Evolving Multi-agent Viewpoints – An Architecture

Pierangelo Dell'Acqua[1], João Alexandre Leite[2], and Luís Moniz Pereira[2]

[1] Department of Science and Technology, Campus Norrköping
Linköping University, Norrköping, Sweden, pier@itn.liu.se
[2] Centro de Inteligência Artificial - CENTRIA, Universidade Nova de Lisboa
2829-516 Caparica, Portugal, {jleite | lmp}@di.fct.unl.pt

Abstract. We present an approach to agents that can reason, react to the environment and are able to update their own knowledge as a result of new incoming information. Each agents' view of the social relationships among agents (itself and others) is represented by a graph, which in turn can be updated, allowing for the representation of such social evolution.

1 Introduction and Motivation

Over recent years, the notion of agency has claimed a major role in defining the trends of modern research. Influencing a broad spectrum of disciplines such as Sociology, Psychology, among others, the agent paradigm virtually invaded every sub-field of Computer Science. 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 *Logic Programming* (LP), such as belief revision, inductive learning, argumentation, preferences, abduction, etc.[15,17] can represent an important added value to the design of rational agents. These results, together with the improvement in efficiency (cf. [5,16,18]), allow *Logic Programming* and *Non-monotonic Reasoning* to accomplish a fruitful degree of combination between reactive and rational behaviours of agents, whilst preserving clear and precise specification enjoyed by declarative languages. This goal in mind, Kowalski and Sadri [11] advanced an agent architecture based on an observe-think-act cycle. It was further developed by Dell'Acqua and Pereira [6] to allow the agents to dynamically update their own knowledge base (whether intentional or extensional) as well as their own goals. The capability of updating provided to these agents is inherited by *Dynamic Logic Programming* [1].

Dynamic Logic Programming (DLP) was introduced, following the eschewing of performing updates on a model basis, to envisage updates as applying to logic programming rules making up a theory [14]. According to DLP, knowledge is conveyed by a set of theories (encoded as generalized logic programs) representing different states of the world. Different states may represent distinct dimensions such as different time periods, different hierarchical instances, or even different domains. Consequently, the individual theories may contain mutually contradictory and overlapping information. The role of DLP is to take into account the mutual relationships extant between different states to precisely determine the declarative and the procedural semantics of the combined theory comprised of all individual theories and the way they relate.

Although DLP can represent several states in one evolving dimension or aspect of a system, no more than one such aspectual evolution can be encoded and combined simultaneously. This is so because DLP is defined only for linear sequences of states. In [15], the states represent different time instants. To overcome this drawback, Leite et al. introduced *Multi-dimensional Dynamic Logic Programming* (MDLP) [12], which generalizes DLP to allow for collections of states organized in arbitrary directed acyclic graphs (DAGs). Within this more general theory, one can encode simultaneously all representational dimensions, which can be particularly useful in the context of multi-agent systems.

In this paper, we formalize agents with such capabilities, generalizing the framework of [15] to allow for an arbitrary number of dimensions represented by an arbitrary DAG. We will show how this new theory is useful for an agent to represent and reason about its own and other agents' knowledge and its evolution in time. Each agent's view of the evolving social relationships among agents (itself and others) is represented by a DAG, itself in turn updatable, to capture the representation of social evolution.

The contribution of the paper is therefore twofold. On the one side, the paper presents an extension of the framework for multi-dimensional dynamic logic programming to incorporate integrity constraints and active rules, and to make the DAG of each agent updatable. On the other, the paper provides a means to incorporate the obtained framework into a multi-agent architecture. For simplicity, we have considered propositional generalized logic programs.

2 Preliminaries

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

Logic Programming Framework In order to represent negative information in logic programs, we need more general logic programs that allow default negation *not* A not only in premises of their clauses but also in their heads[1]. We call such programs generalized logic programs. It is convenient to syntactically represent generalized logic programs as propositional Horn theories. In particular, we represent default negation *not* A as a standard propositional variable.

Propositional variables whose names do not begin with "*not*" and do not contain the symbols ":" and "÷" are called **objective atoms**. Propositional variables of the form *not* A are called **default atoms**. Objective atoms and default atoms are generically called **atoms**.

