A Language for Multi-dimensional Updates

João Alexandre Leite[1], José Júlio Alferes[1], Luís Moniz Pereira[1], Halina Przymusinska[2], and Teodor C. Przymusinski[3]

[1] CENTRIA, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal
jleite|jja|lmp@di.fct.unl.pt
[2] Computer Science, California State Polytechnic Univ. Pomona, CA 91768, USA
halina@cs.ucr.edu
[3] Computer Science, Univ. of California Riverside, CA 92521, USA
teodor@cs.ucr.edu

Abstract. Dynamic Logic Programming (DLP) was introduced to deal with knowledge about changing worlds, by assigning semantics to sequences of generalized logic programs, each of which represents a state of the world. These states permit the representation, not only of time, but also of specificity, strength of updating instance, hierarchical position of the knowledge source, etc. Subsequently, the Language of Updates LUPS was introduced to allow for the association, with each state, of a set of transition rules. It thereby provides for an interleaving sequence of states and transition rules within an integrated declarative framework. DLP (and LUPS), because defined only for a linear sequence of states, cannot deal simultaneously with more than a single dimension (e.g. time, hierarchies,...). To overcome this limitation, Multi-dimensional Dynamic Logic Programming (MDLP) was therefore introduced, so as to make it possible to organize states into arbitrary acyclic digraphs (DAGs). In this paper we now extend LUPS, setting forth a Language for Multi-dimensional Updates (MLUPS). MLUPS admits the specification of flexible evolutions of such DAG organized logic programs, by allowing not just the specification of the logic programs representing each state, but to the evolution of the DAG topology itself as well.

1 Introduction

Inspired by the earlier work on program updates [14, 20, 22, 25], [1] introduces the paradigm of *Dynamic Logic Programming (DLP)*. According to *DLP*, knowledge is given by a linearly ordered sequence of theories (each encoded as a generalized logic program, i.e. one where default negation may appear both in rule bodies and heads) that represent distinct and dynamically changing states of the world. The semantics of *DLP* ensures that all previous rules remain valid (by inertia) so long as they are not contradicted by newer (prevailing) rules, in which case they are rejected[1].

DLP can be used to model the stages of evolution of a single agent over time. And it can be also employed to model a linear hierarchy relationship of a group

[1] Similar approaches exist (eg. [4],[7]) and comparisons to DLP can be found in [7].

of agents, where rules from supervenient agents may be used to reject rules from superseded ones. But it cannot deal with both settings at once, and model the evolution of one such group of agents over time, inasmuch *DLP* is defined for linear sequences of states alone. To overcome this limitation, *Multi-dimensional Dynamic Logic Programming* ($\mathcal{MDLP}$) [16] was introduced. According to $\mathcal{MDLP}$ knowledge is given by a set of logic programs, indexed by collections of states organized into arbitrary directed acyclic graphs (*DAG*s) representing precedence relations (where an edge from a to b means that state b prevails over state a).

$\mathcal{MDLP}$ can provide a declarative semantics for modelling the evolution of a group of agents over time, where each node in the DAG is made to stand for the (updating) program acquired by some agent at some time point [17]. However, it does not comprise a language for specifying (or programming) forms of updating the agents. Indeed, if the knowledge of the various agents is already represented by an appropriate *DAG* of programs, $\mathcal{MDLP}$ determines its semantics. But how is knowledge evolution specified? How does the *DAG* itself grow by introducing new nodes after an update? How do new nodes connect with extant ones?

In order to address this issue, in this paper, and on the basis of the language *LUPS* [2], we define the *Language for Multi-dimensional Updates* (*MLUPS*). *MLUPS* is an update command language with the powerful capability of expressing concurrent updates of a number of agents. Intuitively, collections of update commands (such as asserts, retracts, persistent asserts, etc) are given to such agents, at each time point. As a result, each agent evolves as a sequence of programs, each containing the rules given to it at each point. In these time lines, according to $\mathcal{MDLP}$, rules from later programs may be used to reject rules from previous ones. Additionally, the agents can be hierarchically organized, so that rules from preferred agents prevail over rules from less preferred ones. The agents' hierarchy imposes a *DAG* configuration among the programs that constitute the agents' time lines. Different policies of imposing configurations over the agents are possible. E.g. by α_1 being preferred over α_2 one may simply want to state that rules given at a time point t to α_1 may be used to reject rules given to α_2 at t (we dub this policy *"equal role representation"*). But one may want to impose more, namely that any rule of α_1, even if given before t, may be used to reject rules given to α_2 at t (*"hierarchy prevalence representation"*). Another policy might be to state that rules given at a time point t to either α_1 or α_2 may be used to reject rules given to α_1 or α_2 at any time $< t$ (*"time prevalence representation"*).

The update commands of *MLUPS* can be made conditional on the present state of a collection of agents, and thus dynamically change with the changes in individual agents or changes in their mutual relationships. Moreover, the hierarchy among agents may be subjected to update commands as well, thereby influencing the dynamic reconfiguration of the *DAG*. Sequences of multiple *MLUPS* commands applied to the initial state of the graph result in a sequence of graphs whose semantics of intermediate and final nodes precisely coincides with the semantics of the $\mathcal{MDLP}$ generated by those commands.

The motivation for some of these concepts is best seen with an example.

Example 1. I have two advisers who help me decide how to manage my money. Each informs me about rules I might use in deciding what to buy, and which I store in two corresponding *MLUPS* agents, $adviser1$ and $adviser2$ (each evolving along a corresponding sequence of programs). These advisers could in turn have themselves their own advisers, in a more elaborate version of the example. Such advice rules could be: whenever there is a bull market, or if you're willing to take risks, buy TMT stocks; whenever there is a bear market do not buy TMT stocks, and opt for treasury bonds instead, unless you're willing to take risks ;... Moreover, in my knowledge base I also store rules concerning my own opinion on what to buy, that prevail over the rules given by my advisers. For example: if my budget is below a certain limit, then I do not buy anything; I never buy both TMT stocks and treasury bonds; if I come in possession of a lot of money, and I have enough invested in bonds, then I'm willing to risk. These can be stored in a *MLUPS* agent called $myself$, which is of higher priority than $adviser1$ and $adviser2$ in the agent hierarchy, and where even older rules from $myself$ prevail over newer ones coming from the advisers (i.e. for configuring the DAG, the connections between the sequence corresponding to $myself$ and those corresponding to $adviser1$ and $adviser2$ are established according to the hierarchy prevalence representation policy). I also consider another *MLUPS* agent, *reality*, where information about what is happening in the world is stored (e.g. about there being a bull or bear market).

