of a *Multi-dimensional Dynamic Logic Program* at a given state, by characterizing its stable models. But we might

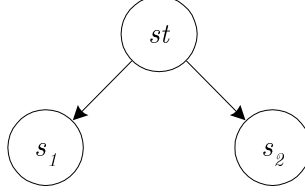


Fig. 4

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 may simply determine the models at that state (Prop. 1). But this might not be the case. For example, imagine a student (st) with two supervisors (s_1 and s_2) whose opinion on something prevails over that of the student, as depicted in Fig.4. If we determine the models at state s_1 we do not consider the rules from s_2 , and vice-versa. The student however might be interested in determining the semantics taking into account both supervisors, i.e. determining the models at a set of states.

For this purpose, we need to characterize the models at a set of states. Formally, the semantics of a *MDLP* at an arbitrary set of its states is determined according to the following definition:

Definition 8 (Stable Models at a set of states 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 $D = (V, E)$. 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

$$M_S = \text{least}([\mathcal{P}_S - \text{Reject}(S, M_S)] \cup \text{Default}(\mathcal{P}_S, M_S)) \quad (39)$$

where

$$\mathcal{P}_S = \bigcup_{s \in S} \left(\bigcup_{i \leq s} P_i \right) \quad (40)$$

$$\text{Reject}(S, M_S) = \left\{ \begin{array}{l} r \in P_i \mid \exists s \in S, \exists r' \in P_j, i < j \leq s, \\ \text{head}(r) = \text{not head}(r') \wedge M_S \models \text{body}(r') \end{array} \right\} \quad (41)$$

$$\text{Default}(\mathcal{P}_S, M_S) = \{\text{not } A \mid \nexists r \in \mathcal{P}_S : (\text{head}(r) = A) \wedge M_S \models \text{body}(r)\} \quad (42)$$

If we omit an explicit reference to S , we consider it, by default, to be equal to V , i.e. by the *stable models of \mathcal{P}* we mean the *stable models of \mathcal{P} at the set of states V* . If S contains only one state, then Def. 8 is equivalent to Def. 7:

Proposition 2

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Let $s \in V$ be a state and $S = \{s\}$. The *stable models of \mathcal{P} at the set of states S* coincide with the *stable models of \mathcal{P} at state s* .

Proof

Just note that (39), (40), (41) and (42) reduce to (25), (26), (27) and (28) respectively, when S contains only one state s . \square

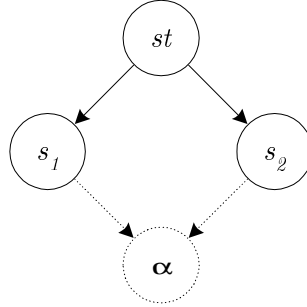


Fig. 5

Intuitively, Def. 8 considers the relevancy graphs wrt all states belonging to the set whose models are being determined.

In fact, it is equivalent to adding a new vertex α to the DAG, connecting all states we want to consider, by adding edges to this new state α , by making the program indexed by α , P_α , empty, and then, determining the stable models of this new MDLP at state α , as condoned by the following theorem:

Theorem 3

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$, such that $\alpha \notin V$. 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 is a *stable model of \mathcal{P}' at state α* where $\mathcal{P}' = (\mathcal{P}_{D_\alpha}, D_\alpha)$ is the *Multi-dimensional Dynamic Logic Program* such that $\mathcal{P}_{D_\alpha} = \mathcal{P}_D \cup P_\alpha$ and $D_\alpha = (V_\alpha, E_\alpha)$ where:

$$P_\alpha = \{\} \quad (43)$$

$$V_\alpha = V \cup \{\alpha\} \quad (44)$$

$$E_\alpha = E \cup \{(i, \alpha) : i \in S\} \quad (45)$$

The proof to this theorem is in Appendix A.

Fig.5 depicts this construction for the student/supervisor example of Fig.4. Note that since P_α is empty, it does not contribute to the rejection of any rules. In the student/supervisors example, intuitively, it has the effect of joining the rules of s_2 and s_1 .

Mark that the addition of state α does not affect the stable models at other states. This is so because α and the newly introduced edges do not belong to the relevancy DAG wrt any other state. States such as α are usefully introduced as *observation* nodes, because they provide *viewpoints* on the original DAG, without affecting the semantics at any of its original nodes.

Conceivably, this notion of observation node can be generalized to apply to a union of two (or more) Multi-dimensional Dynamic Logic Programs, $\mathcal{P}_1 = (\mathcal{P}_{D_1}, D_1)$ and $\mathcal{P}_2 = (\mathcal{P}_{D_2}, D_2)$, where the resulting DAG would be the union of both DAG's i.e., $D = D_1 \cup D_2 = \{V_1 \cup V_2, E_1 \cup E_2\}$.

Note further that since P_α is empty, stable models defined here coincide with those defined in Def.7, when $S = \{s\}$.

In the next subsection, we provide some equivalence preserving simplifications to this definition, according to which, a subset of the new edges added in E_α can be removed whilst preserving the stable models.

3.3 Properties of MDLP

In this section we study the basic properties of *Multi-dimensional Dynamic Logic Programming*.

The following theorem states that adding or removing edges from the DAG of a *Multi-dimensional Dynamic Logic Program* preserves the semantics if the transitive closure of the DAG is maintained. In particular, it allows the use of a transitive reduction of the original graph to determine the stable models.

Theorem 4 (DAG Simplification)

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Let $\mathcal{P}_1 = (\mathcal{P}_D, D_1)$ be a *Multi-dimensional Dynamic Logic Program*, where $D_1 = (V, E_1)$ such that $D^+ = D_1^+$. Then, for any state $s \in V$, M is a *stable model of \mathcal{P} at state s* iff M is a *stable model of \mathcal{P}_1 at state s* .

Proof

Since the relation $<$ (and \leq) is invariant wrt. the transitive closure of a DAG, we have that, $\forall u, v \in V$, the following holds:

$$u < v \text{ in } D \iff u < v \text{ in } D_1$$

$$u = v \text{ in } D \iff u = v \text{ in } D_1$$

therefore, according to Def.7, \mathcal{P}_s , $Reject(s, M)$ and $Default(\mathcal{P}_s, M)$ coincide for both \mathcal{P} and \mathcal{P}_1 , and thus, their stable models coincide. \square

The following corollary says that to determine the stable models at a set of states we only need to connect the sinks of the relevancy DAG wrt that set of states, to the new node α .

Corollary 5

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. M_S is a *stable model of \mathcal{P} at states $S \subseteq V$* iff M_S is a *stable model of \mathcal{P}_α at state α* where $\mathcal{P}_\alpha = (\mathcal{P}_{D_\alpha}, D_\alpha)$ is the *Multi-dimensional Dynamic Logic Program* such that $\mathcal{P}_{D_\alpha} = \mathcal{P}_D \cup P_\alpha$ and $D_\alpha = (V_\alpha, E_\alpha)$ where:

$$P_\alpha = \{\} \tag{46}$$

$$V_\alpha = V \cup \{\alpha\} \tag{47}$$

$$E_\alpha = E \cup \{(i, \alpha) : i \in S \text{ and } i \text{ is a sink of } D_S\} \tag{48}$$

where D_S is the relevancy DAG of D wrt S .

The following proposition relates the stable models of normal logic programs with the stable models of *MDLPs* whose set of programs only contains normal logic programs.

Proposition 6

Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a *Multi-dimensional Dynamic Logic Program*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Let $S \subseteq V$ be a set of states and $D_S = (V_S, E_S)$ the relevancy DAG of D wrt S . If all $P_v : v \in V_S$ are normal logic programs, then M is a *stable model of \mathcal{P} at states S* iff M is a *stable model* of the (normal) logic program

$$\bigcup_{v \in V_S} P_v \quad (49)$$

Proof

The stable models of \mathcal{P} at states S are, according to Theorem 3, the stable models at state α of \mathcal{P}_α . Since the programs indexed by the vertices of the relevancy DAG wrt α are normal logic programs (note that $P_\alpha = \{\}$), i.e. without default atoms in the heads of clauses, from Def. 7 we conclude that for any interpretation M , $\text{Reject}(\alpha, M) = \{\}$. Therefore, M is a stable model iff

$$M = \text{least}(\mathcal{P}_\alpha \cup \text{Default}(\mathcal{P}_\alpha, M)) \quad (50)$$

where

$$\mathcal{P}_\alpha = \bigcup_{v \in V_S} P_v \quad (51)$$

$$\text{Default}(\mathcal{P}_\alpha, M) = \{\text{not } A \mid \nexists r \in \mathcal{P}_\alpha : (\text{head}(r) = A) \wedge M \models \text{body}(r)\} \quad (52)$$

Since $\text{Default}(\mathcal{P}_\alpha, M)$ contains the same default atoms as M^- in Def. 1, (50) reduces to the definition of the stable models of \mathcal{P}_α , i.e.

$$M = \text{least}(\mathcal{P}_\alpha \cup M^-) \quad (53)$$

□

3.4 Adding Strong Negation

The class of generalized logic programs, used throughout this article, can easily be extended with *strong negation* $\neg A$ ((Gelfond & Lifschitz, 1990; Alferes *et al.*, 1996; Alferes *et al.*, 1998a)). This permits the use of this form of negation in multi-dimensional dynamic logic programming.

Definition 9 (Adding strong negation)

Let \mathcal{K} be an arbitrary set of propositional variables. In order to add strong negation to the language $\mathcal{L} = \mathcal{L}_\mathcal{K}$ we just augment the set \mathcal{K} with new propositional symbols $\{\neg A : A \in \mathcal{K}\}$, obtaining the new set \mathcal{K}^* , and consider the extended language $\mathcal{L}^* = \mathcal{L}_{\mathcal{K}^*}$. In order to ensure that A and $\neg A$ cannot be both true we also assume, for all $A \in \mathcal{K}$, the following strong negation axioms (**SN**), themselves generalized logic program clauses:

$$\text{not } A \leftarrow \neg A \quad (54)$$

$$\text{not } \neg A \leftarrow A. \quad (55)$$

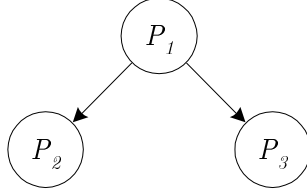


Fig. 6

In order to use strong negation in a multi-dimensional dynamic logic program, all needs doing is to add the strong negation axioms (SN) to the state whose models are being determined.