Propositional variables of the form $i{:}C$ (where C is defined below) are called **projects**. $i{:}C$ denotes the intention (of some agent j) of proposing the updating the theory of agent i with C. Propositional variables of the form $j \div C$ are called **updates**. $j \div C$ denotes an update that has been proposed by j of the current theory of agent (of some agent i) with C. We assume that updates cannot be negated (i.e., we disallow *not* $i{\div}C$). Instead projects can be negated. A negated project *not* $i{:}C$ denotes the intention of the agent not to perform project $i{:}C$. Let $\mathcal{K}$ be a set of propositional variables consisting of objective atoms and projects such that $\mathit{false} \notin \mathcal{K}$. The propositional language $\mathcal{L}_{\mathcal{K}}$, generated by $\mathcal{K}$, consists of the following set of propositional variables: $\mathcal{L}_{\mathcal{K}} \ = \ \mathcal{K} \cup \{\mathit{not}\, A \mid \forall \text{ atom } A \in \mathcal{K}\} \cup \ \{i{\div}C, \mathit{not}\, i{:}C \mid \forall i{:}C \in \mathcal{K}\}$. A **generalized rule** in $\mathcal{L}_{\mathcal{K}}$ is of the form: $L_0 \leftarrow L_1 \wedge \ldots \wedge L_n, \ (n \geq 0)$ where every L_i $(0 \leq i \leq n)$ is an atom from $\mathcal{L}_{\mathcal{K}}$. An **integrity constraint** in $\mathcal{L}_{\mathcal{K}}$ is a rule of the form: $\mathit{false} \leftarrow L_1 \wedge \ldots \wedge L_n \wedge Z_1 \wedge \ldots \wedge Z_m, \ (n \geq 0, m \geq 0)$ where every L_i $(1 \leq i \leq n)$ is an atom, and every Z_j $(1 \leq j \leq m)$ is a project from $\mathcal{L}_{\mathcal{K}}$. Integrity constraints are rules that enforce some condition over the state, and therefore take the form of denials. A **generalized logic program** P in $\mathcal{L}_{\mathcal{K}}$ is a set of generalized rules and integrity constraints in $\mathcal{L}_{\mathcal{K}}$. A **query** Q in $\mathcal{L}_{\mathcal{K}}$ is of the form: $?-\ L_1 \wedge \ldots \wedge L_n, \ (n \geq 1)$ where every L_i $(1 \leq i \leq n)$ is an atom from $\mathcal{L}_{\mathcal{K}}$. The following definition introduces rules that are evaluated bottom-up. To emphasize this aspect, we employ a different notation for them. An **active rule** $\mathcal{L}_{\mathcal{K}}$ is a rule of the form: $L_1 \wedge \ldots \wedge L_n \Rightarrow Z, \ (n \geq 0)$ where every L_i $(1 \leq i \leq n)$ is an atoms, and Z is a project from $\mathcal{L}_{\mathcal{K}}$. Active rules are rules that modify the current state when executed. Active rules take the form: $\mathit{action\_name} \wedge \mathit{Preconditions} \Rightarrow Z$ where *action_name* is an abducible. If the *Preconditions* of the rule are satisfied, then the project (fluent) Z can be selected and executed. The head of an active rule must be a project that is either internal or external. An *internal project* operates on the state of the agent, e.g., if an agent gets an observation, then it updates its knowledge, or if some conditions are met, then it proves some goal, etc. *External projects* instead are performed on the environment, e.g., when an agent sends a message to another agent.

Given a set of vertices V, we assume that for every project $i{:}C$ in $\mathcal{K}$, C is either a generalized rule, an integrity constraint, an active rule, a query or an atom of the form $\mathit{modify\_edge}(j, u, v, x)$, $\mathit{not\, modify\_edge}(j, u, v, x)$, $\mathit{not}\ \ \mathit{edge}(u, v)$, or $\mathit{edge}(u, v)$ where $u, v \in V$. Thus, a project can only take one of the following

[1] For further motivation and intuitive reading of logic programs with default negations in the heads see [1].

form:

$i{:}(L_0 \leftarrow L_1 \wedge \ldots \wedge L_n)$ $\quad$ $i{:}(L_1 \wedge \ldots \wedge L_n \Rightarrow Z)$
$i{:}(\mathit{false} \leftarrow L_1 \wedge \ldots \wedge L_n \wedge Z_1 \wedge \ldots \wedge Z_m)$ $\quad$ $i{:}(?{-}L_1 \wedge \ldots \wedge L_n)$
$i{:}\mathit{modify\_edge}(j, u, v, x)$ $\quad$ $i{:}\mathit{edge}(u, v)$
$i{:}\mathit{not\ modify\_edge}(j, u, v, x)$ $\quad$ $i{:}\ \mathit{not\ edge}(u, v)$

Note that the predicates *edge* and *modify_edge* can only occur inside projects or updates since those predicates do not belong to $\mathcal{L}_\mathcal{K}$. We assume that $i{:}\mathit{edge}(u, v)$ and $i{:}(?{-}L_1, \ldots, L_n)$ can only be internal projects, that is, only the agent itself can issue a project to update its own DAG (i.e., $\mathit{edge}(u, v)$) and goals (i.e., $?{-}L_1, \ldots, L_n$). The project $i{:}\mathit{edge}(u, v)$ issued by the agent i denotes the intention of i to modify its own DAG by establishing an edge between the vertices u and v in V. By issuing a project $i{:}\mathit{modify\_edge}(j, u, v, x)$ the agent i expresses the intention to modify the DAG of another agent j by adding/deleting (depending on whether $x = a$ or $x = d$) an edge between u and v.

3 Agents and Multi-agent Systems

This section presents the notion of agent and multi-agent system. The initial theory of an agent is characterized by a multi-dimensional abductive logic program, inspired by [12], which expresses the agent's viewpoint on the relationships amongst a collection of agents, and encoded in a directed acyclic graph.

Definition 1. Let $\mathcal{L}_\mathcal{K}$ be a propositional language. Let $\mathcal{M}$ be a multi-agent system (defined below) and α an agent in $\mathcal{M}$. A **multi-dimensional abductive logic program** for α, written as Ψ_α, is a tuple $\mathcal{T} = \langle D, \mathcal{P}_D, \mathcal{A}, \mathcal{R}_D \rangle$ where:

– $D = (V, E)$ is an acyclic directed graph, $V = \{v \mid \text{for every } \Psi_v \in \mathcal{M}\} \cup \{\alpha'\}$;
– $\mathcal{P}_D = \{P_v \mid v \in V\}$ is a set of generalized logic programs in the language $\mathcal{L}_\mathcal{K}$, indexed by the vertices $v \in V$ of D;
– $\mathcal{A}$ is a set of atoms;
– $\mathcal{R}_D = \{R_v \mid v \in V\}$ is a set of sets of active rules in the language $\mathcal{L}_\mathcal{K}$, indexed by the vertices $v \in V$ of D.

We call the distinguished vertex $\alpha' \in V$ the **inspection point** of agent α, and we call the atoms in $\mathcal{A}$ **abducibles**.

The initial theory of an agent α is determined (1) by a DAG D that represents the relation between the agents (represented by the vertices in V) in the multi-agent system; (2) by a set of generalized logic programs, one program P_v for each agent $v \in V$; (3) by a set of abducibles; and (4) by a set of sets of active rules R_v, for each agent $v \in V$.