The relationship between the *MLUPS* agents for the advisers may change over time. For example, initially I may not consider any hierarchical relation between them, and set a configuration policy so that more recent advice, no matter which adviser issued it, prevails over all older ones (i.e. time prevalence). Later I can lend priority to one of the advisers, and relate them through a hierarchical prevalence representation. Later still, I may change the hierarchy, and assign priority to the other adviser instead.

The modelling in *MLUPS* of a concrete simplified update history for this example is presented below in Example 4.

This scenario illustrates one possible application of MLUPS, corresponding to its use in specifying knowledge integration agents that keep an up-to-date view of the knowledge produced by a dynamic multi-agent community, assigning semantics to the combination of all knowledge pieces. MLUPS specifiable knowledge integration agents can be useful, for example, in legal reasoning scenarios where different legislative bodies constantly and independently produce laws that have to be combined and integrated according to existing collision principles. Such collision principles are directly encodable with MLUPS, either by means of the hierarchical and temporal relations (e.g. the collision principles *Lex Superior Derogat Legi Inferiori* and *Lex Posterior Derogat Legi Priori*) or by means of the prevalence modes(e.g. the collision principle *Lex Superior Priori Derogat Legi Inferiori Posterior* which corresponds to the *"hierarchy prevalence representation"*). In general, MLUPS allows for the specification of the evolution of agents whose knowledge depends on their view of other agent's knowledge within a dynamic multi-agent system. The generality provided by the underlying updatable logic programming based knowledge representation, and the flexibility provided by the commands that allow the specification and update of the hierarchies relating different agents and the evolution of such relations keep all doors open for MLUPS to be used as the update specification language on top of existing logic programming based multi-agent systems.

Throughout this paper, *MLUPS* will be introduced in an incremental fashion. We start with a simple version comprising a set of commands that permit the specification of the next temporal state of a group of agents, but whose organizing *DAG* evolves according to a fixed preference policy. Subsequently, we introduce a new set of commands to allow the hierarchy among the agents to evolve from state to state. Finally, we include yet another set of commands with the purpose of being able to change the policy of connecting the agents. In this submitted version, for the sake of the referees, we include an annex recapitulating the basic definitions of $\mathcal{MDLP}$.

2 MLUPS

In this section we set forth the *core* language of *MLUPS*. We start with a fixed set of agents $\mathcal{A} = \left\{\alpha^0, \alpha^1, \ldots, \alpha^n\right\}$ represented by nodes linked according to a fixed hierarchy encoded by the *hierarchy (labelled) graph* $\mathcal{H} = (\mathcal{A}, HE)$ where HE is a set of labelled edges of the form $\left(\alpha^i, \alpha^j, t\right)$ where $\alpha^i, \alpha^j \in \mathcal{A}$ and $t \in \mathcal{T}$, where $\mathcal{T} = \{0, 1, ..., n, ...\}$ is a set of (time) states. We call it *core* because the evolution of the topology of the DAG is fixed, in the sense that the edges linking the various nodes are determined as per an initial hierarchy (the temporal label in each edge of $\mathcal{H}$ is not used in this core version) plus the sequence of time states. At each time state a new node for each agent in $\mathcal{A}$ is created, linked to its previous instance. Such new nodes are linked amongst themselves according to the hierarchy $\mathcal{H}$. In Sect. 3 we extend *MLUPS* to cater for the specification of the evolution of $\mathcal{H}$. Because each new node is also linked to its predecessor, i.e. the node corresponding to the same agent at the previous time state, there is created a time state line in each agent. In this scenario, which gives rise to the "*equal role representation*" mentioned in the introduction, a rule given to an agent α^i at time t may be rejected by a rule given to the same α^i at any later time $t_l > t$, or by a rule given to a preferred agent α^j at the same t. No precedence exists over a higher ranked older program by a lower ranked but newer one. In Sect. 3, we also extend *MLUPS* to allow other evolution modes.

2.1 Syntax

The syntax of *MLUPS* is based on the *LUPS* commands [2], but now extended to cater for specifying in which agents are the updates performed, and in which agents are conditions testes. The simplest command consists of adding a rule to a new state of an agents α: **assert** *Rule*@α. For instance, in the setting of Example 1, the addition to $myself$, of a rule stating that I'm willing to risk if I have lots of money is accomplished by: **assert** $(risk \leftarrow money)@myself$.

In general, the addition of a rule to an agent may depend upon some preconditions to be verified at the most recent nodes of some set of agents. For this purpose, a general assert statement has the form:

$$\textbf{assert } \mathit{Rule}@\alpha \textbf{ when } L_1@\Omega_1, \ldots, L_k@\Omega_k$$

The meaning of this statement is that, if L_1 holds at the most recent nodes of the agents in the set Ω_1, and ..., and L_k holds at the most recent nodes of the agents in the set Ω_k, then the command **assert** $\mathit{Rule}@\alpha$ must be executed (i.e *Rule* must be added to a new node of agent α).

While some update commands, such as the one above for $myself$, represent newly incoming information, and thus are one-time non-persistent commands (i.e. the rule $risk \leftarrow money$, though it may remain valid by inertia for subsequent states, is added only once), some other commands are liable to be persistent, i.e. to remain in force until cancelled. An example of such persistent commands can be found in Example 1, e.g. in the statement "whenever there is a bear market do not buy TMT stocks". This statement is stating that the fact $not\,buy(stocks)$ is to be asserted (at $adviser2$) if *bear* is true at *reality*, and that this command, instead of being valid just at the time it is given, should persist in the future (perhaps till cancelled later). Such persistent update statements have the form: **always assert** $\mathit{Rule}@\alpha$ **when** *Conds*, where *Conds* is as explained above for assert statements. For cancelling persistent assert commands, *MLUPS* includes a commands **cancel assert** $\mathit{Rule}@\alpha$, and a corresponding general statement with the **when** conditions.

For the retraction of added rules, *MLUPS* offers non-persistent and persistent retract commands. The former indicates the retraction of a rule from an agent at the moment it is given (if some conditions are met, in case **when** conditions are mentioned); the latter states that from the moment it is given onwards, whenever some conditions are met, the retraction is to be performed.

As in LUPS, in the commands for asserting or retracting rules (both persistent and non-persistent) the rule may be preceded by the keyword **event**, in which case it is added to (resp. retracted from) the next state but retracted (resp. reasserted) immediately afterwards.