The following example, adapted from (Buccafurri *et al.*, 1999)¹, shows the use of strong negation:

Example 5

Consider the set of security specification, about a hierarchy of objects, represented by the following three programs indexed by the DAG of Fig.6:

$$\begin{aligned}
 P_1 &= \{ \text{authorize}(\text{bob}) \leftarrow \text{not } \text{authorize}(\text{ann}) \\
 &\quad \text{authorize}(\text{ann}) \leftarrow \text{not } \text{authorize}(\text{tom}), \text{not } \neg \text{authorize}(\text{alice}) \\
 &\quad \text{authorize}(\text{tom}) \leftarrow \text{not } \text{authorize}(\text{ann}), \text{not } \neg \text{authorize}(\text{alice}) \} \\
 P_2 &= \{ \neg \text{authorize}(\text{alice}) \leftarrow \} \\
 P_3 &= \{ \neg \text{authorize}(\text{bob}) \leftarrow \}
 \end{aligned}$$

The strong negation axioms (SN)² are:

$$\begin{aligned}
 SN &= \{ \text{not } \text{authorize}(X) \leftarrow \neg \text{authorize}(X) \\
 &\quad \text{not } \neg \text{authorize}(X) \leftarrow \text{authorize}(X) \}
 \end{aligned}$$

To determine the stable models at state 2, we add SN to P_2 and obtain, as the only stable model:

$$M_2 = \{ \neg \text{authorize}(\text{alice}), \text{authorize}(\text{bob}) \}$$

Accordingly, to determine the stable models at state 3, we add SN to P_3 and obtain the following two stable models:

$$\begin{aligned}
 M_3 &= \{ \text{authorize}(\text{ann}), \neg \text{authorize}(\text{bob}) \} \\
 M'_3 &= \{ \neg \text{authorize}(\text{tom}), \neg \text{authorize}(\text{bob}) \}
 \end{aligned}$$

4 Transformational Semantics for MDLP

Definition 7 above, establishes the semantics of *Multi-dimensional Dynamic Logic Programming* by characterizing its stable models at each state. Next we present

¹ The only difference from the original lies in the fact that, as usual, we encode disjunction as an even loop through default negation.

² As usual, rules with variables stand for all their ground instances.

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 equivalence relation with the stable models of the multi-dimensional dynamic logic program previously characterized. This transformation also provides 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 . Similarly, if we want to query a set of states, all needs doing is extending the *Multi-dimensional Dynamic Logic Program* with a new state α , as in Theorem 3, prior to the transformation. For our purposes, we will extend $\bar{\mathcal{K}}$ with an ersatz predicate *reject/1*. Therefore, from now on $\bar{\mathcal{K}}$ denotes the superset of the set \mathcal{K} of propositional variables:

$$\bar{\mathcal{K}} = \mathcal{K} \cup \{A^-, A_s, A_s^-, A_{P_s}, A_{P_s}^-, reject(A_s), reject(A_s^-) : A \in \mathcal{K}, s \in V \cup \{s_0\}\} \quad (56)$$

Definition 10 (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 $\bar{\mathcal{L}}$, where $D_s = (V_s, E_s)$ is relevancy DAG of D wrt s :

(RP) Rewritten program clauses:

$$A_{P_v} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad (57)$$

or

$$A_{P_v}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad (58)$$

for any clause:

$$A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (59)$$

respectively, for any clause:

$$\text{not } A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (60)$$

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:

$$A_v \leftarrow A_u, \text{ not } reject(A_u) \quad (61)$$

$$A_v^- \leftarrow A_u^-, \text{ not } reject(A_u^-) \quad (62)$$

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:

$$reject(A_u^-) \leftarrow A_{P_v} \quad (63)$$

$$reject(A_u) \leftarrow A_{P_v}^- \quad (64)$$

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:

$$A_v \leftarrow A_{P_v} \quad (65)$$

$$A_v^- \leftarrow A_{P_v}^- \quad (66)$$

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:

$$A_{s_0}^- \quad (67)$$

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

(CS_s) Current State Rules:

$$A \leftarrow A_s \quad (68)$$

$$A^- \leftarrow A_s^- \quad (69)$$

$$not A \leftarrow A_s^- \quad (70)$$

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 $not A$.

This transformation 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 need not be so: one can also declaratively specify, by means of a logic program, the notion of relevancy graph. This will be further explored in Sect.5.

Example 6

Consider again the MDLP $\mathcal{P} = (\mathcal{P}_D, D)$ of Example 3, where $\mathcal{P}_D = \{P_t, P_u, P_v, P_w\}$,

$$\begin{aligned} P_t &= \{a \leftarrow not b\} & P_u &= \{c \leftarrow\} \\ P_v &= \{not a \leftarrow c\} & P_w &= \{\} \end{aligned} \quad (71)$$

and $D = (V, E)$ where

$$V = \{s_0, t, u, v, w\} \quad (72)$$

$$E = \{(t, u), (t, v), (u, w), (v, w), (s_0, t)\} \quad (73)$$

The program $\boxplus_w \mathcal{P}$ contains the rules:

$$a_{P_t} \leftarrow b^- \quad (74)$$

$$c_{P_u} \leftarrow \quad (75)$$

$$a_{P_v}^- \leftarrow c \quad (76)$$

$$A_i \leftarrow A_j, \text{ not reject}(A_j) \quad \forall (j, i) \in E, \forall A \in \{a, b, c\} \quad (77)$$

$$A_i^- \leftarrow A_j^-, \text{ not reject}(A_j^-) \quad \forall (j, i) \in E, \forall A \in \{a, b, c\} \quad (78)$$

$$\text{reject}(A_i^-) \leftarrow A_{P_j} \quad \forall i, j \in V : i < j, \forall A \in \{a, b, c\} \quad (79)$$

$$\text{reject}(A_i) \leftarrow A_{P_j}^- \quad \forall i, j \in V : i < j, \forall A \in \{a, b, c\} \quad (80)$$

$$A_i \leftarrow A_{P_i} \quad \forall i \in V, \forall A \in \{a, b, c\} \quad (81)$$

$$A_i^- \leftarrow A_{P_i}^- \quad \forall i \in V, \forall A \in \{a, b, c\} \quad (82)$$

$$A_{s_0}^- \quad \forall A \in \{a, b, c\} \quad (83)$$

$$A \leftarrow A_w \quad \forall A \in \{a, b, c\} \quad (84)$$

$$A^- \leftarrow A_w^- \quad \forall A \in \{a, b, c\} \quad (85)$$

$$\text{not } A \leftarrow A_w^- \quad \forall A \in \{a, b, c\} \quad (86)$$

whose only stable model, restricted to the initial language is $M_w = \{\text{not } a, \text{not } b, c\}$, as expected.

As mentioned before, the stable models of the program obtained by the previous transformation coincide with those characterized in Def.7, as expressed in the following theorem:

Theorem 7

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 \mathcal{P} at state s , according to Def.7.

The proof to this theorem is in Appendix B.

4.1 Relationship to Dynamic Logic Programming

Since this work is rooted in *Dynamic Logic Programming*, it should properly extend it, i.e. *Multi-dimensional Dynamic Logic Programming* should be a generalization of *Dynamic Logic Programming*. The following Theorem guarantees this:

Theorem 8 (Embedding of DLP)

Let $\mathcal{P}_D = \{P_s : s \in S\}$ be a finite or infinite sequence of generalized logic programs in the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$, indexed by set of natural numbers $S = \{1, 2, 3, \dots, n, \dots\}$. Let $\mathcal{P} = (\mathcal{P}_D, D)$ be the *Multi-dimensional Dynamic Logic Program*, where $D = (S, E)$ is the acyclic digraph such that $E = \{(1, 2), (2, 3), \dots, (n-1, n), \dots\}$. 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 , $\boxplus_s \mathcal{P}_D$, if and only if N is the extension $N = \overline{M}$ of an interpretation M such that M is a stable model of \mathcal{P} at state s .

The proof to this theorem is in Appendix C.

In (Alferes *et al.*, 1998b; Alferes *et al.*, 2000a), *Dynamic Logic Programming* was defined by means of the transformational semantics of Defs.2 and 3. Theorems 7 and 8 immediately provide us with the following alternative characterization of *Dynamic Logic Programming*.

Corollary 9 (Characterization of Stable Models of Dynamic Logic Programming)

Let $\bigoplus \mathcal{P} = \bigoplus \{ P_s : s \in S \}$ be a *Dynamic Program Update*, let $s \in S$. An interpretation M_s is a *stable model of $\bigoplus \mathcal{P}$ at state s* iff:

$$M_s = \text{least} ([\mathcal{P}_s - \text{Reject}(s, M_s)] \cup \text{Default}(\mathcal{P}_s, M_s)) \quad (87)$$

where

$$\mathcal{P}_s = \bigcup_{i \leq s} P_i \quad (88)$$

$$\text{Reject}(s, M_s) = \left\{ \begin{array}{l} r \in P_i \mid \exists r' \in P_j, i < j \leq s, \\ \text{head}(r) = \text{not head}(r') \wedge M_s \models \text{body}(r') \end{array} \right\} \quad (89)$$

$$\text{Default}(\mathcal{P}_s, M_s) = \{ \text{not } A \mid \nexists r \in \mathcal{P}_s : (\text{head}(r) = A) \wedge M_s \models \text{body}(r) \} \quad (90)$$

In (Alferes *et al.*, 1998b), the authors have shown that *interpretation updates*, originally introduced under the name “*revision programs*” by Marek and Truszczyński (Marek & Truszczyński, 1994) constitute a special case of program updates. This result, together with the previous theorem, immediately imply the corollary:

Corollary 10 (Generalization of Interpretation Updates)

Multi-dimensional Dynamic Logic Programming generalizes *Interpretation Updates* in the sense of (Marek & Truszczyński, 1994).

5 On the Implementation

The transformation of a *MDLP* into a single, equivalent, logic program directly provides us with a means to implement *MDLP*. Via a preprocessor that, given a *MDLP* and a state s , produces the corresponding transformed logic program, the computation of the stable models at state s is reduced to the computation of the stable models of the logic program. For the latter purpose, a system that computes stable models of logic programs, such as the DLV-system (DLV, 2000) or smodels (Niemelä & Simons, 1997) can be used.