Without loss of generality, we can assume that abducibles have no matching clauses in $P = \bigcup_{v \in V} P_v$. Abducibles can be thought of as hypotheses that can be used to extend the given logic program in order to provide an "explanation", or conditional answer, to a query. Explanations are required to satisfy the integrity constraints in P.

The multi-dimensional abductive logic program formalizes the initial knowledge state of the agent and its reactive behaviour. The knowledge of an agent can dynamically evolve when the agent receives new knowledge, albeit by self-updating rules. This is represented in the form of an updating program.

Definition 2. Let $\mathcal{M}$ be a multi-agent system (defined below). An **updating program** U is a finite set of updates such that if $v \div C \in U$ then $\Psi_v \in \mathcal{M}$.

An updating program contains the updates that will be performed on the current knowledge state of the agent. To characterize the evolution of the knowledge of an agent we need to introduce the notion of sequence of updating programs. Let $S = \{0, 1, \ldots, m\}$ be a set of natural numbers. We call the elements $s \in S$ *states*. A *sequence of updating programs* $\mathcal{U} = \{U^s \mid s \in S \text{ and } s > 0\}$ is a set of updating programs U^s superscripted by the states $s \in S$.

Definition 3. Let $s \in S$ be a state. An **agent** α **at state** s, written as Ψ_α^s, is a pair $(\mathcal{T},\mathcal{U})$, where $\mathcal{T}$ is the multi-dimensional abductive logic program of α, and $\mathcal{U} = \{U^1, \ldots, U^s\}$ is a sequence of updating programs. If $s = 0$, then $\mathcal{U} = \{\}$.

An agent α at state 0 is defined by its initial theory and an empty sequence of updating programs, that is $\Psi_\alpha^0 = (\mathcal{T}, \{\})$. At state 1, α is defined by $(\mathcal{T}, \{U^1\})$, where U^1 is the updating program containing all the updates that α has received at state 1, either from other agents or as self-updates. In general, an agent α at state s is defined by $\Psi_\alpha^s = (\mathcal{T}, \{U^1, \ldots, U^s\})$, where each U^i is the updating program containing the updates that α has received at state i.

Definition 4. A **multi-agent system** $\mathcal{M} = \{\Psi_{v_1}^s, \ldots, \Psi_{v_n}^s\}$ at state s is a set of agents $v_1, \ldots, v_n$ each of which at state s.

Note that the definition of multi-agent system characterizes a static society of agents in the sense that it is not possible to add/remove agents from the system, and all the agents are at one common state.

To begin with, the system starts at state 0 where each agent is defined by $\Psi_\alpha^0 = (\mathcal{T}_\alpha, \{\})$. Suppose that at state 0 an agent β wants to propose an update of the knowledge state of agent α with C by triggering the project $\alpha : C$. Then, at the next common state α will receive the update $\beta \div C$ indicating that an update has been proposed by β. Thus, at state 1, α will be $(\mathcal{T}_\alpha, \{U^1\})$, where $U^1 = \{\beta \div C\}$ if no other updates occur for α.

Within logic programs we refer to agents by using the corresponding subscript. For instance, if we want to express the update of the theory of an agent Ψ_α with C, we write the project $\alpha{:}C$.

The agent's semantics is given by a syntactical transformation (defined in Sect. 3.2), producing a generalized logic program for each agent, and apply to queries for it as well. Nevertheless, to increase readability, we will immediately present a case study example that, though simple, allows one to glean some of the capabilities of the framework. To increase readability, we rely on natural language when explaining some details. The agent cycle, mentioned in the example and defined in Sect. 4, operates on the results of the transformation, but the example can be understood, on a first reading, without requiring detailed knowledge of the transformations.

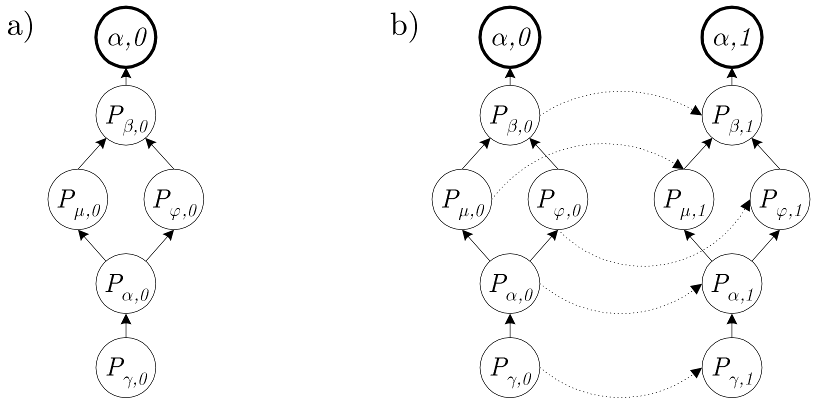

Fig. 1. *DAG of Alfredo at states 0 and 1*

3.1 Case Study

This example shows how the DAG of an agent can be modified through updates, and how its knowledge state and reactions are affected by such modifications. With this aim, we hypothesize two alternative scenarios. Alfredo lives together with his parents, and he has a girlfriend. Being a conservative lad, Alfredo never does anything that contradicts those he considers to be higher authorities, in this case his mother, his father, and the judge who will appear later in the story. Furthermore, he considers the judge to be hierarchically superior to each of his parents. As for the relationship with his girlfriend, he hears her opinions but his own always prevail. Therefore Alfredo's view of the relationships between himself and other agents can be represented by the following DAG $D = (V, E)$, depicted in Fig.1a) (where the inspection point of Alfredo, i.e. the vertex corresponding to Alfredo's overall semantics, is represented by a bold circle): $V = \{\alpha, \beta, \gamma, \mu, \varphi, \alpha'\}$ and $E = \{(\gamma, \alpha), (\alpha, \mu), (\alpha, \varphi), (\mu, \beta), (\varphi, \beta), (\beta, \alpha')\}$ where α is Alfredo, β is the judge, γ is the girlfriend, μ is the mother, φ is the father and α' is the inspection point of Alfredo.