More precisely:

Definition 1 (MLUPS Commands and Statements). *An MLUPS* command *is a propositional expression of any of the forms*[2]*:*

$$\begin{array}{ll} [\textbf{always}]\ \textbf{assert}\ [\textbf{event}]\ R@\alpha & \textbf{cancel assert}\ R@\alpha \\ [\textbf{always}]\ \textbf{retract}\ [\textbf{event}]\ R@\alpha & \textbf{cancel retract}\ R@\alpha \end{array}$$

where R is a rule and $\alpha \in \mathcal{A}$.

An MLUPS statement *is a command extended with conditions, of the form:*

$$<\mathit{command}>\ \textbf{when}\ L_1@\Omega_1, \ldots, L_k@\Omega_k$$

[2] By [**keyword**] we mean either the presence or absence of **keyword**. For example, **assert event** $R@\alpha$ and **always assert event** $R@\alpha$ are both commands of *MLUPS*.

where < command> is any one of the above commands, each L_i is literal from $\mathcal{L}$, and each $\Omega_i \subseteq \mathcal{A}$.

We establish several conventions (used throughout the paper) to simplify the syntax of the **when** statement, namely: if conjunction of literals refers to the same set of agents Ω, instead of $L_1@\Omega, \ldots, L_k@\Omega$ we write $\{L_1, \ldots, L_k\} @\Omega$; if a set of agents Ω has a single element, instead of $L@\{\alpha\}$ we write $L@\alpha$; if $\Omega = \mathcal{A}$, instead of $L@\mathcal{A}$ we write L.

Definition 2 (MLUPS Program). *An* MLUPS program *is a sequence of sets of statements.*

We use the notation $U_1 \otimes ... \otimes U_n$ to represent an *MLUPS* program where each U_i is a set of statements.

Knowledge can be queried, wrt sets of agents, at any time state $t \leq n$, where n is the current time state. A query is denoted by **holds** $L_1@\Omega_1, \ldots, L_k@\Omega_k$ **at** t?, where each L_i is a literal from $\mathcal{L}$, and each $\Omega_i \subseteq \mathcal{A}$.

2.2 Semantics

An *MLUPS* program builds a corresponding *MDLP*, and its semantics is determined by the semantics of the *MDLP*. Accordingly, for defining the semantics of an *MLUPS* programs, all it needs to be done is to define what is the corresponding built *MDLP*.

Let $\mathcal{U} = U_1 \otimes ... \otimes U_n$ be an *MLUPS* programs. At every time state t we determine the corresponding *MDLP*, $\Upsilon_t(\mathcal{U}) = \mathcal{P}_t = (\mathcal{P}_{D_t}, D_t)$ where $D_t = (V_t, E_t)$ is the *MDLP DAG* and $\mathcal{P}_{D_t} = \{P_v : v \in V_t\}$. The *MDLP DAG* D_t will contain all the existing nodes at the previous time state, together with a new node for each agent, indexed by the new time state t. These new nodes will be connected among each other according to the hierarchy in $\mathcal{H}$, i.e., if in $\mathcal{H}$ $\alpha 1$ prevails over $\alpha 2$ we add an edge$(\alpha 2_t, \alpha 1_t)$. Moreover, they are connected to the remainder of the graph according to the intuition presented before, i.e. for each agent α we add an edge (α_{t-1}, α_t). Note again that in this simple version there are no edges directly relating nodes that are not related either by $\mathcal{H}$ or by $\mathcal{T}$. Formally:

Definition 3 (MDLP DAG at time state t). *The* MDLP DAG *at time state t is $D_t = (V_t, E_t)$, where V_t is defined as follows: $V_0 = \left\{\alpha_0^k : \alpha^k \in \mathcal{A}\right\}$ and $V_t = V_{t-1} \cup \left\{\alpha_t^k : \alpha^k \in \mathcal{A}\right\}$. E_t is defined as follows:*

$$E_0 = \{(\alpha_0^j, \alpha_0^k) : \left(\alpha^j, \alpha^k, \_\right) \in HE\}$$
$$E_t = E_{t-1} \cup \{\left(\alpha_{t-1}^k, \alpha_t^k\right) : \alpha^k \in \mathcal{A}\} \cup \{(\alpha_t^j, \alpha_t^k) : \left(\alpha^j, \alpha^k, \_\right) \in HE\}$$

Given a set of *MLUPS* statements, one has to first determine which of those are executable, i.e. which of those have their **when** conditions verified. For this purpose, we have first to evaluate the **when** conditions at the appropriate sets of agents. Based on this, the executable commands may be determined.

Definition 4 (Valuation). *Given an* MDLP $\mathcal{P}_t = (\mathcal{P}_{D_t}, D_t)$*, a time state* $i \leq t$*, and an expression* $\phi = L_1@\Omega_1, \ldots, L_k@\Omega_k$*. We say that* $\bigoplus_i \mathcal{P}_t \models \phi$ *iff* $\bigoplus_{\{\alpha_i^j : \alpha^j \in \Omega_1\}} \mathcal{P}_t \models L_1 \wedge ... \wedge \bigoplus_{\{\alpha_i^j : \alpha^j \in \Omega_k\}} \mathcal{P}_t \models L_k$*. If* $i = t$ *we simply write* $\bigoplus \mathcal{P}_t \models \phi$*.*

Definition 5 (Executable Commands). *Let* U *be a set of statements and* $\mathcal{P}_t = (\mathcal{P}_{D_t}, D_t)$ *an MDLP. By the set of executable commands corresponding to* U*, wrt* $\mathcal{P}_t$ *we mean* $\Delta_U^{\mathcal{P}_t}$ *defined as follows:*

$$\Delta_U^{\mathcal{P}_t} = \{<command> : \ (<command> \ \textbf{when} \ \psi \in U) \wedge \bigoplus \mathcal{P}_t \models \psi\}$$

We can now determine the object level programs that are associated with each vertex of the *MDLP DAG*. This will be accomplished, as in *LUPS*, by determining a set of persistent commands at each time state, to which we add the new commands. The resulting set of commands determines each of the object level programs. To be able to retract rules and properly handle non-inertial commands, we need to augment the language of the resulting multi-dimensional dynamic program with a new propositional variables "$n(R)$" for every rule R appearing in the original *MLUPS* program, and new propositional variables "$ev(R, S)$" for every rule R appearing in a non-inertial command in the original *MLUPS* program, and every time state S. The object level programs are then obtained as per the following definition:

Definition 6 (Object Level Programs at time state t). *The set of object level programs at time state t,* $\mathcal{P}_{D_t} = \{P_v : v \in V_t\}$*, is inductively defined as follows:*