A preprocessor, that translates *MDLP* programs into logic programs that can be run in the DLV-system, has been implemented by the authors (in Prolog) and is available at:

<http://centria.di.fct.unl.pt/~jja/updates/>

MDLPs can be given to that preprocessor, as a sequence of labeled programs, the DAG being provided as facts for predicate `edge(x,y)`, signifying that in the DAG there is an edge from program labeled x to program labeled y . The preprocessor also receives as input the label of the state at which to compute the stable models.

One point about this implementation deserves some more attention. The transformation requires that, for the inheritance and rejection rules, one rule be added

for every pair (u, v) in the relevancy DAG and, respectively, for every pair (u, v) where there is a path from u to v in the relevancy graph for state s . Of course this doesn't need to be done in such a specific manner. Instead, the preprocessor simply adds general rules with u and v as variables, plus some predicates for paths on graphs to be instantiated by each call with the appropriate values, and according to the definition. More precisely, the rules added for each predicate A in the language of the *MDLP* are:

$$\begin{aligned} A(V) &:- \text{edge}(U, V), A(U), \text{not reject}(A(U)). \\ A-(V) &:- \text{edge}(U, V), A-(U), \text{not reject}(A-(U)). \\ \text{reject}(A-(U)) &:- \text{relPath}(U, V), \text{Ap}(V). \\ \text{reject}(A(U)) &:- \text{relPath}(U, V), \text{Ap-}(V). \end{aligned}$$

$\text{relPath}(U, V)$ is a predicate that, given the predicate $\text{edge}/2$ determines the paths for U to V in the relevancy path for state s . I.e.:

$$\text{relPath}(U, V) :- \text{path}(U, V), \text{path}(U, s), \text{path}(V, s).$$

where $\text{path}(X, Y)$ is the usual predicate for determining paths in a graph. Note that the state identifier appearing in the transformation as subscript becomes in the implementation an extra argument of the predicate.

Another implementation developed by the authors for *MDLP*, and also to be found at the URL above, includes a preprocessor and a meta-interpreter for query answering under the well-founded semantics, and runs under XSB-Prolog (XSB-Prolog, 1999). Though not complete according to the SM-like semantics defined in this paper, this alternative implementation coincides with it on the broad class of stratified programs. This follows from the well known result that the well-founded and the stable semantics coincide on that class. And, for such programs, using XSB-Prolog may have some advantage over using DLV, including better efficiency in query answering, its less restrictive usage of variables and functors in programs, and its easier use for a query-answer interface. In fact, all of the existing systems for computing stable models, compute the whole stable model at a time, or sets thereof, and do not allow for an interactive Prolog-like interface, where the users poses queries to the system and gets answers in the form of variable-substitutions. Such an interactive query-answer interface is quite convenient for the applications of *MDLP* (e.g. those described in the next section), where typically the user wants to query the system about some predicate at some state, rather than wanting the whole model at that state.

In order to take advantage of the tabling mechanism of XSB-Prolog³, and to allow for a more flexible interactive use of the system, some modifications have been made to the transformation described in the previous section.

³ The description of tabling mechanisms is beyond the scope of this paper. In a nutshell, tabling relies on memoizing techniques which guarantee that each predicate call is executed exactly once. Moreover, this is also used to ensure that program calls do never enter into infinite cycles. The reader may familiarize herself or himself with tabling mechanisms in e.g. (XSB-Prolog, 1999).

Both in the transformation and the preprocessor into DLV, the current state is treated as a constant given in the program included input query. In the preprocessor into XSB-Prolog, the transformation of the *MDLP* is made independently of any current state, which is then separately provided by the user at each query. This way, changing from querying one state to another state of the same DAG does not require any additional preprocessing of the *MDLP*. This can be achieved simply by adding an extra argument variable to every predicate in the transformed logic program, which is later instantiated by the query with the current state. This way, e.g. (57), (61), (64) and (68) become, respectively:

$$\begin{aligned} \text{Ap}(V, \text{Cur}) & :- \text{B1}(\text{Cur}), \dots, \text{C-n}(\text{Cur}). \\ \text{A}(V, \text{Cur}) & :- \text{edge}(U, V), \text{A}(U, \text{Cur}), \text{not reject}(\text{A}(U, \text{Cur})). \\ \text{reject}(\text{A}(U, \text{Cur})) & :- \text{relPath}(U, V, \text{Cur}), \text{Ap}(V, \text{Cur}). \\ \text{A}(\text{Cur}) & :- \text{A}(\text{Cur}, \text{Cur}). \end{aligned}$$

Another important difference between the preprocessor into XSB-Prolog and the transformation of section 4 was performed on the rejection rules, in order to take better advantage of tabling. Before showing how the rejection rules figure in the preprocessed XSB-program, let's take a brief look on how the XSB top-down interpreter executes a transformed program as per section 4.

To prove any given predicate *A* at current state *s*, the interpreter uses the corresponding current state rules, and calls $\text{A}(\mathbf{s}, \mathbf{s})$. This call may use inheritance rules that call various $\text{A}(U, \mathbf{s})$ for states *U* above *s* in the DAG (where by “a node *x* above a note *y*” we simply mean that there is a path in the DAG from *x* to *y*), until one of those $\text{A}(U, \text{Cur})$ succeeds via an update rule (if it doesn't succeed for any of them, then the predicate simply fails). The interpreter then looks, by using the rejection rule for $\text{reject}(\text{A}(U, \text{Cur}))$, for states between *U* and *s*, to check if *A* is rejected at one of those states.

Example 7

Consider the *MDLP* consisting of 4 programs, P_1, \dots, P_4 , where $P_1 = \{a\}$, $P_2 = \{\text{not } a\}$, and $P_3 = P_4 = \{\}$, and where the DAG has the following edges: (P_1, P_2) , (P_2, P_4) , (P_1, P_3) , and (P_3, P_4) .

To check whether *a* is true at state P_4 , the user calls $\mathbf{a}(\mathbf{p4})$. The interpreter then calls $\mathbf{a}(\mathbf{p4}, \mathbf{p4})$ (through a current state rule), $\mathbf{a}(\mathbf{p3}, \mathbf{p4})$, and $\mathbf{a}(\mathbf{p1}, \mathbf{p4})$ (both via an inheritance rule). The latter call succeeds by virtue of an update rule (since there is a rule in P_1 with head *a* and true body). The interpreter then calls $\text{reject}(\mathbf{a}(\mathbf{p1}, \mathbf{p4}))$ which checks if there is some rule in a state between P_1 and P_4 that rejects the rule for *a* in P_1 . This call eventually succeeds, by finding such a rule in P_3 . This causes a failure of the call $\mathbf{a}(\mathbf{p3}, \mathbf{p4})$, and consequently a failure also of $\mathbf{a}(\mathbf{p3}, \mathbf{p4})$. Note that this is the correct results as the only stable model at P_4 is $\{\text{not } a\}$.

Mark that the call for *a* at state $\mathbf{p3}$ fails. However the reader can easily check that the only stable model at P_3 is $\{a\}$. This means that what was computed for predicates in state P_3 , as an intermediate step of a call where the current state was P_4 , cannot be used when the call has current state P_3 . In other words, this entails that from calls with one current state to calls with another current state no memoized (i.e. tabled) results are of any use.

The above example shows that this transformation does not take advantage of the tabling mechanisms. However, a similar transformation can be defined where a better use of tabling is obtained, thus gaining in efficiency. This new transformation is the one implemented by our preprocessor into XSB. To do that, the transformation rules described in section 4, after proving predicate **A** with some rule at state **U**, must look for rules in states between **U** and **s** which reject it. Intuitively, the transformation implemented by the preprocessor now records in an extra predicate argument the state **U** where **A** is proven, and only checks for rejection just before inheriting the truth of **A** into state **s**. More precisely, rules (61), (64), (65) and (68) become, respectively⁴:

```

A(V,Proven,Cur) :- edge(U,V), A(U,Proven,Cur),
                    not reject(V,A(U,Proven,Cur)).
reject(V,A(U,Proven,Cur)) :- between(W,U,V), Ap(W,Cur).
A(V,Proven,Cur) :- Ap(V,Cur), Proven = V.
A(Cur) :- A(Cur,_,Cur).

```

where `between(X,Y,Z)` is a predicate that succeeds iff in the DAG vertex *X* is equal to *Z*, or is strictly between vertex *Y* and vertex *Z*.

Example 8

With this new transformation, to check whether *a* is true at state P_4 , in the MDLP of example 7, the following computation is made.

First the user calls `a(p4)`. The interpreter then calls `a(p4,X,p4)` (via a current state rule), `a(p3,X,p4)`, and `a(p1,X,p4)` (both via an inheritance rule). This last call succeeds, by using an update rule and instantiates **X** with **p1**. The interpreter then calls `reject(p1,a(p3,p1,p4))`, that fails because there is no state between **p1** and **p3** with a rule that rejects **a**. Thus `a(p3,X,p4)` succeeds with **X=p1**. However, for `a(p4,X,p4)` to succeed via the corresponding inheritance rule, the call to `reject(p1,a(p4,p1,p4))` must fail. But that call succeeds because there is a state between P_1 and P_4 with a rule that rejects *a*: namely the only rule in P_2 .

Note now that, despite the fact that the call for **a** at state **p4** fails, as it should, the call for **a** at state **p3** succeeds. With this transformation, subsequent calls to predicates with current state **p3**, can use the intermediate results for that state computed when the current state was another one below it in the DAG (in this case, **p4**). This way, a better usage of the memoizing mechanisms of XSB-Prolog can be had, and efficiency is gained.

The transformation implemented by our preprocessor into XSB-programs has some additional differences from the one of section 4. However, except for the ones that we describe above, all of them are of exiguous nature and result from mere implementation simplification. That is why they are not detailed here. The interested reader can check on them in the source code of the preprocessor available at the URL mentioned above.

⁴ The modification of rules (62), (63), (66) and (69) is similar, and is omitted for brevity.

6 Illustrative Examples

In this section, we discuss some domains where \mathcal{MDLP} can be applied, illustrating each of them with a small example, each having been run and tested on our implementation of \mathcal{MDLP} .

6.1 Organizational Decision Making

By its very motivation and design, \mathcal{MDLP} is well suited for combining knowledge from various sources, specially when some of these sources have priority over the others. More precisely, when rules from some sources are used to reject rules of other, less prior, sources. In particular, \mathcal{MDLP} is well suited for combining knowledge originating within hierarchically organized sources, as the following schematic example illustrates by combining knowledge coming from divers sectors of such an organization.