Initially, Alfredo's programs P_{α_0} and R_{α_0} at state 0 contain the rules:

$$
\begin{array}{ll}
r1 : girlfriend \leftarrow & r3 : get\_married \wedge girlfriend \Rightarrow \gamma : propose \\
r4 : not\, happy \Rightarrow \varphi : (?{-}happy) & r2 : move\_out \Rightarrow \alpha : rent\_apartment \\
r5 : not\, happy \Rightarrow \mu : (?{-}happy) & r6 : modify\_edge(\beta, u, v, a) \Rightarrow \alpha : edge(u, v)
\end{array}
$$

stating that: Alfredo has a girlfriend (r1); if he decides to move out, he has to rent an apartment (r2); if he decides to get married, provided he has a girlfriend, he has to propose (r3); if he is not happy he will ask his parents how to be happy (r4, r5); Alfredo must create a new edge between two agents (represented by u and v) in his DAG every time he proves the judge told him so (r6). Alfredo's set of abducibles, corresponding to the actions he can decide to perform to reach his goals, is: $A = \{move\_out, get\_married\}$. In the first scenario, Alfredo's goal is to be happy, this being represented at the agent cycle level (cf. Sect. 4) together with *not false* (to preclude the possibility of violation of integrity constraints) and with the conjunction of active rules: $G = ?{-}(\mathit{happy} \wedge \mathit{not\,false} \wedge r2 \wedge r3 \wedge r4 \wedge r5 \wedge r6)$ During the first cycle (we will assume that there are always enough computational units to complete the proof procedure), the projects $\varphi : (? - \mathit{happy})$ and $\mu : (? - \mathit{happy})$ are selected. Note that these correspond to the only two active rules (r4 and r5) whose premises are verified. Both projects are selected to be performed, producing 2 messages to agents φ and μ (the reader is referred to [8] for details on how communication can be combined into this framework).

In response to Alfredo's request to his parents, Alfredo's mother, as most latin mothers, tells him that if he wants to be happy he should move out, and that he should never move out without getting married. This correspond to the observed updates $\mu \div (\mathit{happy} \leftarrow \mathit{move\_out})$ and $\mu \div (\mathit{false} \leftarrow \mathit{move\_out} \wedge \mathit{not\,get\_married})$ which produce the program P_{μ_1} at state 1 containing the following rules:

$$r7 : \mathit{happy} \leftarrow \mathit{move\_out} \qquad r8 : \mathit{false} \leftarrow \mathit{move\_out} \wedge \mathit{not\,get\_married}$$

Alfredo's father, on the other hand, not very happy with his own marriage and with the fact that he had never lived alone, tells Alfredo that, to be happy he should move out and live by himself, i.e. he will not be happy if he gets married now. This corresponds to the observed updates $\varphi \div (\mathit{happy} \leftarrow \mathit{move\_out})$ and $\varphi \div (\mathit{not\,happy} \leftarrow \mathit{get\_married})$ which produce the program P_{φ_1} at state 1 containing the following rules:

$$r9 : \mathit{happy} \leftarrow \mathit{move\_out} \qquad r10 : \mathit{not\,happy} \leftarrow \mathit{get\_married}$$

The situation after these updates is represented in Fig.1b).

After another cycle, the IFF proof procedure returns no projects because the goal is not reachable without producing a contradiction. From r7 and r8 one could abduce *move_out* to prove *happy* but, in order to satisfy r8, *get_married* would have to be abduced also, producing a contradiction via r10. This is indeed so because, according to the DAG, the rules of P_{φ_1} and P_{μ_1} cannot reject one another other since there is no path from one to the other. Thus, in this scenario, Alfredo is not reactive at all being not able to take any decision.

Consider now this second scenario where the initial theory of Alfredo is the same as in the first, but his parents have decided to get divorced. Suppose that at state 1 Alfredo receives from the judge the update $\beta \div \mathit{modify\_edge}(\beta, \varphi, \mu, a)$ corresponding to the decision to give Alfredo's custody to his mother after his parents divorce. This produces the program P_{β_1} containing the rule:

$$r11 : \mathit{modify\_edge}(\beta, \varphi, \mu, a) \leftarrow$$

After this update, the projects $\varphi : (? - \mathit{happy})$, $\mu : (? - \mathit{happy})$ and $\alpha : \mathit{edge}(\varphi, \mu)$ are selected for execution, which correspond to the evaluation of the active rules r4, r5 and r6. In response to Alfredo's request to his parents, given the precedence of opinion imposed by the judge, suppose that they reply the same rules as in the first scenario, producing the following two programs: P_{μ_2} containing the rules received from the mother:

$$r12 : \mathit{happy} \leftarrow \mathit{move\_out} \qquad r13 : \mathit{false} \leftarrow \mathit{move\_out} \wedge \mathit{not\,get\_married}$$

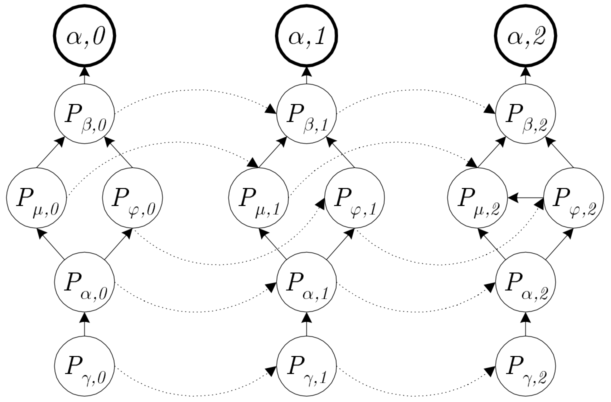

Fig. 2. *DAG of Alfredo at state 2*

and P_{φ_2} containing the rules received from the father:

$$r14 : \mathit{happy} \leftarrow \mathit{move\_out} \qquad r15 : \mathit{not\,happy} \leftarrow \mathit{get\_married}$$