Base Step: $P_{\alpha_0^k} = \{\}$ *and* $PC_0 = \{\}$

Inductive Step:

$$\begin{aligned}
PC_t = PC_{t-1} &\cup \{\textbf{assert}\, R@\alpha\, \textbf{when}\, \phi : \textbf{always assert}\, R@\alpha\, \textbf{when}\, \phi \in U_t\} \cup \\
&\cup \{\textbf{retract}\, R@\alpha\, \textbf{when}\, \phi : \textbf{always retract}\, R@\alpha\, \textbf{when}\, \phi \in U_t\} \cup \\
&\cup \{\textbf{assert event}\, R@\alpha\, \textbf{when}\, \phi : \textbf{always assert event}\, R@\alpha\, \textbf{when}\, \phi \in U_t\} \cup \\
&\cup \{\textbf{retract event}\, R@\alpha\, \textbf{when}\, \phi : \textbf{always retract event}\, R@\alpha\, \textbf{when}\, \phi \in U_t\} - \\
&- \left\{\textbf{assert [event]}\, R@\alpha\, \textbf{when}\, \phi : \textbf{cancel assert}\, R@\alpha \in \Delta_{U_t}^{\mathcal{P}_{t-1}}\right\} - \\
&- \left\{\textbf{assert [event]}\, R@\alpha\, \textbf{when}\, \phi : \textbf{always retract [event]}\, R@\alpha \in \Delta_{U_t}^{\mathcal{P}_{t-1}}\right\} - \\
&- \left\{\textbf{retract [event]}\, R@\alpha\, \textbf{when}\ \phi : \textbf{cancel retract}\, R@\alpha \in \Delta_{U_t}^{\mathcal{P}_{t-1}}\right\} - \\
&- \left\{\textbf{retract [event]}\, R@\alpha\ \textbf{when}\ \phi : \textbf{always assert [event]}\, R@\alpha \in \Delta_{U_t}^{\mathcal{P}_{t-1}}\right\}
\end{aligned}$$

$$NU_t = U_t \cup PC_t$$

$$\begin{aligned}
P_{\alpha_t^k} = &\left\{not\, n(R) \leftarrow : \textbf{retract}\, R@\alpha^k \in \Delta_{NU_t}^{\mathcal{P}_{t-1}}\right\} \cup \\
\cup &\left\{n(R) \leftarrow; h(R) \leftarrow b(R), n(R) : \textbf{assert}\, R@\alpha^k \in \Delta_{NU_t}^{\mathcal{P}_{t-1}}\right\} \cup \\
\cup &\left\{h(R) \leftarrow b(R), ev(R, t) : \textbf{assert event}\, R@\alpha^k \in \Delta_{NU_t}^{\mathcal{P}_{t-1}}\right\} \cup \\
\cup &\left\{not\, n(R) \leftarrow ev(R, t) : \textbf{retract event}\, R@\alpha^k \in \Delta_{NU_t}^{\mathcal{P}_{t-1}}\right\} \cup \\
\cup &\{not\, ev(R, t-1) \leftarrow; ev(R, t) \leftarrow\}
\end{aligned}$$

where if r is a clause (or rule) of the form $L_0 \leftarrow L_1, \ldots, L_n$, by $h(r)$ we mean L_0, by $b(r)$ we mean $L_1, \ldots, L_n$.

As mentioned before, the semantics of an *MLUPS* program is determined by the semantics of the so built *MDLP*:

Definition 7 (MLUPS Semantics). *Let $\mathcal{U} = U_1 \otimes ... \otimes U_n$ be an* MLUPS *program. A query* **holds** $L_1@\Omega_1, \ldots, L_k@\Omega_k$ **at** t? *is true in $\mathcal{U}$ iff $\bigoplus \Upsilon_t(\mathcal{U}) \models L_1@\Omega_1, \ldots, L_k@\Omega_k$, or, equivalently, iff $\bigoplus \mathcal{P}_t \models L_1@\Omega_1, \ldots, L_k@\Omega_k$.*

The semantics of *LUPS* [2, 15] coincides with a fragment of *MLUPS*. Such fragment is obtained by restricting the set of agents $\mathcal{A}$ to contain one agent only.

Theorem 1. *Let $\mathcal{Q} = Q_1 \otimes ... \otimes Q_n$ be an* MLUPS *program (with $\mathcal{A} = \{\alpha\}$) and $\mathcal{U} = U_1 \otimes ... \otimes U_n$ be a* LUPS *update program such that:*

$$< command> \; R@\alpha \textbf{ when } L_1@\alpha, \ldots, L_k@\alpha \in Q_i \quad iff$$
$$< command> \; R \textbf{ when } L_1, \ldots, L_k \in U_i$$

Then, the query **holds** $L_1, \ldots, L_k$ **at** t? *is true in $\mathcal{U}$ (according to [15]) iff the query* **holds** $L_1@\alpha, \ldots, L_k@\alpha$ **at** t? *is true in $\mathcal{Q}$.*

3 Extending MLUPS with DAG Commands

The *MLUPS* framework presented in the previous section only allows the evolution of an *MDLP* whose structure, encoded by the *MDLP DAG*, is quite strict in the sense that the hierarchy relating the different agents is fixed and there are no edges directly relating different agents in different time states. In this section we propose two general extensions to the basic *MLUPS* that allow for a more flexible evolution of the *MDLP DAG*, in particular, enabling removal of these two limitations.

3.1 Hierarchy Commands

We start with an extension that permits the hierarchy *DAG* to evolve, instead of the fixed hierarchy among agents $\mathcal{H}$ of the previous section. We thus need some way to specify the addition and removal of hierarchy edges between pairs of agents. To this purpose, to the basic *MLUPS* commands we add commands for the manipulation of the hierarchy $\mathcal{H}$ (with $\alpha^j, \alpha^k \in \mathcal{A}$):

$$\textbf{add\_hierarchy\_edge } \alpha^j \rightarrow \alpha^k$$
$$\textbf{remove\_hierarchy\_edge } \alpha^j \rightarrow \alpha^k$$

The intuitive reading of these commands is straightforward: the first indicates that to the hierarchy graph we must add an edge from α^j to α^k, and the second one indicates that from the hierarchy graph we must remove any existing edge from α^j to α^k.[3] With these commands, we no longer need an initial fixed hierarchy graph. The hierarchy is given by a hierarchy graph that is initially empty and evolves from time state to time state, and defined as follows (considering an *MLUPS* program $\mathcal{U} = U_1 \otimes ... \otimes U_n$):

Definition 8 (Hierarchy DAG at time state t). *The hierarchy (labelled) DAG at time state t is $\mathcal{H}_t = (\mathcal{A}, HE_t)$ where HE_t is defined as follows: $HE_0 = \{\}$ and*

$$\begin{aligned} HE_t = HE_{t-1} \cup \left\{ \left(\alpha^j, \alpha^k, t\right) : \textbf{add\_hierarchy\_edge}\ \alpha^j \rightarrow \alpha^k \in U_t \right\} - \\ - \left\{ \left(\alpha^j, \alpha^k, \_\right) : \textbf{remove\_hierarchy\_edge}\ \alpha^j \rightarrow \alpha^k \in U_t \right\} \end{aligned}$$

if $\mathcal{H}_t$ is a DAG. Otherwise, it is not defined.