Example 9

Consider a company with a president, a board of directors and (at least) two departments: the quality management and the financial ones.

To improve the quality of the products produced by the company, the quality management department has decided not to buy any product whose reliability is less than guaranteed. In other words, it has to adopted the rule:

$$\text{not buy}(X) \leftarrow \text{not reliable}(X)$$

On the other hand, to save money, the financial department has decided to buy products of a type in need if they are cheap, i.e.

$$\text{buy}(X) \leftarrow \text{type}(X, T), \text{needed}(T), \text{cheap}(X)$$

The board of directors, in order to keep production going, has decided that whenever there is still a need for a type of product, exactly one product of that type must be bought. This can be coded by the following logic programming rules, stating that if X is a product of a needed type, and if the need for that type of product has not been already satisfied by buying some other product of that type, then X must be bought; if the need is satisfied by buying some other product of that type, then X should not be bought:

$$\begin{aligned} \text{buy}(X) &\leftarrow \text{type}(X, T), \text{needed}(T), \text{not satByOther}(T, X) \\ \text{not buy}(X) &\leftarrow \text{type}(X, T), \text{needed}(T), \text{satByOther}(T, X) \\ \text{satByOther}(T, X) &\leftarrow \text{type}(Y, T), \text{buy}(Y), X \neq Y \end{aligned}$$

Finally, the president decided for the company never to buy products that have a cheap alternative. I.e. if two products are of the same type, and one of them is cheap, the company should not buy the other one:

$$\text{not buy}(X) \leftarrow \text{type}(X, T), \text{type}(Y, T), X \neq Y, \text{cheap}(Y), \text{not cheap}(X)$$

For this example, suppose that there are two products, a and b , the first being cheap and the latter reliable, both of type t and both of needed type t .

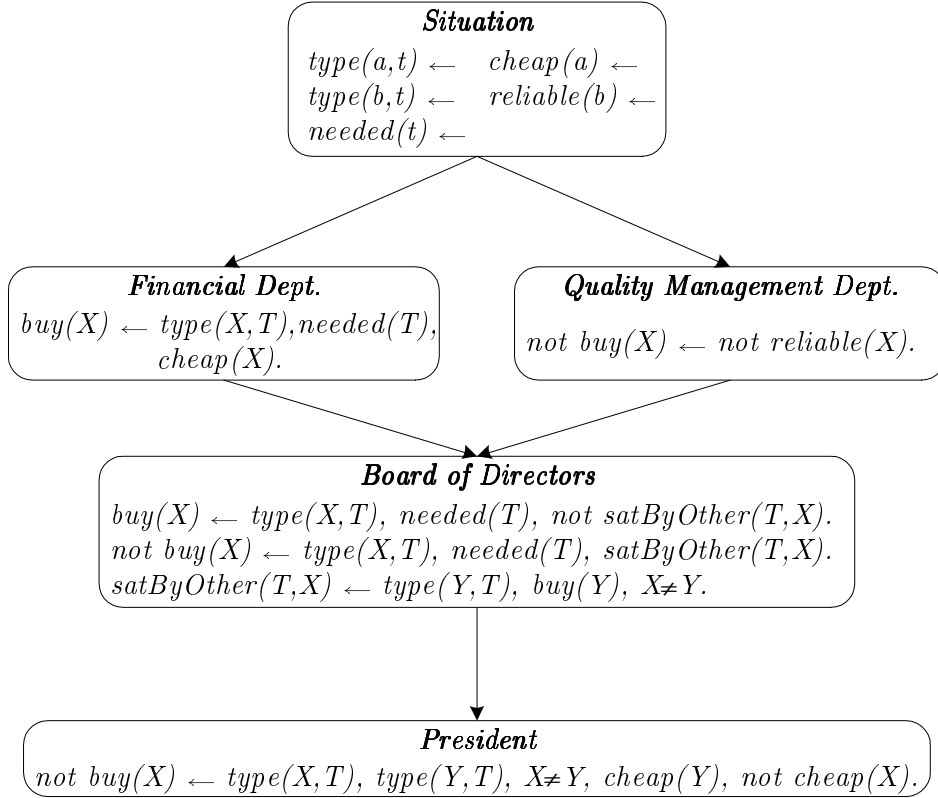


Fig. 7

According to the company's organizational chart, the rules of the president can overrule those of all other sectors, and those established by the board can overrule those decided by the departments. No department has precedence over any other.

This situation can easily be modeled by the MDLP depicted in figure 7.

To know what would be the decision of each of the sectors, about which products to buy, not taking under consideration the deliberation of its superiors, all needs to be done is to determine the stable models at the state corresponding to that sector. For example, the reader can check that at state *QMD* there is a single stable model in which both *not buy(a)* and *not buy(b)* are true. At the state *BD* there are two stable models: one in which *buy(a)* and *not buy(b)* are true; another where *not buy(a)* and *buy(b)* are true instead.

More interesting would be to know what is the decision of the company as a whole, when taking into account the rules of all sectors and their hierarchical organization. This is reflected by the stable models of the whole *MDLP*, i.e. the stable models at the set of all states of the *MDLP*. The reader can check that, in this instance, there is a single stable model in which *buy(a)* and *not buy(b)* are true. It coincides with the single stable model at state *president* because all other states belong to its relevancy graph.

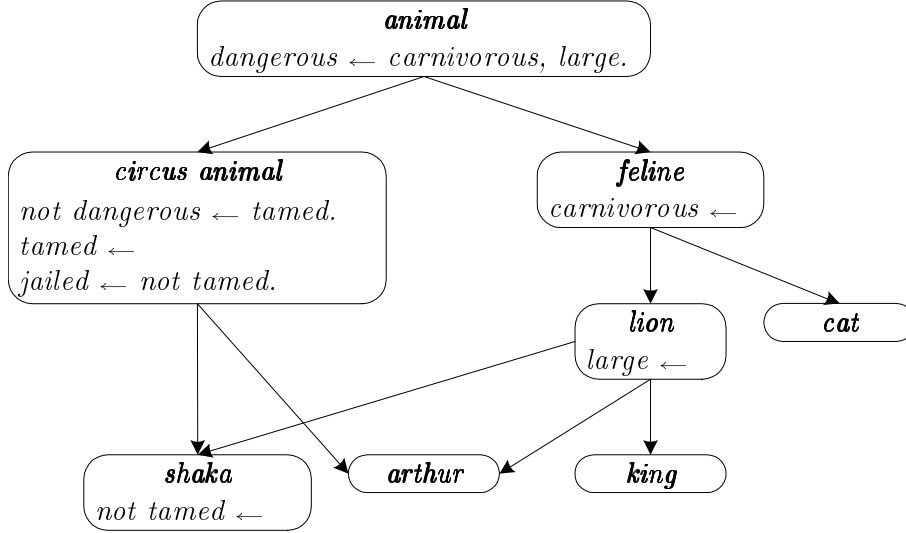


Fig. 8

6.2 Multiple Inheritance

Multi-dimensional Dynamic Logic Programming can be used to represent and reason about complex taxonomies, as exemplified in the following two examples:

Example 10

Consider the taxonomy with multiple inheritance, as per Fig. 8 (adapted from (David, 1994)). The reader can easily verify that the stable models at states *shaka*, *arthur* and *king* are, as expected:

$$\begin{aligned}
 M_{shaka} &= \{large, carnivorous, dangerous, jailed\} \\
 M_{arthur} &= \{large, carnivorous, tamed\} \\
 M_{king} &= \{large, carnivorous, dangerous\}
 \end{aligned}$$

The next example shows how the well known *royal-elephant* scenario (adapted from (Sandewall, 1986)) can be easily encoded in *MDLP*.

Example 11

The *royal-elephant* scenario is described by the sentences: Elephants are gray; African elephants are elephants; Royal elephants are elephants; Clyde is a royal elephant; Clyde is an african elephant; Royal elephants aren't gray.

This can be encoded by the *MDLP* with $\mathcal{P} = (\mathcal{P}_D, D)$ such that

$$\mathcal{P}_D = \{P_{elephant}, P_{african}, P_{royal}, P_{clyde}\}$$

where:

$$\begin{aligned}
 P_{elephant} &= \{gray \leftarrow\} & P_{african} &= \{\} \\
 P_{royal} &= \{not\ gray \leftarrow\} & P_{clyde} &= \{\}
 \end{aligned} \tag{91}$$

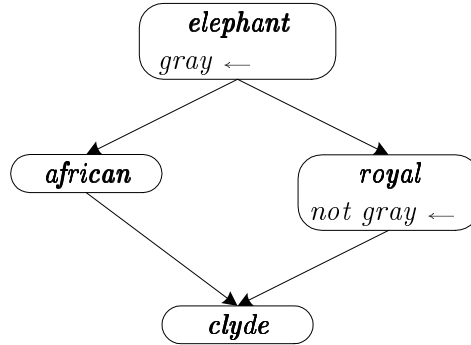


Fig. 9

and $D = (V, E)$ where

$$V = \{elephant, african, royal, clyde\} \quad (92)$$

$$E = \{(elephant, african), (elephant, royal), (african, clyde), (royal, clyde)\} \quad (93)$$

as depicted in Fig.9. The only stable model at state *clyde* is $M_{clyde} = \{not\ gray\}$. To confirm, we have that:

$$Reject(clyde, M_{clyde}) = \{gray \leftarrow\} \quad Default(\mathcal{P}_{clyde}, M_{clyde}) = \{not\ gray\} \quad (94)$$

and, finally,

$$[\mathcal{P}_{clyde} - Reject(s, M_{clyde})] \cup Default(\mathcal{P}_{clyde}, M_{clyde}) = \{not\ gray \leftarrow; not\ gray\} \quad (95)$$

whose least model is M_{clyde} . Note that at state *african* the only stable model is $M_{african} = \{gray\}$ because the rule *not gray ←* only rejects the rule *gray ←* at state *clyde*, that is, when both the rules *not gray ←* and *gray ←* are present in the relevancy graph.

6.3 Legal Reasoning

The next example describes how $MDLP$ can deal with collision principles, found in legal reasoning, such as *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 *Lex Posterior (Lex Posterior Derogat Legi Priori)* according to which the rule enacted at a later point in time overrides the earlier one, i.e how the combination of a temporal and an hierarchical dimensions can be combined into a single $MDLP$.