The update $\alpha \div \mathit{edge}(\varphi, \mu)$ produces a change in the graph, and the current situation is the one depicted in Fig.2. Now, the proof procedure returns the projects α : *rent_apartment* and γ : *propose*. These projects correspond to the evaluation of rules r2 and r3 after the abduction of $\{\mathit{move\_out}, \mathit{get\_married}\}$ to prove the goal $(?{-}\mathit{happy})$. Note that now it is possible to achieve this goal without reaching a contradictory state since henceforth the advice from Alfredo's mother (r12 and r13) prevails over and rejects that of his father (r14 and r15). To this second scenario, Alfredo gets married, rents an apartment, moves out with his new wife and lives happily ever after.

3.2 Syntactical Transformation

The semantics of an agent α at state m, Ψ_α^m, is established by a syntactical transformation (called *multi-dimensional dynamic program update* $\bigoplus$) that, given $\Psi_\alpha^m = (\mathcal{T}, \mathcal{U})$, produces an *abductive logic program* [6] $\langle P, \mathcal{A}, R \rangle$, where P is a normal logic program (that is, default atoms can only occur in bodies of rules), $\mathcal{A}$ is a set of abducibles and R is a set of active rules. Default negation can then be removed from the bodies of rules in P via the *dual transformation* defined by Alferes et al [3]. The transformed program P' is a definite logic program. Consequently, a version of the IFF proof procedure proposed by Kowalski et al [10] and simplified by Dell'Acqua et al [6] to take into consideration propositional definite logic programs, can then be applied to $\langle P', \mathcal{A}, R \rangle$.

In the following, to keep notation simple we write rules as $A \leftarrow B \wedge \mathit{not}\, C$, rather than as $A \leftarrow B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn$.

Suppose that $\langle D, \mathcal{P}_D, \mathcal{A}, \mathcal{R}_D \rangle$, with $D = (V, E)$, is the multi-dimensional abductive logic program of agent α, and $S = \{1, \ldots, m\}$ a set of states. Let the vertex $\alpha' \in V$ be the inspection point of α. We will extend the DAG D with two source vertices: an initial vertex sa for atoms and an initial vertex sp for projects. We will then extend D with a set of directed edges (sa_0, s') and (sp_s, s') connecting sa to all the sources s' in D at state 0, and connecting sp to all the sources s' in D at every state $s \in S$. Let $\mathcal{K}$ be an arbitrary set of propositional variables as described above, and $\overline{\mathcal{K}}$ the propositional language extending $\mathcal{K}$ to:

$$\overline{\mathcal{K}} = \mathcal{K} \cup \left\{ \begin{array}{l} A^-, A_{v_s}, A^-_{v_s}, A_{P_{v_s}}, A^-_{P_{v_s}}, reject(A_{v_s}), \\ reject(A^-_{v_s}) \mid A \in \mathcal{K},\ v \in V \cup \{sa, sp\},\ s \in S \end{array} \right\}$$

$$\cup \left\{ \begin{array}{l} rel\_edge(u_s, v_s), rel\_path(u_s, v_s), rel\_vertex(u_s), \\ edge(u_s, v_s), edge(u_s, v_s)^- \mid u, v \in V \cup \{sa, sp\},\ s \in S \end{array} \right\}$$

Transformation of Rules and Updates

(RPR) Rewritten program rules: Let γ be a function defined as follows:

$$\gamma(v, s, F) = \begin{cases} A_{P_{v_s}} \leftarrow B \wedge C^- & \text{if } F \text{ is } A \leftarrow B \wedge not\, C \\ A^-_{P_{v_s}} \leftarrow B \wedge C^- & \text{if } F \text{ is } not\, A \leftarrow B \wedge not\, C \\ false_{P_{v_s}} \leftarrow B \wedge C^- \wedge & \text{if } F \text{ is } false \leftarrow B \wedge not\, C \wedge \\ \qquad\qquad \wedge Z1 \wedge Z2^- & \qquad\qquad \wedge Z1 \wedge not\, Z2 \\ B \wedge C^- \Rightarrow Z_{P_{v_s}} & \text{if } F \text{ is } B \wedge not\, C \Rightarrow Z \\ B \wedge C^- \Rightarrow Z^-_{P_{v_s}} & \text{if } F \text{ is } B \wedge not\, C \Rightarrow not\ Z \end{cases}$$

The rewritten rules are obtained from the original ones by replacing their heads A (respectively, the *not* A) by the atoms $A_{P_{v_s}}$ (respectively, $A^-_{P_{v_s}}$) and by replacing negative premises $not\, C$ by C^-.

(RU) Rewritten updates: Let σ be a function defined as follows:

$$\sigma(s, i{\div}C) = \gamma(i, s, C)$$

where $i{\div}C$ is not one of the cases below. Note that updates of the form $i{\div}(?{-}L_1 \wedge \ldots \wedge L_n)$ are not treated here. They will be treated at the agent cycle level (see Section 4). The following cases characterize the DAG rewritten updates:

$$\begin{array}{l} \sigma(s, i{\div}edge(u, v)) = edge(u_s, v_s) \\ \sigma(s, i{\div}not\ edge(u, v)) = edge(u_s, v_s)^- \\ \sigma(s, i{\div}modify\_edge(j, u, v, x)) = modify\_edge(j, u, v, x) \\ \sigma(s, i{\div}not\ modify\_edge(j, u, v, x)) = modify\_edge(j, u, v, x)^- \end{array}$$

(IR) Inheritance rules:

$$A_{v_s} \leftarrow A_{u_t} \wedge not\, reject(A_{u_t}) \wedge rel\_edge(u_t, v_s)$$
$$A^-_{v_s} \leftarrow A^-_{u_t} \wedge not\, reject(A^-_{u_t}) \wedge rel\_edge(u_t, v_s)$$

for all objective atoms $A \in \mathcal{K}$, for all $u, v \in V \cup \{sa\}$ and for all $s, t \in S$. The inheritance rules state that an atom A is true (resp., false) in a vertex v at state s if it is true (resp., false) in any ancestor vertex u at state t and it is not rejected, i.e., forced to be false.