The semantics of this extended language is equal to the one in the previous section, except that, in Definition 3, we replace HE by HE_i.

3.2 Prevalence Mode Commands

In the previous section we've proposed *MLUPS* to construct *MDLP*s that evolve according to the "*equal role representation*". As we have seen in Example 1, other representation policies might be needed in practice. For allowing for other policies, in this section we introduce another extension to the *MLUPS* language, by means of a set of commands to allow a flexible evolution of the *MDLP DAG*. In this more general setting, the *MDLP DAG* contains, besides the edges relating each agent in different time states and several agents inside each time state, another set of edges specified by user defined functions. Such functions specify the evolution of the *MDLP DAG*, by defining which edges should be created at each time state transition. This fosters the construction of more general *MDLP DAG*s, among which those representing the "*hierarchy prevalence representation*" and "*time prevalence representation*" modes [17]. According to *hierarchy prevalence*, any rule indexed by a higher ranked agent overrides any lower ranked agent's rule, independently of the time state it is indexed by. According to *time prevalence*, any rule indexed by a more recent time state overrides any older rule, independently of which agents these rules belong to.

Instead of concentrating on some specific policies (such as the ones mentioned above and in the example, here we consider general functions $f \in \mathcal{F}$ with signature $f : \mathcal{A}^2 \times \{<, =\} \times \mathcal{T}^3 \longrightarrow 2^{\{+,-\} \times (\mathcal{A} \times \mathcal{T})^2}$. Each function $f \in \mathcal{F}$, defines a set of edges of the forms $+(\alpha^j_{t_1}, \alpha^k_{t_2})$ and $-(\alpha^j_{t_1}, \alpha^k_{t_2})$ where $\alpha^j, \alpha^k \in \mathcal{A}, t_1, t_2 \in \mathcal{T}$, given a pair of agents, their relation, the current time state, and two other time states indicating when the prevalence mode represented by f and the agents relation were set. Below we will show some examples of such functions. The new commands are (with $\alpha^j, \alpha^k \in \mathcal{A}$, and $f \in \mathcal{F}$):

$$\textbf{add\_prevail\_mode}\ \alpha^j \overset{f}{\longleftrightarrow} \alpha^k$$
$$\textbf{remove\_prevail\_mode}\ \alpha^j \overset{f}{\longleftrightarrow} \alpha^k$$

[3] A **when** statement could also be added to these commands, its effect being as for the commands of the basic language. For simplicity we omit it.

Since the prevalence modes should persist until removed, we need to keep info about the prevalence mode at each time state. As will become clear when we look closer at the interesting cases of such functions, we also need to keep track of when such prevalence modes were set. This is formalized as follows:

Definition 9 (Prevalence Mode at time state t). *The prevalence mode at time state t, PM_t, is a set of tuples of the form $\{(\{\alpha^j, \alpha^k\}, f, n) : \alpha^j, \alpha^k \in \mathcal{A}, f \in \mathcal{F}, n \in \mathcal{T}\}$, where each tuple references a function, the time state when it was set, and the two agents involved. It is defined as follows: $PM_0 = \{\}$ and*

$$PM_t = PM_{t-1} \cup \{(\{\alpha^j, \alpha^k\}, f, t) : \textbf{add\_prevail\_mode}\ \alpha^j \overset{f}{\longleftrightarrow} \alpha^k \in U_t\} - \\ -\{(\{\alpha^j, \alpha^k\}, f, \_) : \textbf{remove\_prevail\_mode}\ \alpha^j \overset{f}{\longleftrightarrow} \alpha^k \in U_t\}$$

These commands affect the DAG at state t by adding and removing edges.

Definition 10 (Added and Removed Prevalence Edges at time state t). *The set of added (resp. removed) prevalence edges at time state t is PE_t^+ (resp. PE_t^-), is defined as follows:*

$$PE_t^+ = \{(\alpha^j_{t_1}, \alpha^k_{t_2}) : +(\alpha^j_{t_1}, \alpha^k_{t_2}) \in PE_t\}$$
$$PE_t^- = \{(\alpha^j_{t_1}, \alpha^k_{t_2}) : -(\alpha^j_{t_1}, \alpha^k_{t_2}) \in PE_t\}$$

where $PE_t = PE_t^< \cup PE_t^=$, where $PE_t^< = \bigcup f(\alpha^j, \alpha^k, <, t, n, m)$ for all $f, \alpha^j, \alpha^k, n, m$ such that $(\{\alpha^j, \alpha^k\}, f, n) \in PM_t$, and $(\alpha^j, \alpha^k, m) \in HE_t$, and where $PE_t^= = \bigcup f(\alpha^j, \alpha^k, =, t, n, 0)$ for all f, α^j, α^k, n such that $(\{\alpha^j, \alpha^k\}, f, n) \in PM_t$, and $(\alpha^j, \alpha^k, \_), (\alpha^k, \alpha^j, \_) \notin HE_t$.

Since some of the functions that specify which edges are to be added or removed are sensitive to the existence (or not) of a hierarchical relation between the pair of involved agents, a test is first performed ($(\alpha^j, \alpha^k, m) \in HE_t$ or $(\alpha^j, \alpha^k, \_), (\alpha^k, \alpha^j, \_) \notin HE_t$) resulting in the parameter $<$ or $=$ being passed to the function. Then, PE_t will contains all edges (added and removed) defined by all current prevalence modes, defined by the functions f, for each pair of agents. Such edges are then separated in two sets, PE_t^+ and PE_t^-, containing the edges to be added and those to be removed, respectively.

Note that a pair of agents can have more than one prevalence mode at each time. To restrict to a single prevalence mode at each time, all that needs to be done is to issue the command **remove_prevail_mode** $\alpha^j \overset{F}{\longleftrightarrow} \alpha^k$ when issuing the command **add_prevail_mode** $\alpha^j \overset{f}{\longleftrightarrow} \alpha^k$.