Example 12

In February 97, the President of Brazil (PB) passed a law determining that, in order to guarantee the safety aboard public transportation airplanes, all weapons were forbidden. Furthermore, all exceptional situations that, due to public interest,

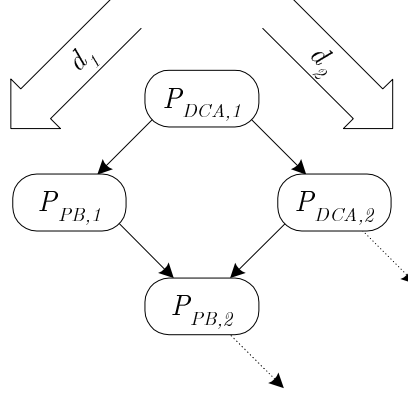


Fig. 10

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 (DCA) 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 generalized logic program clauses:

$$\begin{aligned} \text{rule1} &: \text{not carry_weapon} \leftarrow \text{not exception} \\ \text{rule2} &: \text{carry_weapon} \leftarrow \text{armed_officer} \end{aligned}$$

Let us consider a lattice with two distinct dimensions, corresponding to the two principles governing this situation: *Lex Superior* (d_1) and *Lex Posterior* (d_2). Besides the two agencies involved in this situation (PB and DCA), we will consider two time points representing the time when the two regulations were enacted. We have then a graph whose vertices are $\{(PB, 1), (PB, 2), (DCA, 1), (DCA, 2)\}$ (in the form (agency,time)) as portrayed in Fig.10. We have that $P_{DCA,1}$ contains rule 2, $P_{PB,2}$ 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 $P_{DCA,1}$. Applying Def.7, for $M_{PB,2} = \{\text{not carry_weapon}, \text{not exception}, \text{armed_officer}\}$ at state $(PB, 2)$ we have that:

$$\begin{aligned} \text{Reject}((PB, 2), M_{PB,2}) &= \{\text{carry_weapon} \leftarrow \text{armed_officer}\} \\ \text{Default}(\mathcal{P}_{PB,2}, M_{PB,2}) &= \{\text{not exception}\} \end{aligned}$$

it is trivial to see that

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

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 $\boxplus_{PB,2}\mathcal{P}$.

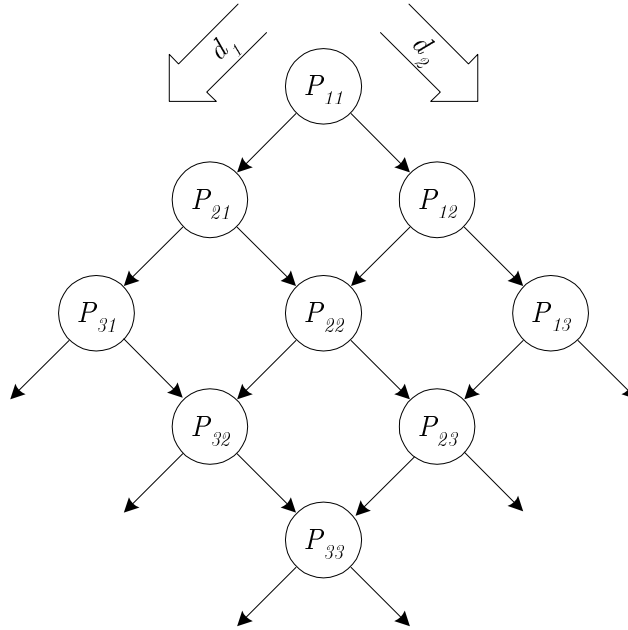


Fig. 11

6.4 MDLP and Multi-agent Systems

The previous example is a particular case suggesting that some particular acyclic digraphs are especially useful to model several aspects of multi-agent systems. We have in mind n -dimensional lattices, where each dimension represents one particular characteristic to be modeled. Suppose a linear hierarchically related society of agents situated in a dynamic environment. Fig.11 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 $\boxplus_{t,n} \mathcal{P}$.

The applicability of MDLP in a multi-agent context 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 a DAG with each agent, representing its own view of its relationships to other agents and of these amongst themselves. The stable models over a set of states from DAGs of different agents can provide us with interagent views.

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 stu-

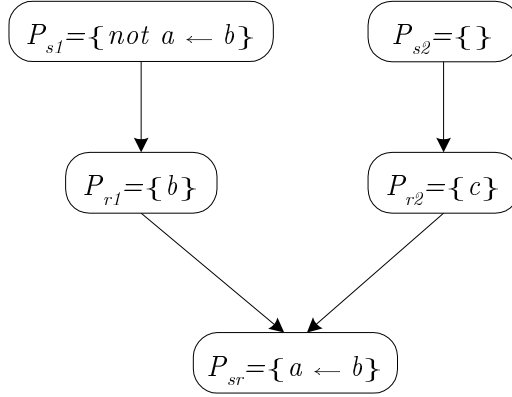


Fig. 12

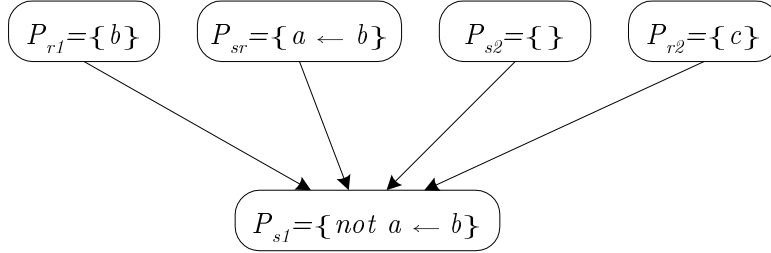


Fig. 13

dents (A_{s1} and A_{s2}) supervised by the two Researchers. The hierarchy is deployed in Fig.12, 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 community is quite different. Fig.13 manifests one possible view by A_{s1} . In this scenario, we could use \mathcal{MDLP} to determine and eventually compare A_{sr} 's view, given by $\boxplus_{sr}\mathcal{P}$ in Fig.12, with A_{s1} 's view, given by $\boxplus_{s1}\mathcal{P}$ in Fig.13. If we assign the following simple logic programs to the five agents:

$$\begin{array}{ll} P_{sr} = \{a \leftarrow b\} & P_{r1} = \{b\} \\ P_{s1} = \{not\ a \leftarrow c\} & P_{r2} = \{c\} \\ P_{s2} = \{\} & \end{array}$$

we have that $\boxplus_{sr}\mathcal{P}$ in Fig.12 has $M_{sr} = \{a, b, c\}$ as the only stable model, and $\boxplus_{s1}\mathcal{P}$ in Fig.13 has $M_{s1} = \{not\ a, b, c\}$ as its only stable model. That is, according to 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.

This example suggests \mathcal{MDLP} to be a useful practical framework to study 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. \mathcal{MDLP} offers an important basic tool in the formal study of the social behaviour in multi-

agent communities. In the final section, we elaborate on extensions to $MDLP$ to enhance this tool.

7 Conclusions and Future Work

We have shown how $MDLP$ generalizes DLP in allowing for collections of states organized by arbitrary acyclic digraphs, and not just sequences of states. Indeed, $MDLP$ assigns semantics to sets and subsets of logic programs, on the basis of how they stand in relation amongst each other, as defined by an acyclic digraph. Such a natural generalization imparts added expressiveness to updating, thereby amplifying the coverage of its application domains. The flexibility provided by a DAG accrues to the scope and variety of the new possibilities. The new characteristics of multiplicity and composition of $MDLP$ impart an innovative “societal” viewpoint to *Logic Programming*.

Much remains to be done. Some of the more immediate themes of ongoing work regarding the further development of multi-dimensional updates comprise:

- Allowing for the DAG to evolve, by updating the DAG itself with new nodes and edges.
- Enhancing the *LUPS* language to adumbrate update commands over DAGs.
- Studying the conditions for and uses of dropping the acyclicity condition.
- Discovering the garbage collection principles that: (1) Permit the forgetting of rules made irrelevant, with regard to the intend query states, by subsequent updates. (2) Permit program transformations to the dynamic program update that make it more compact and efficient while preserving the semantics. (3) Take increased advantage of tabling mechanisms.
- Capturing with $MDLP$ assorted program composition operators (Brogi *et al.*, 1999).
- Establishing a paraconsistent $MDLP$ semantics and defining contradiction removal over DAGs.
- Coupling deliberative goal oriented abduction of updates with reactive update actions.
- Introducing communication primitives among nodes in the form of message updates.

Inevitably, updating raises other issues, such as about revising and preferring, and work is emerging on the articulation of these distinct but highly complementary aspects (Alferes & Pereira, 2000b). Learning is usefully seen as successive approximate change, as opposed to exact change, and combining the results of learning by multiple agents, multiple strategies, or multiple data sets, inevitably poses problems within the province of updating, cf. (Lamma *et al.*, 2000). Goal directed planning has been fruitfully construed as abductive updating in (Alferes *et al.*, 2000b). Mutually updating and communicating agents have been studied in (Dell’Acqua & Pereira, 1999). For a survey on updating postulates, structural properties, and complexity check (Eiter *et al.*, 2000). A well-founded semantics for

generalized programs has also been defined (Alferes *et al.*, 1999b), which allows carrying over results to 3-valued updates.⁵

Moreover, the issue is not just that of conceptual integration but that of conceptual cross-fertilization too. How can one employ learning in the service of updating? How does one remove contradictions from updates? How can rule preferences (Brewka & Eiter, 1999) be combined with the type of preferences enacted by updates, and the preferences themselves be subjected to updating?

Indeed, how can other logic programming based computational reasoning abilities such as assuming by default, abducing, revising beliefs, etc., be integrated into a multi-dimensional updating framework involving a multiplicity of agents?

Not only do the aforementioned topics combine naturally together – and so require precise, formal, means and tools to do so –, but their combination results in turn in a nascent complex architectural basis and component for Logic Programming rational agents, which can update one another and common, structured, updatable blackboards. It can be surmised, consequently, that the fostering of this meshing of topics within the *Logic Programming* community is all of opportune, seeding, and fruitful. In fact, application areas such as software development, multi-strategy learning, abductive planning, model-based diagnosis, agent architecture, and others, are already being successfully pursued while employing precisely this outlook.

The use of logic programming for the overall endeavour is justified on the basis 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.

Were it not for the impressive development in *Logic Programming* semantics in the past 12 years or so, and its extensions to non-monotonic and other forms of reasoning, we would not have today the impressive, sound, and efficient system implementations of such semantics (be it the stable or well-founded varieties or their hybrid combination), which have been opening up a whole new gamut of application areas as well as a spate of sophisticated reasoning abilities over them. This was made possible only through successive and prolonged efforts at theoretical generalization and synthesis, and by way of the combined integration of theory, procedure development, and practical implementation.