(RR) Rejection Rules:

$$reject(A^-_{u_s}) \leftarrow A_{P_{v_t}} \wedge rel\_path(u_s, v_t) \qquad reject(A_{u_s}) \leftarrow A^-_{P_{v_t}} \wedge rel\_path(u_s, v_t)$$

for all objective atoms $A \in \mathcal{K}$, for all $u, v \in V \cup \{sa\}$ and for all $s, t \in S$. The rejection rules say that if an atom A is true (respectively, false) in the program P_v at state t, then it rejects inheritance of any false (respectively, true) atoms of any ancestor u at any state s.

(UR) Update Rules:

$$A_{v_s} \leftarrow A_{P_{v_s}} \qquad A^-_{v_s} \leftarrow A^-_{P_{v_s}}$$

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

(DR) Default Rules:

$$A^-_{sa_0}$$

for all objective atoms $A \in \mathcal{K}$. Default rules describe the initial vertex sa for atoms (at state 0) by making all objective atoms initially false.

(CSR$_{v_s}$) Current State Rules:

$$A \leftarrow A_{v_s} \qquad A^- \leftarrow A^-_{v_s}$$

for every objective atoms $A \in \mathcal{K}$. Current state rules specify the current vertex v and state s in which the program is being evaluated and determine the values of the atoms A and A^-.

Transformation of projects

(PPR) Project Propagation rules:

$$Z_{u_s} \wedge not\, rejectP(Z_{u_s}) \wedge rel\_edge(u_s, v_s) \Rightarrow Z_{v_s}$$
$$Z^-_{u_s} \wedge not\, rejectP(Z^-_{u_s}) \wedge rel\_edge(u_s, v_s) \Rightarrow Z^-_{v_s}$$

for all projects $Z \in \mathcal{K}$, for all $u, v \in V \cup \{sp\}$ and for all $s \in S$. The project propagation rules propagate the validity of a project Z through the DAG at a given state s. In contrast to inheritance rules, the validity of projects is not propagated along states.

(PRR) Project Rejection Rules:

$$rejectP(Z^-_{u_s}) \leftarrow Z_{P_{v_s}} \wedge rel\_path(u_s, v_s)$$
$$rejectP(Z_{u_s}) \leftarrow Z^-_{P_{v_s}} \wedge rel\_path(u_s, v_s)$$

for all projects $Z \in \mathcal{K}$, for all $u, v \in V \cup \{sp\}$ and for all $s \in S$. The project rejection rules state that if a project Z is true (respectively, false) in the program P_v at state s, then it rejects propagation of any false (respectively, true) project of any ancestor u at state s.

(PUR) Project Update Rules:

$$Z_{P_{v_s}} \Rightarrow Z_{v_s} \qquad Z^-_{P_{v_s}} \Rightarrow Z^-_{v_s}$$

for all projects $Z \in \mathcal{K}$, for all $v \in V \cup \{sp\}$ and for all $s \in S$. The project update rules declare that a project Z must be true (respectively, false) in v at state s if it is true (respectively, false) in the program P_v at state s.

(PDR) Project Default Rules:

$$Z^-_{sp_s}$$

for all projects $Z \in \mathcal{K}$ and for all $s \in S$. Project default rules describe the initial vertex *sp* for projects at every state $s \in S$ by making all projects initially false.

(PAR$_{v_s}$) Project Adoption Rules:

$$Z_{v_s} \Rightarrow Z$$

for every project $Z \in \mathcal{K}$. These rules, only applying to positive projects, specify the current vertex v and state s in which the project is being evaluated. Projects evaluated to true are selected at the agent cycle level and executed.

Transformation rules for edge

(ER) Edge Rules:

$$edge(u_0, v_0)$$

for all $(u, v) \in E$. Edge rules describe the edges of D at state 0. Plus the following rules that characterize the edges of the initial vertices for atoms (sa) and projects (sp): $edge(sa_0, u_0)$ and $edge(sp_s, u_s)$ for all sources $u \in V$ and for all $s \in S$.

(EIR) Edge Inheritance rules:

$$\begin{aligned} edge(u_{s+1}, v_{s+1}) &\leftarrow edge(u_s, v_s) \wedge not\ edge(u_{s+1}, v_{s+1})^- \\ edge(u_{s+1}, v_{s+1})^- &\leftarrow edge(u_s, v_s)^- \wedge not\ edge(u_{s+1}, v_{s+1}) \\ edge(u_s, u_{s+1}) &\leftarrow \end{aligned}$$

for all $u, v \in V$ such that u is not the inspection point of the agent, and for all $s \in S$. Note that EIR do not apply to edges containing the vertices *sa* and *sp*.

(RER$_{v_s}$) Current State Relevancy Edge Rules:

$$rel\_edge(X, v_s) \leftarrow edge(X, v_s) \quad rel\_edge(X, Y) \leftarrow edge(X, Y) \wedge rel\_path(Y, v_s)$$

Current state relevancy edge rules define which edges are in the relevancy graph wrt. the vertex v at state s.