The *MDLP DAG* at time state t is now defined as:

Definition 11 (MDLP DAG at time state t). *The* MDLP DAG *at time state t is $D_t = (V_t, E_t)$, where V_t is defined as follows: $V_0 = \{\alpha^k_0 : \alpha^k \in \mathcal{A}\}$ and $V_t = V_{t-1} \cup \{\alpha^k_t : \alpha^k \in \mathcal{A}\}$. E_t is defined as follows: $E_0 = \{\}$ and*

$$E_t = E_{t-1} \cup \{(\alpha^k_{t-1}, \alpha^k_t) : \alpha^k \in \mathcal{A}\} \cup \{(\alpha^j_t, \alpha^k_t) : (\alpha^j, \alpha^k, \_) \in HE_t\} \cup PE_t^+ - PE_t^-$$

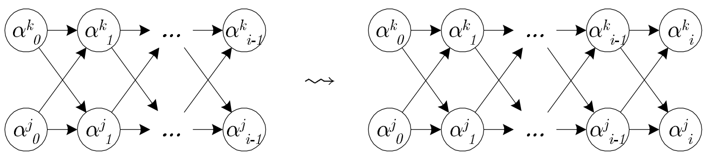
if D_t is a DAG. Otherwise, it is not defined.

We follow up with interesting examples of functions. Recall that function $f\left(\alpha^j, \alpha^k, h, t, n, m\right)$ parameters have the following meaning: α^j and α^k are the two agents involved; $h \in \{<, =\}$ contains the hierarchy relation between α^j and α^k: $h =$ " $<$ " means that $\alpha^j < \alpha^k$, and $h =$ " $=$ " means that α^j and α^k are not directly hierarchically related; t is the current time state; n is the time state when the prevalence mode f was set; m is the time state when the current hierarchical relation between α^j and α^k was set. Not all parameters will be used by all functions.

Time Prevalence: According to this prevalence mode, any rule indexed by a more recent time state overrides any older rule, independently of which of the two agents these rules belong to. Motivation for this prevalence mode can be found in [17]. The function specifying this mode is defined as follows:

$$f_{tp}\left(\alpha^j, \alpha^k, \_, t, \_, \_\right) = \{+(\alpha^j_{t-1}, \alpha^k_t), +(\alpha^k_{t-1}, \alpha^j_t)\}$$

Example 2. Consider the programs $U_1 = \{\textbf{add\_prevail\_mode}\ \alpha^j \xleftrightarrow{f_{tp}} \alpha^k\}$. The transition from time states $i-1$ to i is represented in the following Figure:



The previous function only makes time prevail after being issued. We may want a function that also sets the past to the same time prevalence mode. Such function would be:

$$f_{atp}\left(\alpha^j, \alpha^k, \_, t, \_, \_\right) = \{+(\alpha^j_{p-1}, \alpha^k_p), +(\alpha^k_{p-1}, \alpha^j_p) : 1 < p \leq t\}$$

Hierarchy Prevalence: According to this prevalence mode, any rule indexed by a higher ranked agent overrides any lower ranked agent's rule, starting when both the prevalence mode and the hierarchy between the two agents are set. Motivation for this prevalence mode can be found in [17].

$$\begin{aligned} f_{hp}\left(\alpha^j, \alpha^k, <, t, n, m\right) &= \{+(\alpha^j_t, \alpha^k_p) : p = max(n, m), 0 < p < t\} \cup \\ &\cup \{-(\alpha^j_{t-1}, \alpha^k_p) : p = max(n, m), 0 < p < t-1\} \\ f_{hp}\left(\_, \_, =, \_, \_, \_\right) &= \{\} \end{aligned}$$

Note that if there is no hierarchical relation between the two agents, no edges are added nor removed.

Example 3. Consider the following program:

$$\begin{aligned} U_1 = &\ \{\textbf{add\_prevail\_mode}\ \alpha^j \xleftrightarrow{f_{hp}} \alpha^k; \textbf{add\_hierarchy\_edge}\ \alpha^j \rightarrow \alpha^k\} \\ U_{i-2} = &\ \{\textbf{remove\_hierarchy\_edge}\ \alpha^j \rightarrow \alpha^k; \textbf{add\_hierarchy\_edge}\ \alpha^k \rightarrow \alpha^j\} \end{aligned}$$

The transition from time states $i-1$ to i is represented in the following Figure:

$$\begin{tikzcd}
\alpha^k_0 \arrow[r] & \alpha^k_1 \arrow[r] & \cdots \arrow[r] & \alpha^k_{i-3} \arrow[r] & \alpha^k_{i-2} \arrow[r] \arrow[d] & \alpha^k_{i-1} \arrow[dl] \arrow[d] & \rightsquigarrow & \alpha^k_0 \arrow[r] & \alpha^k_1 \arrow[r] & \cdots \arrow[r] & \alpha^k_{i-3} \arrow[r] & \alpha^k_{i-2} \arrow[r] \arrow[d] & \alpha^k_{i-1} \arrow[r] \arrow[d] & \alpha^k_i \arrow[d] \arrow[dll] \\
\alpha^j_0 \arrow[r] & \alpha^j_1 \arrow[r] \arrow[u] & \cdots \arrow[r] & \alpha^j_{i-3} \arrow[r] \arrow[u] \arrow[ull] & \alpha^j_{i-2} \arrow[r] & \alpha^j_{i-1} & & \alpha^j_0 \arrow[r] & \alpha^j_1 \arrow[r] \arrow[u] & \cdots \arrow[r] & \alpha^j_{i-3} \arrow[r] \arrow[u] \arrow[ull] & \alpha^j_{i-2} \arrow[r] & \alpha^j_{i-1} \arrow[r] & \alpha^j_i
\end{tikzcd}$$

At this stage, where the general *MLUPS* language has been defined, we can come back to the Example of the Introduction.

Example 4. Consider the following simplified update history for Example 1, and corresponding *MLUPS* program. Lack of space prevents us from elaborating the example further.

At the start I consider that $myself$ is higher in the hierarchy than the advisers, that hierarchy prevails, and that the advisers are related by a time prevalence mode. Moreover reality is the highest in the hierarchy. This can be easily coded in MLUPS, and is omitted for brevity. Then, at time 2, I add to $myself$ a rule stating that I'm willing to risk if I have lots of money. Moreover, $adviser1$ tells me to buy bonds, and whenever there is a bull market $adviser2$ tells me that, in a bull market situation, his advice is then to buy stocks, and not to buy bonds:

assert $(risk \leftarrow money)@myself$
assert $buy(bonds)@adviser1$
always $(buy(stocks) \leftarrow bull)@adviser2$ **when** $bull@reality$
always $(not\, buy(bonds) \leftarrow bull)@adviser2$ **when** $bull@reality$

The reader can check that, at this point, $buy(bonds)$ holds at $myself$.