The ongoing work on multi-dimensional updating reported here is just another outcome of this joint enterprise.

⁵ This semantics is actually the basis for our implementation under the XBS system (XSB-Prolog, 1999). The other implementation, relies on a preprocessor that produces programs to be run under the DLV-system (DLV, 2000)

Acknowledgments

We would like to thank Thomas Eiter, Domenico Saccà and Teodor Przymusinski for valuable discussions. We would also like to thank Pierangelo Dell'Acqua for his comments on a draft of this paper.

References

- Alferes, J. J., & Pereira, L. M. (2000a). *Logic programming updating - a guided tour*. Submitted.
- Alferes, J. J., & Pereira, L. M. (2000b). Updates plus preferences. Aciego, M. O., Guzmán, I. P., Brewka, G., & Pereira, L. M. (eds), *Proceedings of the european workshop on logics in artificial intelligence (JELIA-00)*. LNAI, vol. 1919. Springer.
- Alferes, J. J., Pereira, L. M., & Przymusinski, T. (1996). Strong and explicit negation in non-monotonic reasoning and logic programs. *Pages 143–163 of: Alferes, J. J., Pereira, L. M., & Orłowska, E. (eds), Proceedings of the european workshop on logics in artificial intelligence (JELIA-96)*. LNAI, vol. 1126. Springer.
- Alferes, J. J., Pereira, L. M., & Przymusinski, T. (1998a). 'Classical' negation in non-monotonic reasoning and logic programming. *Journal of automated reasoning*, **20**(1 & 2), 107–142.
- Alferes, J. J., Leite, J. A., Pereira, L. M., Przymusinska, H., & Przymusinski, T. C. (1998b). Dynamic logic programming. *Pages 98–111 of: Cohn, A., Schubert, L., & Shapiro, S. (eds), Proceedings of the 6th international conference on principles of knowledge representation and reasoning (KR-98)*. San Francisco: Morgan Kaufmann Publishers.
- Alferes, J. J., Pereira, L. M., Przymusinska, H., & Przymusinski, T. (1999a). LUPS : A language for updating logic programs. *Pages 162–176 of: Gelfond, Michael, Leone, Nicola, & Pfeifer, Gerald (eds), Proceedings of the 5th international conference on logic programming and nonmonotonic reasoning (LPNMR-99)*. LNAI, vol. 1730. Berlin: Springer.
- Alferes, J. J., Pereira, L. M., Przymusinski, T., Przymusinska, H., & Quaresma, P. (1999b). Preliminary exploration on actions as updates. Meo, M. C., & Ferro, M. V. (eds), *Proceedings of the 1999 joint conference on declarative programming (AGP-99)*.
- Alferes, J. J., Leite, J. A., Pereira, L. M., Przymusinska, H., & Przymusinski, T. C. (2000a). Dynamic updates of non-monotonic knowledge bases. *The journal of logic programming*, **45**(1–3), 43–70.
- Alferes, J. J., Leite, J. A., Pereira, L. M., & Quaresma, P. (2000b). Planning as abductive updating. *Pages 1–8 of: Kitchin, D. (ed), Proceedings of the AISB'00 symposium on AI planning and intelligent agents*. AISB.
- Brewka, G., & Eiter, T. (1999). Preferred answer sets for extended logic programs. *Artificial intelligence*, **109**. A short version appeared in A. Cohn and L. Schubert (eds.), *KR'98*, Morgan Kaufmann.
- Brogi, A., Contiero, S., & Turini, F. (1999). Programming by combining general logic programs. *Journal of logic and computation*, **9**(1), 7–24.
- Buccafurri, F., Faber, W., & Leone, N. (1999). Disjunctive logic programs with inheritance. *Pages 79–93 of: Schreye, D. De (ed), Proceedings of the 1999 international conference on logic programming (ICLP-99)*. Cambridge: MIT Press.
- David, G. T. (1994). *Semantics of multiple inheritance with exceptions in hierarchically structured logic theories*. Ph.D. thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

- Dell'Acqua, P., & Pereira, L. M. (1999). Updating agents. Rochefort, S., Sadri, F., & Toni, F. (eds), *ICLP-99 workshop on multi-agent systems in logic*.
- DLV. (2000). *The DLV project - a disjunctive datalog system (and more)*. Available at <http://www.dbai.tuwien.ac.at/proj/dlv/>.
- Eiter, Thomas, Fink, Michael, Sabbatini, Giuliana, & Tompits, Hans. (2000). Considerations on updates of logic programs. *Pages 2–20 of: Ojeda-Aciego, Manuel, de Guzmán, Inma P., Brewka, Gerhard, & Pereira, Luís Moniz (eds), Proceedings of the european workshop on logics in artificial intelligence (JELIA-00)*. LNAI, vol. 1919. Springer.
- Gelfond, M., & Lifschitz, V. (1988). The stable semantics for logic programs. *Pages 1070–1080 of: Kowalski, R., & Bowen, K. (eds), Proceedings of the 5th international symposium on logic programming*. Cambridge, MA.: MIT Press.
- Gelfond, M., & Lifschitz, V. (1990). Logic Program with Classical Negation. *Pages 579–597 of: Warren, David H. D., & Szeredi, Peter (eds), Proceedings of the 7th int. conf. on logic programming*. MIT.
- Inoue, K., & Sakama, C. (1998). Negation as failure in the head. *Journal of logic programming*, **35**, 39–78.
- Katsuno, Hirojumi, & Mendelzon, Alberto O. (1991). On the difference between updating a knowledge base and revising it. *Pages 387–394 of: Allen, James, Fikes, Richard, & Sandewall, Erik (eds), Proceedings of the 2nd international conference on principles of knowledge representation and reasoning*. San Mateo, CA, USA: Morgan Kaufmann Publishers.
- Kowalski, R. (1992). Legislation as logic programs. *Pages 203–230 of: Logic programming in action*. Springer-Verlag.
- Lamma, E., Riguzzi, F., & Pereira, L. M. (2000). Strategies in combined learning via logic programs. *Machine learning*, **38**(1/2), 63–87.
- Leite, J. A. 1997 (November). *Logic program updates*. M.Phil. thesis, Dept. de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.
- Leite, J. A., & Pereira, L. M. (1997). Generalizing updates: From models to programs. *Pages 224–246 of: Dix, J., Pereira, L. M., & Przymusiński, T. C. (eds), Selected extended papers of the ILPS'97 3th international workshop on logic programming and knowledge representation (LPKR-97)*. LNAI, vol. 1471. Berlin: Springer Verlag.
- Leite, J. A., & Pereira, L. M. (1998). Iterated logic program updates. *Pages 265–278 of: Jaffar, J. (ed), Proceedings of the 1998 joint international conference and symposium on logic programming (JICSLP-98)*. Cambridge: MIT Press.
- Leite, J. A., Pereira, F. C., Cardoso, A., & Pereira, L. M. (2000). Metaphorical mapping consistency via dynamic logic programming. *Pages 41–50 of: Wiggins, G. (ed), Proceedings of the AISB'00 symposium on creative and cultural aspects and applications of AI and cognitive science*. AISB.
- Lifschitz, V., & Woo, T. (1992). Answer sets in general non-monotonic reasoning (preliminary report). Nebel, B., Rich, C., & Swartout, W. (eds), *Proceedings of the 3th international conference on principles of knowledge representation and reasoning (KR-92)*. Morgan-Kaufmann.
- Marek, V. W., & Truszczyński, M. (1994). Revision specifications by means of programs. *Pages 122–136 of: MacNish, C., Pearce, D., & Pereira, L. M. (eds), Proceedings of the european workshop on logics in artificial intelligence (JELIA-94)*. LNAI, vol. 838. Berlin: Springer.
- Newtono, Isaaco. (1726). *Philosophiæ naturalis principia mathematica*. Editio tertia & aucta emendata. Apud Guil & Joh. Innys, Regiæ Societatis typographos. Original quotation: “*Corpus omne perseverare in statu suo quiescendi vel movendi uniformiter in directum, nisi quatenus illud a viribus impressis cogitur statum suum mutare.*”.

- Niemelä, I., & Simons, P. (1997). Smodels: An implementation of the stable model and well-founded semantics for normal LP. *Pages 420–429 of: Dix, J., Furbach, U., & Nerode, A. (eds), Proceedings of the 4th international conference on logic programming and nonmonotonic reasoning (LPNMR-97)*. LNAI, vol. 1265. Berlin: Springer.
- Przymusiński, Teodor C., & Turner, Hudson. (1997). Update by means of inference rules. *Journal of logic programming*, **30**(2), 125–143.
- Quaresma, P., & Pereira, L. M. (1998). Modelling agent interaction in logic programming. Barenstein, O. (ed), *Procs. of the INAP-98*.
- Quaresma, P., & Rodrigues, I. P. (1999). A collaborative legal information retrieval system using dynamic logic programming. *Pages 190–191 of: Proceedings of the seventh international conference on artificial intelligence and law (ICAIL-99)*. ACM SIGART. N.Y.: ACM Press.
- Sakama, C., & Inoue, K. (1999). Updating extended logic programs through abduction. Gelfond, M., Leone, N., & Pfeifer, G. (eds), *Proceedings of the 5th international conference on logic programming and nonmonotonic reasoning (LPNMR-99)*. Springer.
- Sandewall, Erik. (1986). Nonmonotonic inference rules for multiple inheritance with exceptions. *IEEE*, **74**(10), 1345–1353.
- Winslett, Marianne. (1988). Reasoning about action using a possible models approach. *Pages 429–450 of: Smith, Reid, & Mitchell, Tom (eds), Proceedings of the seventh national conference on artificial intelligence*. Menlo Park, California: AAAI Press, for American Association for Artificial Intelligence.
- XSB-Prolog. (1999). *The XSB logic programming system, version 2.0*. Available at <http://www.cs.sunysb.edu/sbprolog>.