(RPR) Relevancy Path Rules:

$$\begin{aligned} rel\_path(X, Y) &\leftarrow rel\_edge(X, Y) \\ rel\_path(X, Z) &\leftarrow rel\_edge(X, Y) \wedge rel\_path(Y, Z). \end{aligned}$$

Relevancy path rules define the notion of relevancy path in a graph.

3.3 Multi-dimensional Updates for Agents

This section introduces the notion of multi-dimensional dynamic program update, a transformation that, given an agent $\Psi^m_\alpha = (\mathcal{T}, \mathcal{U})$, produces an abductive logic program (as defined in [6]). Note first that an updating program U for an agent α can contain updates of the form $\alpha \div (?{-}L_1 \wedge \ldots \wedge L_n)$. Such updates are intended to add $L_1 \wedge \ldots \wedge L_n$ to the current query of the agent α. Thus, those updates have to be treated differently. To achieve this, we introduce a function β defined over updating programs. Let U be an updating program and α an agent in $\mathcal{M}$. If U does not contain any update starting with "$\alpha \div (?-$", then, $\beta(\alpha, U) = (\mathit{true}, U)$. Otherwise, suppose that U contains n updates starting with "$\alpha \div (?-$", say $\alpha \div (?{-}C_1), \ldots, \alpha \div (?{-}C_n)$ for n conjunctions $C_1, \ldots, C_n$ of atoms. Then, $\beta(\alpha, U) = (C_1 \wedge \ldots \wedge C_n, U - \{\alpha \div (?{-}C_1), \ldots, \alpha \div (?{-}C_n)\})$. Given a set Q of rules, integrity constraints and active rules, we write Ω_1 and Ω_2 to indicate: $\Omega_1(Q) = \{C \mid \text{for every rule and integrity constraint } C \text{ in } Q\}$ and $\Omega_2(Q) = \{R \mid \text{ for every active rule } R \text{ in } Q\}$. Follows the notion of multi-dimensional dynamic program updates for agents.

Definition 5. Let $m \in S$ be a state and $\Psi_\alpha^m = (\mathcal{T}, \mathcal{U})$ an agent α at state m. Let $\mathcal{T} = \langle D, \mathcal{P}_D, \mathcal{A}, \mathcal{R}_D \rangle$ and $\mathcal{U} = \{U_s \mid s \in S \text{ and } s > 0\}$. The **multi-dimensional dynamic program update for agent** α **at the state** m is $\bigoplus \Psi_\alpha^m = \langle P', \mathcal{A}, R' \rangle$ where: $P' = \bigcup_{v \in V} \gamma(v, 0, P_v) \cup \bigcup_{s \in S}(\Omega_1(Q_s)) \cup \mathit{IR} \cup \mathit{RR} \cup \mathit{UR} \cup \mathit{DR} \cup \mathit{CSR}(\alpha'_m) \cup \mathit{PRR} \cup \mathit{PDR} \cup \mathit{ER} \cup \mathit{EIR} \cup \mathit{RER}(\alpha'_m) \cup \mathit{RPR}$ and $R' = \bigcup_{v \in V} \gamma(v, 0, R_v) \cup \bigcup_{s \in S}(\Omega_2(Q_s)) \cup \mathit{PPR} \cup \mathit{PUR} \cup \mathit{PAR}(\alpha'_m)$, where $\beta(\alpha, U_s) = (\_, P_s)$, and $\sigma(s, P_s) = Q_s$, for every $s \in S$.

Note that P' is a normal logic program, that is, default atoms can only occur in bodies of clauses.

4 The Agent Cycle

In this section we briefly sketch the behaviour of an agent. We omit the details to keep the description general and abstract. Every agent can be thought of as a multi-dimensional abductive logic program equipped with a set of *inputs* represented as *updates*. The abducibles are (names of) *actions* to be executed as well as explanations of observations made. Updates can be used to solve the goals of the agent as well as to trigger new goals. The basic "engine" of the agent is the IFF proof procedure [10,6], executed via the cycle represented in Fig.3. Assume that $\beta(\alpha, U_s) = (L_1 \wedge \ldots \wedge L_n, P_s)$, $\sigma(s, P_s) = Q_s$ and $\Psi_\alpha^{s-1} = (\mathcal{T}, \{U_1, \ldots, U_{s-1}\})$.

The cycle of an agent α starts at state s by observing any inputs (projects from other agents) from the environment (step 1), and by recording them in the form of updates in the updating program U_s. Then, the proof procedure is applied for r units of time (steps 2 and 3) with respect to the abductive logic program $\bigoplus \Psi_\alpha^s$ obtained by updating the current one with the updating program U_s. The new goal is obtained by adding the goals $\mathit{not\,false} \wedge L_1 \wedge \ldots \wedge L_n$ and the active rules in $\Omega_2(Q_s)$ received from U_s to the current goal G. The amount of resources available in steps 2 and 3 is bounded by some amount r. By decreasing r the agent is more *reactive*, by increasing r the agent is more *rational*.

Afterwards (steps 4 and 5), the selectable projects are selected from G', the last formula of the derivation, and executed under the control of an abductive logic programming proof procedure such as ABDUAL in [3]. If the selected

```
Cycle($\alpha$,$r$,$s$,$\Psi_{\alpha}^{s-1}$,$G$)

1. Observe and record any input in the updating program $U_s$.
2. Resume the IFF procedure by propagating the inputs wrt. the
       program $\bigoplus \Psi_{\alpha}^{s}$ and the goal $G \wedge \mathit{not\,false} \wedge L_1 \wedge \ldots \wedge L_n \wedge \Omega_2(Q_s)$.
3. Continue applying the IFF procedure, using for steps 2 and 3 a
       total of $r$ units of time. Let $G'$ be the last formula in this derivation.
4. Select from $G'$ the projects that can be executed.
5. Execute the selected projects.
6. Cycle with ($\alpha$,$r$,$s+1$,$\Psi_{\alpha}^{s}$,$G'$).
```

Fig. 3. *The agent cycle*

project takes the form $j : C$ (meaning that agent α intends to update the theory of agent j with C at state s), then (once executed) the update $\alpha \div C$ will be available (at the next cycle) in the updating program of j. Selected projects can be thought of as outputs into the environment, and observations as inputs from the environment. From every agent's viewpoint, the environment contains all other agents. Every disjunct in the formulae derived from the goal G represents an *intention*, i.e., a (possibly partial) plan executed in stages. A sensible action selection strategy may select actions from the same disjunct (intention) at different iterations of the cycle. Failure of a selected plan is obtained via logical simplification, after having propagated `false` into the selected disjunct.