At time 3, I'm informed that there is a bull market: **assert** $bull@reality$. Now, $buy(stocks)$ and $not\, buy(bonds)$ both hold.

At time 4 I decide to change my priorities, impose a hierarchical relation where $adviser1$ is higher than $adviser2$, and consider, from now on, that hierarchy should prevail. This can be accomplished by giving the commands (where f_{hp} is as defined in page 11):

add_hierarchy_edge $adviser2 \rightarrow adviser1$
add_prevail_mode $adviser2 \overset{f_{hp}}{\longleftrightarrow} adviser1$

Moreover, $adviser1$ tells me that, if I'm not willing to risk, I should definitely not buy stocks: **assert** $(not\, buy(stocks) \leftarrow not\, risk)@adviser1$. At this point both $not\, buy(stocks)$ and $not\, buy(bonds)$ hold.

Suppose that at time 5 I receive lots of money: **assert** $money@myself$. As expected, $buy(stocks)$ and $not\, buy(bonds)$ hold.

Finally, at time 6, I'm informed that there is no longer a bull market: **assert** $not\, bull@reality$. Accordingly, both $buy(bonds)$ and $not\, buy(stocks)$ now hold.

4 Concluding Remarks

We have presented *MLUPS*, a language for specifying dynamic and multi-dimensional updates in non-monotonic agents knowledge bases. These are represented by generalized logic programs allowing default negation in rule heads. We provided a declarative semantics for the language, by translating *MLUPS* programs into sequences of logic programs, whose semantics is determined by multi-dimensional dynamic logic programming *MDLP*. Though not described here, we have also implemented the *core MLUPS* language, with a fixed hierarchy prevalence mode, and enriched it with the capability to dynamically change the hierarchy *DAG*. The implementation is available from the authors.

Over recent years, the notion of agency has claimed a fundamental role in defining the trends of contemporary research, virtually invading every sub-field of Computer Science [13]. Although commonly implemented by means of imperative languages, mainly for reasons of efficiency, the agent concept has more recently increased its influence in the research and development of computational logic based systems. Since efficiency is not always the crucial issue, but clear specification and correctness is, *Logic Programming* and *Non-monotonic Reasoning* have been brought back into the spotlight [3, 23]. To this accrues the recent significant improvements in the efficiency of *Logic Programming* implementations for *Non-monotonic Reasoning* (e.g. [21, 26, 5]). Besides allowing for a unified declarative and procedural semantics, eliminating the traditional wide gap between theory and practice, the use of several and quite powerful results in the field of non-monotonic extensions to *LP*, such as belief revision, inductive learning, argumentation, preferences, abduction, etc.[23] can represent an important composite added value to the design of rational agents.

The language MLUPS is at the core of an agent architecture [18] conceived with the intention of providing, on a sound theoretical basis, a common agent framework based on the strengths of Logic Programming, so as to allow the combination of such non-monotonic knowledge representation and reasoning mechanisms developed in recent years. Rational agents, in our opinion, will require an admixture of any number of those reasoning mechanisms for carrying out their tasks.

Our language for updates of logic programs borrows from and is closely related to action languages, which can be translated into logic programs (cf. [10]), but extends them to multiple dimensions and agents. A change to the knowledge base may be considered as an action, where the execution of actions may depend on other actions and conditions. However, the two approaches are significantly different, even in the single dimension single agent case. Indeed, action languages are tailored for planning and reasoning about actions, rather than for update specification, and actions are restricted to sets of literals (fluents) rather than representing updates of sets of rules or logic programs.

In [18] we relate our architecture, and to some extent MLUPS, with the somehow related approaches of [11, 8, 24].

A deeper study of applications of *MLUPS* is the subject of ongoing and future work. Namely, bridging the gap between knowledge updates and *reasoning* *about actions*, applying *MLUPS* as a language for combining knowledge and beliefs in multi-agent systems (e.g. combining different e-commerce policies), applying knowledge update methodology to the domain of software engineering for software maintenance and verification of program correctness.

We are also studying the combination of *MLUPS* with other extensions of *LUPS*, such as the specification of updates conditional on external events [6], and the nesting of update commands.

We believe *MLUPS* has an enormous potential begging to be tapped, and opens up new vistas for the logic programming approach to distributed dynamic knowledge change.

Acknowledgment The authors acknowledge the support of FCT project 40958 "FLUX - FleXible Logical Updates".

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000. Abstract titled *Dynamic Logic Programming* appeared in Procs. of KR-98.
2. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS : A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002. Short version appeared in Procs of LPNMR-99, LNAI-1730.
3. M. Bozzano, G. Delzanno, M. Martelli, V. Mascardi, and F. Zini. Logic programming and multi-agent system: A synergic combination for applications and semantics. In *The Logic Programming Paradigm - A 25-Year Perspective*. Springer, 1999.
4. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In *Procs. of ICLP-99*. MIT Press, 1999.
5. DLV. The DLV project - a disjunctive datalog system (and more), 2000. Available at `http://www.dbai.tuwien.ac.at/proj/dlv/`.
6. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In *Procs of IJCAI'01*. Morgan Kaufmann, 2001.
7. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002. To appear.
8. M. Fisher. A survey of concurrent METATEM: The language and its applications. In *Procs of ICTL'94*, volume 827 of *LNAI*. Springer, 1994.
9. M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Procs. of ICLP-88*. MIT Press, 1988.
10. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 3(16), 1998.
11. K. V. Hindriks, F. S. de Boer, W. van der Hoek, and J. C. Meyer. Formal semantics for an abstract agent programming language. In *Procs of ATAL'97*, volume 1365 of *LNAI*. Springer, 1998.
12. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35:39–78, 1998.