A Proof of Theorem 3

For the proof of the theorem it is sufficient to prove that for every interpretation M_S the following holds (where in all three equalities the left hand side is according to Def. 8 and the right hand side according to Def. 7:

1. $\mathcal{P}_S = \mathcal{P}'_\alpha$
2. $Reject(S, M_S) = Reject(\alpha, M_S)$
3. $Default(\mathcal{P}_S, M_S) = Default(\mathcal{P}'_\alpha, M_S)$

1. \mathcal{P}_S contains all rules belonging to all programs indexed by a vertex of the relevancy DAG wrt S , D_S . From the construction of D_α it follows that the relevancy DAG of D_α wrt α contains precisely the same set of vertices as D_S , plus vertex α . Since the program indexed by α , P_α , is empty, it follows that \mathcal{P}'_α contains the same rules as \mathcal{P}_S .
2. Let us start with

$$Reject(\alpha, M_S) = \left\{ \begin{array}{l} r \in P_i \mid \exists r' \in P_j, i < j \leq \alpha, \\ head(r) = not\ head(r') \wedge M_S \models body(r') \end{array} \right\} \quad (A1)$$

Since for all $v \in V, s \in S$ such that $v \leq s$ we have that $\alpha > v$, we can rewrite

the set $Reject(\alpha, M_S)$ as:

$$Reject(\alpha, M_S) = \left\{ \begin{array}{l} r \in P_i \mid \exists s \in S, \exists r' \in P_j, i < j \leq s, \\ head(r) = not\ head(r') \wedge M_s \models body(r') \end{array} \right\} \cup \quad (\text{A } 2)$$

$$\cup \left\{ \begin{array}{l} r \in P_i \mid \exists r' \in P_\alpha, i < \alpha, \\ head(r) = not\ head(r') \wedge M_s \models body(r') \end{array} \right\} \quad (\text{A } 3)$$

Since $\nexists r' \in P_\alpha$, the set $Reject(\alpha, M_S)$ can be reduced to

$$Reject(\alpha, M_S) = \left\{ \begin{array}{l} r \in P_i \mid \exists s \in S, \exists r' \in P_j, i < j \leq s, \\ head(r) = not\ head(r') \wedge M_s \models body(r') \end{array} \right\} \quad (\text{A } 4)$$

i.e. $Reject(\alpha, M_S) = Reject(S, M_S)$.

3. According to Def.8,

$$Default(\mathcal{P}_S, M_S) = \{not\ A \mid \nexists r \in \mathcal{P}_S : (head(r) = A) \wedge M_S \models body(r)\} \quad (\text{A } 5)$$

Since $\mathcal{P}_S = \mathcal{P}'_\alpha$ then

$$\begin{aligned} Default(\mathcal{P}_S, M_S) &= \{not\ A \mid \nexists r \in \mathcal{P}'_\alpha : (head(r) = A) \wedge M_S \models body(r)\} = \\ &= Default(\mathcal{P}'_\alpha, M_S) \end{aligned}$$

B Proof of Theorem 7

According to Proposition 1 and that rules belonging to those programs indexed by the vertices that do not belong to the relevancy graph play no role in $\boxplus_s \mathcal{P}$, according to Def. 10, we can safely remove such rules from \mathcal{P} and such vertices and corresponding edges from D , thus obtaining $\mathcal{P}_s = (\mathcal{P}_{D_s}, D_s)$ where $D_s = (V_s, E_s)$ is the relevancy DAG of D wrt s . and $\mathcal{P}_{D_s} = \{P_v : v \in V_s\}$. Therefore, we need only prove the theorem for \mathcal{P}_s or, alternatively, consider $D = D_s$ and, consequently, $\mathcal{P} = \mathcal{P}_s$.

Let r^+ be a rule of the form:

$$A \leftarrow B_1, \dots, B_m, not\ C_1, \dots, not\ C_n \quad (\text{B } 1)$$

and let r^- be a rule of the form:

$$not\ A \leftarrow B_1, \dots, B_m, not\ C_1, \dots, not\ C_n \quad (\text{B } 2)$$

So $\boxplus_s \mathcal{P}$ contains the rules:

$$A_{P_v} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_v \quad (\text{B } 3)$$

$$A_{P_v}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_v \quad (\text{B } 4)$$

$$A_v \leftarrow A_u, \text{not reject}(A_u) \quad \forall (u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B } 5)$$

$$A_v^- \leftarrow A_u^-, \text{not reject}(A_u^-) \quad \forall (u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B } 6)$$

$$\text{reject}(A_u^-) \leftarrow A_{P_v} \quad \forall u, v \in V : u < v, \forall A \in \mathcal{K} \quad (\text{B } 7)$$

$$\text{reject}(A_u) \leftarrow A_{P_v}^- \quad \forall u, v \in V : u < v, \forall A \in \mathcal{K} \quad (\text{B } 8)$$

$$A_v \leftarrow A_{P_v} \quad \forall v \in V, \forall A \in \mathcal{K} \quad (\text{B } 9)$$

$$A_v^- \leftarrow A_{P_v}^- \quad \forall v \in V, \forall A \in \mathcal{K} \quad (\text{B } 10)$$

$$A_{s_0}^- \quad \forall A \in \mathcal{K} \quad (\text{B } 11)$$

$$A \leftarrow A_s \quad \forall A \in \mathcal{K} \quad (\text{B } 12)$$

$$A^- \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B } 13)$$

$$\text{not } A \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B } 14)$$

(\Rightarrow) Suppose that N is a stable model of the program $\boxplus_s \mathcal{P}$ and let $R = \boxplus_s \mathcal{P} \cup N^-$. From Def.1 it follows that:

$$N = \text{least}(R) = \text{least}(\boxplus_s \mathcal{P} \cup N^-) \quad (\text{B } 15)$$

Let

$$T = [\mathcal{P}_s - \text{Reject}(s, M)] \cup \text{Default}(\mathcal{P}_s, M) \quad (\text{B } 16)$$

and let

$$H = \text{least}(T) \quad (\text{B } 17)$$

be its least model (in the language \mathcal{L}). We will show that the restriction $M = N \upharpoonright \mathcal{L}$ of N to the language \mathcal{L} coincides with H .

Denote by \mathcal{S} the sub-language of $\widehat{\mathcal{L}}$ that includes only propositional symbols $\{A : A \in \mathcal{K}\} \cup \{A^- : A \in \mathcal{K}\}$. By means of several simple reductions we will transform the program $R = \boxplus_s \mathcal{P} \cup N^-$ in the language $\widehat{\mathcal{L}}$ into a simpler program Q in the language \mathcal{S} so that:

- The least model $J = \text{least}(Q)$ of Q is equal to the least model $N = \text{least}(R)$ of R when restricted to the language \mathcal{S} , i.e., $J = N \upharpoonright \mathcal{S}$;
- The program Q in the language \mathcal{S} is syntactically identical to the program $T = [\mathcal{P}_s - \text{Reject}(s, M)] \cup \text{Default}(\mathcal{P}_s, M)$ in the language \mathcal{L} , except that $\text{not } A$ is everywhere replaced by A^- .

First of all, according to Def.1, we observe that $\text{not reject}(A_u)$ belongs to N^- , iff $N \not\models A_{P_v}^-, \forall u, v \in V : u < v$, since there are no rules for $\text{not reject}(A_u)$ (remember that $\text{not } A$ is treated as a propositional symbol). Similarly, $\text{not reject}(A_u^-)$ belongs to N^- , iff $N \not\models A_{P_v}, \forall u, v \in V : u < v$. We can thus replace rules (B 5) and (B 6)

with:

$$A_v \leftarrow A_u, \text{not } A_{P_w} \quad \forall (u, v) \in E, w \in V, u < w \quad (\text{B 18})$$

$$A_v^- \leftarrow A_u^-, \text{not } A_{P_w}^- \quad \forall (u, v) \in E, w \in V, u < w \quad (\text{B 19})$$

Since we are not interested in the truth value of the atoms of the form $\text{reject}(_)$ (they do not belong to \mathcal{S}) and they no longer appear in the body of any rules, we can eliminate rules (B 7) and (B 8). The transformed program, R' , now looks like:

$$A_{P_v} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_v \quad (\text{B 20})$$

$$A_{P_v}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_v \quad (\text{B 21})$$

$$A_v \leftarrow A_u, \text{not } A_{P_w} \quad \forall (u, v) \in E, w \in V, u < w, \forall A \in \mathcal{K} \quad (\text{B 22})$$

$$A_v^- \leftarrow A_u^-, \text{not } A_{P_w}^- \quad \forall (u, v) \in E, w \in V, u < w, \forall A \in \mathcal{K} \quad (\text{B 23})$$

$$A_v \leftarrow A_{P_v} \quad \forall v \in V, \forall A \in \mathcal{K} \quad (\text{B 24})$$

$$A_v^- \leftarrow A_{P_v}^- \quad \forall v \in V, \forall A \in \mathcal{K} \quad (\text{B 25})$$

$$A_{s_0}^- \quad \forall A \in \mathcal{K} \quad (\text{B 26})$$

$$A \leftarrow A_s \quad \forall A \in \mathcal{K} \quad (\text{B 27})$$

$$A^- \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B 28})$$

$$\text{not } A \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B 29})$$

$$\text{not } A \quad \forall A \in \overline{\mathcal{K}}, \text{not } A \in N^- \quad (\text{B 30})$$

If we partially evaluate rules (B 24) and (B 25), using rules (B 20) and (B 21), we obtain

$$A_v \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_v \quad (\text{B 31})$$

$$A_v^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_v \quad (\text{B 32})$$

We can now simplify rules (B 22) and (B 23) by replacing them with the simpler rules

$$A_v \leftarrow A_u \quad \forall (u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B 33})$$

$$A_v^- \leftarrow A_u^- \quad \forall (u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B 34})$$

if we also remove from this new program the rules (B 31) such that

$$\exists (u, v) \in E, \exists w > v, \exists A_{P_v}^- \leftarrow \text{Body} \in R' : N \models \text{Body} \quad (\text{B 35})$$

and the rules (B 32) such that

$$\exists (u, v) \in E, \exists w > v, \exists A_{P_v} \leftarrow \text{Body} \in R' : N \models \text{Body} \quad (\text{B 36})$$

This is the same as saying that one should remove rules (B 31) and (B 32) such that the original corresponding rules belong to $\text{Reject}(s, M)$. We also have to deal with the rules (B 26), which play a no different role than rules (B 31) and (B 32), but with respect to the initial state. We should then eliminate all those such that

$$\exists (s_0, v) \in E, \exists w > v, \exists A_{P_v} \leftarrow \text{Body} \in R' : N \models \text{Body} \quad (\text{B 37})$$

which is the same as saying that we should preserve only those rules $A_{s_0}^-$ if *not* $A \in \text{Default}(\mathcal{P}_s, M)$.