Integrity constraints provide a mechanism for constraining explanations and plans, and action rules for allowing a condition-action type of behaviour.

5 Conclusions and Future Work

Throughout this paper we have extended the *Multi-dimensional Dynamic Logic Programming* framework to include integrity constraints and active rules, as well as to allow the associated acyclic directed graph to be updatable. We have shown how to express, in a multi-agent system, each agent's viewpoint regarding its place in its perception of others. This is achieved by composing agents' view of one another in acyclic directed graphs, one for each agent, which can evolve over time by updating its edges. Agents themselves have their knowledge expressed by abductive generalized logic programs with integrity constraints and active rules. They are kept active through an observe-think-act cycle, and communicate and evolve by realizing their projects to do so, in the form of updates acting on other agents or themselves. Projects can be knowledge rules, active rules, integrity constraints, or abductive queries.

With respect to future work, we are following two lines of research. At the agent level, we are investigating how to combine logical theories of agents expressed over graph structures [13], and how to express preferences among the beliefs of agents [7]. At the multi-agent system level, we are investigating into the representation and evolution of dynamic societies of agents. A first step towards this goal is to allow the set of vertices of the acyclic directed graph to be also updatable, representing the birth (and death) of the society's agents. Our ongoing work and future projects is discussed in [2].

Acknowledgments We would like to thank José Alferes for all his comments. L. M. Pereira acknowledges the support of POCTI project FLUX. The work of J. A. Leite was also supported by PRAXIS Scholarship no. BD/13514/97.

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinski and T. C. Przymusinski. *Dynamic Updates of Non-Monotonic Knowledge Bases.* Journal of Logic Programming 45(1-3), pages 43-70, 2000.
2. J. J. Alferes, P. Dell'Acqua, E. Lamma, J. A. Leite, L. M. Pereira, and F. Riguzzi. *A logic based approach to multi-agent systems.* ALP Newsletter, 14(3), August 2001.
3. J. J. Alferes, L. M. Pereira and T. Swift. *Well-founded Abduction via Tabled Dual Programs.* In Proc. of ICLP'99. MIT Press. 1999
4. 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*, pages 5-32, Springer 1999.
5. D. De Schreye, M. Hermenegildo and L. M. Pereira, *Paving the Roadmaps: Enabling and Integration Technologies.* Available from `http://www.compulog.org/net/Forum/ Supportdocs.html`
6. P. Dell'Acqua, L. M. Pereira, *Updating Agents.* In S. Rochefort, F. Sadri and F. Toni (eds.), Procs. of the ICLP'99 Workshop MASL'99, 1999.
7. P. Dell'Acqua, L. M. Pereira, *Preferring and Updating with Multi-Agents.* In 9th DDLP-WS. on "Deductive Databases and Knowledge Management", Tokyo, 2001
8. P. Dell'Acqua, F. Sadri, and F. Toni. *Combining introspection and communication with rationality and reactivity in agents.* In *Logics in Artificial Intelligence*, LNCS 1489, pages 17–32, Berlin, 1998. Springer-Verlag.
9. N. Jennings, K. Sycara and M. Wooldridge. *A Roadmap of Agent Research and Development.* In Autonomous Agents and Multi-Agent Systems, 1, Kluwer, 1998.
10. T. H. Fung and R. Kowalski. *The IFF proof procedure for abductive logic programming. J. Logic Programming*, 33(2):151–165, 1997.
11. R. Kowalski and F. Sadri. *Towards a unified agent architecture that combines rationality with reactivity.* In D. Pedreschi and C. Zaniolo (eds), Procs. of LID'96, pages 137-149, Springer-Verlag, LNAI 1154, 1996.
12. J. A. Leite, J. J. Alferes and L. M. Pereira, *Multi-dimensional Dynamic Logic Programming*, In Procs. of CLIMA'00, pages 17-26, July 2000.
13. J. A. Leite, J. J. Alferes and L. M. Pereira, *Multi-dimensional Dynamic Knowledge Representation.* In Procs. of LPNMR-01, pp 365-378, Springer, LNAI 2173, 2001.
14. J. A. Leite and L. M. Pereira. *Generalizing updates: from models to programs.* In Procs. of LPKR'97, pages 224-246, Springer, LNAI 1471, 1998.
15. S. Rochefort, F. Sadri and F. Toni, editors, *Proceedings of the International Workshop on Multi-Agent Systems in Logic Programming*, Las Cruces, New Mexico, USA,1999. Available from `http://www.cs.sfu.ca/conf/MAS99`.
16. I. Niemelä and P. Simons. *Smodels - an implementation of the stable model and well-founded semantics for normal logic programs.* In Procs. of the 4th LPNMR'97, Springer, July 1997.
17. F. Sadri and F. Toni. *Computational Logic and Multiagent Systems: a Roadmap*, 1999. Available from `http://www.compulog.org`.
18. The XSB Group. *The XSB logic programming system, version 2.0*, 1999. Available from `http://www.cs.sunysb.edu/~sbprolog`.