13. N. R. Jennings, K. Sycara, and M. Wooldridge. A roadmap of agent research and development. *Journal of Autonomous Agents and Multi-Agent Systems*, 1(1):7–38, 1998.
14. H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In *Procs. of KR-91*. Morgan Kaufmann, 1991.
15. J. A. Leite. A modified semantics for LUPS. In *Procs of EPIA'01*, volume 2258 of *LNAI*, pages 261–275. Springer, 2001.
16. J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic knowledge representation. In *Procs. of LPNMR-01*, volume 2173 of *LNAI*. Springer, 2001. A preliminary version appeared in Procs. of CLIMA-00.
17. J. A. Leite, J. J. Alferes, and L. M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. In *Procs of EPIA'01*, volume 2258 of *LNAI*, pages 276–289. Springer, 2001.
18. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In J. J. Meyer and M. Tambe, editors, *Intelligent Agents VIII — Procs. of ATAL'01*, volume 2333 of *LNAI*. Springer, 2002.
19. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In *Procs. of KR-92*. Morgan-Kaufmann, 1992.
20. Victor W. Marek and Mirosław Truszczyński. Revision programming. *Theoretical Computer Science*, 190(2):241–277, 20 January 1998.
21. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal LP. In *Procs. of LPNMR'97*, volume 1265 of *LNAI*. Springer, 1997.
22. T. C. Przymusinski and H. Turner. Update by means of inference rules. *Journal of Logic Programming*, 30(2):125–143, 1997.
23. F. Sadri and F. Toni. Computational logic and multiagent systems: A roadmap, 1999. Available from `http://www.compulog.org`.
24. V. S. Subrahmanian, Piero Bonatti, Jürgen Dix, Thomas Eiter, Sarit Kraus, Fatma Ozcan, and Robert Ross. *Heterogeneous Agent Systems*. MIT Press/AAAI Press, 2000.
25. Marianne Winslett. Reasoning about action using a possible models approach. In *Procs. of NCAI-88*. AAAI Press, 1988.
26. XSB-Prolog. The XSB logic programming system, version 2.0, 1999. Available at `http://www.cs.sunysb.edu/ sbprolog`.

A Background

Object language: Following the tradition of [1, 2, 16], we will use *generalized logic programs (GLP)* which we briefly recapitulate here[4].

By a *generalized logic program* P in a language $\mathcal{L}$ we mean a finite or infinite set of propositional clauses of the form $L_0 \leftarrow L_1, \ldots, L_n$ where each L_i is a literal (i.e. an atom A or the default negation of an atom *not* A). If r is a clause

[4] The class of GLPs (i.e. logic programs that allow default negation in the premisses and heads of rules) can be viewed as a special case of yet broader classes of programs, introduced earlier in [12] and in [19], and, for the special case of normal programs, their semantics coincides with the stable models semantics [9]. In [2] the reader can find a possible motivation for using such programs, instead of using LPs with integrity constraints.

(or rule), by $H(r)$ we mean L_0, and by $B(r)$ we mean $L_1, \ldots, L_n$. If $H(r) = A$ (resp. $H(r) = not\, A$) then $not\, H(r) = not\, A$ (resp. $not\, H(r) = A$). By a (2-valued) *interpretation* M of $\mathcal{L}$ we mean any set of literals from $\mathcal{L}$ that satisfies the condition that for any A, *precisely one* of the literals A or $not\, A$ belongs to M. Given an interpretation M we define $M^+ = \{A : A \text{ is an atom}, A \in M\}$ and $M^- = \{not\, A : A \text{ is an atom, } not\, A \in M\}$. Wherever convenient we omit the default (negative) atoms when describing interpretations and models. Also, rules with variables stand for the set of their ground instances. We say that a (2-valued) interpretation M of $\mathcal{L}$ is a stable model of a generalized logic program P if $\xi(M) = least\,(\xi(P) \cup \xi(M^-))$, where $\xi(.)$ univocally renames every default literal $not\, A$ in a program or model into new atoms, say not_A. In the remaining, we refer to a GLP simply as a logic program (or LP).

Graphs: A *directed graph*, or *digraph*, $D = (V, E)$ is a pair of two finite or infinite sets $V = V_D$ of *vertices* and $E = E_D$ of pairs of vertices or (*directed*) *edges*. A *directed edge sequence from* v_0 *to* v_n is a sequence of edges $e_1, e_2, ..., e_n \in E_D$ such that $e_i = (v_{i-1}, v_i)$ for $i = 1, ..., n$. A *directed path* is a directed edge sequence in which all the edges are distinct. A *directed acyclic graph*, or *acyclic digraph (DAG)*, is a digraph D such that there are no directed edge sequences from v to v, for all vertices v of D. We say that $v < u$ if there is a directed path from v to u and that $v \leq u$ if $v < u$ or $v = u$. A labelled digraph is a digraph where a label is associated with each edge. For simplicity, we represent such edges by triples of the form (v_i, v_j, w) where $v_i, v_j \in V$ and w is the label of the edge. Labels are elements of some predefined set (e.g. natural numbers). All other notions defined above follow if we consider the digraph obtained from the labelled digraph by replacing each labelled edge (v_i, v_j, w) by the edge (v_i, v_j).

Multi-dimensional Dynamic Logic Programming: $\mathcal{MDLP}$ [16] is a generalization of *DLP* inasmuch as it allows for collections of states organized by arbitrary acyclic digraphs, and not just sequences of states, therefore assigning 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. A *Multi-dimensional Dynamic Logic Program (MDLP)*, $\mathcal{P}$, is a pair $(\mathcal{P}_D, D)$ where $D = (V, E)$ is a DAG and $\mathcal{P}_D = \{P_v : v \in V\}$ is a set of generalized logic programs in the language $\mathcal{L}$, indexed by the vertices $v \in V$ of D. We call *states* such vertices of D. For simplicity, we often leave the language $\mathcal{L}$ implicit.

Definition 12 (Stable Models at a set of states S). *Let $\mathcal{P} = (\mathcal{P}_D, D)$ be a* MDLP*, where $\mathcal{P}_D = \{P_v : v \in V\}$ and $D = (V, E)$. Let S be a set of states such that $S \subseteq V$. An interpretation M is a* stable model of $\mathcal{P}$ at the set of states S *iff $M = least\,([\rho\,(\mathcal{P})_S - Rej(S, M)] \cup Default\,(S, M))$, where:*

$$\rho\,(\mathcal{P})_S = \bigcup_{s \in S} \left(\bigcup_{i \leq s} P_i \right)$$
$$Rej(S, M) = \{r \in P_i \mid \exists s \in S, \exists r' \in P_j, i < j \leq s, h(r) = not\, h(r') \wedge M \vDash b(r')\}$$
$$Default\,(S, M) = \{not\, A \mid \nexists r \in \rho\,(\mathcal{P})_S : (h(r) = A) \wedge M \vDash b(r)\}$$

If some literal or conjunction of literals ϕ holds in all stable models of $\bigoplus \mathcal{P}$ at the set of states S, we write $\bigoplus_S \mathcal{P} \models \phi$. If $S = V$ we simply omit the reference and write $\bigoplus \mathcal{P} \models \phi$. If $S = \{s\}$ we write $\bigoplus_s \mathcal{P} \models \phi$.