In this proof, we can eliminate rules (B20) and (B21) since atoms of the form $A_{P_v}^-$ and A_{P_v} do not belong to \mathcal{S} and they no longer appear in the body of any clauses. The transformed program, R'' , now looks like:

$$A_v \leftarrow A_u \quad \forall(u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B } 38)$$

$$A_v^- \leftarrow A_u^- \quad \forall(u, v) \in E, \forall A \in \mathcal{K} \quad (\text{B } 39)$$

$$A_v \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_v, r^+ \notin \text{Reject}(s, M) \quad (\text{B } 40)$$

$$A_v^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_v, r^- \notin \text{Reject}(s, M) \quad (\text{B } 41)$$

$$A_{s_0}^- \quad \forall \text{not } A \in \text{Default}(\mathcal{P}_s, M). \quad (\text{B } 42)$$

$$A \leftarrow A_s \quad \forall A \in \mathcal{K} \quad (\text{B } 43)$$

$$A^- \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B } 44)$$

$$\text{not } A \leftarrow A_s^- \quad \forall A \in \mathcal{K} \quad (\text{B } 45)$$

$$\text{not } A \quad \forall A \in \overline{\mathcal{K}}, \text{not } A \in N^- \quad (\text{B } 46)$$

We can now remove all the negative facts in N^- and the default rule $\text{not } A \leftarrow A_s^-$ from R'' because they only involve propositional symbols *not* A which no longer appear in bodies of any other clauses from R'' and thus do not affect the least model of R'' restricted to the language \mathcal{S} . Also, because there are edges $\forall(u, v) \in E$ from all states to state s (E is the relevancy graph of itself wrt s), we can perform several partial evaluations and remove the rules for A_v and A_v^- until we are left only with rules for A and A^- . The result is the final program Q :

$$A \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in \mathcal{P}_s, r^+ \notin \text{Reject}(s, M) \quad (\text{B } 47)$$

$$A^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in \mathcal{P}_s, r^- \notin \text{Reject}(s, M) \quad (\text{B } 48)$$

$$A^- \quad \forall \text{not } A \in \text{Default}(\mathcal{P}_s, M). \quad (\text{B } 49)$$

Clearly, this program is entirely identical to the program $T = [\mathcal{P}_s - \text{Reject}(s, M)] \cup \text{Default}(\mathcal{P}_s, M)$, except that *not* A is everywhere replaced by A^- . Consequently, the least model J of Q is identical to the least model H of T , except that *not* A is everywhere replaced by A^- . Moreover, due to the way in which it was obtained, the least model $J = \text{least}(Q)$ of the program Q is equal to the least model $N = \text{least}(R)$ of R restricted to the language \mathcal{S} , i.e., $J = N \mid \mathcal{S}$. This entails that for any $A \in \mathcal{K}$:

$$A \in N \quad \text{iff} \quad A \in J \quad \text{iff} \quad A \in H \quad (\text{B } 50)$$

$$A^- \in N \quad \text{iff} \quad A^- \in J \quad \text{iff} \quad \text{not } A \in H. \quad (\text{B } 51)$$

We conclude that $M = N \mid \mathcal{L} = H$, because $\text{not } A \in N \quad \text{iff} \quad A^- \in N$. This completes the proof in one direction. The converse implication is established in an analogous way.

C Proof of Theorem 8

Since the stable models of the multi-dimensional update at state s coincide with the stable models of the multi-dimensional dynamic logic program $\boxplus_s \mathcal{P}$, all we have to show is that:

- for every stable model N_1 of $\boxplus_s \mathcal{P}_D$, there exists a stable model N_2 of $\boxplus_s \mathcal{P}$ such that $N_1 \upharpoonright \mathcal{L} = N_2 \upharpoonright \mathcal{L}$;
- for every stable model N_2 of $\boxplus_s \mathcal{P}$, there exists a stable model N_1 of $\boxplus_s \mathcal{P}_D$ such that $N_2 \upharpoonright \mathcal{L} = N_1 \upharpoonright \mathcal{L}$;

Since, as we've seen before, the programs indexed by states greater than s , in a dynamic logic program, do not affect the semantics at state s , and the programs indexed by states that do not belong to the relevancy graph wrt s , in a multi-dimensional dynamic logic program, do not affect the semantics at state s , we only need to prove the equivalence for $s = n$.

Let r^+ be a rule of the form:

$$A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (\text{C1})$$

and let r^- be a rule of the form:

$$\text{not } A \leftarrow B_1, \dots, B_m, \text{ not } C_1, \dots, \text{ not } C_n \quad (\text{C2})$$

The dynamic program update $\boxplus_n \mathcal{P}_D$ at state n contains the following rules:

$$A_{P_i} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_i \quad (\text{C3})$$

$$A_{P_i}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_i \quad (\text{C4})$$

$$A_i \leftarrow A_{i-1}, \text{ not } A_{P_i}^- \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C5})$$

$$A_i^- \leftarrow A_{i-1}^-, \text{ not } A_{P_i} \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C6})$$

$$A_i \leftarrow A_{P_i} \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C7})$$

$$A_i^- \leftarrow A_{P_i}^- \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C8})$$

$$A_0^- \quad \forall A \in \mathcal{K} \quad (\text{C9})$$

$$A \leftarrow A_n \quad \forall A \in \mathcal{K} \quad (\text{C10})$$

$$A^- \leftarrow A_n^- \quad \forall A \in \mathcal{K} \quad (\text{C11})$$

$$\text{not } A \leftarrow A_n^- \quad \forall A \in \mathcal{K} \quad (\text{C12})$$

We can transform this program, by introducing a predicate *reject*, with any $A \neq \text{reject}$, without changing its models (in what concerns all other atoms), and replace rules (C5) and (C6) with the rules:

$$A_i \leftarrow A_{i-1}, \text{ not } \text{reject}(A_{i-1}) \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C13})$$

$$A_i^- \leftarrow A_{i-1}^-, \text{ not } \text{reject}(A_{i-1}^-) \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C14})$$

$$\text{reject}(A_{i-1}^-) \leftarrow A_{P_i} \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C15})$$

$$\text{reject}(A_{i-1}) \leftarrow A_{P_i}^- \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C16})$$

The multi-dimensional dynamic logic program $\boxplus_n \mathcal{P}$ at state n contains the following rules:

$$A_{P_v} \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^+ \in P_i \quad (\text{C 17})$$

$$A_{P_v}^- \leftarrow B_1, \dots, B_m, C_1^-, \dots, C_n^- \quad \forall r^- \in P_i \quad (\text{C 18})$$

$$A_i \leftarrow A_{i-1}, \text{not reject}(A_{i-1}) \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C 19})$$

$$A_i^- \leftarrow A_{i-1}^-, \text{not reject}(A_{i-1}^-) \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C 20})$$

$$\text{reject}(A_{i-1}^-) \leftarrow A_{P_j} \quad \forall A \in \mathcal{K}, \forall i, j : 1 \leq i < j \leq n \quad (\text{C 21})$$

$$\text{reject}(A_{i-1}) \leftarrow A_{P_j}^- \quad \forall A \in \mathcal{K}, \forall i, j : 1 \leq i < j \leq n \quad (\text{C 22})$$

$$A_i \leftarrow A_{P_i} \quad \forall A \in \mathcal{K} \quad (\text{C 23})$$

$$A_i^- \leftarrow A_{P_i}^- \quad \forall A \in \mathcal{K} \quad (\text{C 24})$$

$$A_0^- \quad \forall A \in \mathcal{K} \quad (\text{C 25})$$

$$A \leftarrow A_n \quad \forall A \in \mathcal{K} \quad (\text{C 26})$$

$$A^- \leftarrow A_n^- \quad \forall A \in \mathcal{K} \quad (\text{C 27})$$

$$\text{not } A \leftarrow A_n^- \quad \forall A \in \mathcal{K} \quad (\text{C 28})$$

In this program, rules (C 21) and (C 22) can be split into the following ones:

$$\text{reject}(A_{i-1}^-) \leftarrow A_{P_i} \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C 29})$$

$$\text{reject}(A_{i-1}) \leftarrow A_{P_i}^- \quad \forall A \in \mathcal{K}, 1 \leq i \leq n \quad (\text{C 30})$$

$$\text{reject}(A_{i-1}^-) \leftarrow A_{P_{i+1}} \quad \dots \quad \text{reject}(A_{i-1}^-) \leftarrow A_{P_n} \quad \forall A \in \mathcal{K}, 1 \leq i < n - 1 \quad (\text{C 31})$$

$$\text{reject}(A_{i-1}) \leftarrow A_{P_{i+1}}^- \quad \dots \quad \text{reject}(A_{i-1}) \leftarrow A_{P_n}^- \quad \forall A \in \mathcal{K}, 1 \leq i < n - 1 \quad (\text{C 32})$$

Now we can see that the only difference between the modified (with the introduction of the *reject* predicate) dynamic program update $\oplus_n \mathcal{P}_D$ and the multi-dimensional dynamic logic program $\boxplus_n \mathcal{P}$ at state n is that the latter contains the extra sets of rules (C 31) and (C 32). We now have to show that these rules do not affect the models, when restricted to \mathcal{L} . In what concerns these rules, one of the following cases could occur:

- none of the A_{P_j} and (resp. $A_{P_j}^-$) is true in the model, in which case both programs are trivially equivalent;
- one of the A_{P_j} (resp. $A_{P_j}^-$) is true in the model: in this case, in $\boxplus_n \mathcal{P}$, this will block all inertia rules from state $i - 1$ forward until state j . At state j , an update rule will make A_j (resp. A_j^-) true. In $\oplus_n \mathcal{P}_D$, the truth of A_{P_j} (resp. $A_{P_j}^-$) only blocks inertia from state $j - 1$ to state j , but the truth of A_j (resp. A_j^-) is the same due to an update rule. Since the current state rules have, as pre-conditions, the atoms A_n (resp. A_n^-), and the rules for the predicate $\text{reject}(A_{n-1})$ are the same in both $\boxplus_n \mathcal{P}$ and $\oplus_n \mathcal{P}_D$, inertia from state $n - 1$ to n works the same way in both programs and thus, the truth value of the atoms A and $\text{not } A$ is the same;

- more than one of the A_{P_j} (resp. $A_{P_j}^-$) is true in the model: the reasoning is similar to the previous case.