Updating Agents

Pierangelo Dell'Acqua[1], Luís Moniz Pereira[2]

[1] Department of Information Science
Uppsala University
Box 311, 751 05 Uppsala, Sweden
pier@csd.uu.se

[2] Centro de Inteligência Artificial - CENTRIA
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825-114 Caparica, Portugal
lmp@di.fct.unl.pt

1 Introduction

Kowalski and Sadri [9] propose an approach to agents (KS-agents) within an extended logic programming framework. KS-agents are *hybrid* in that they exhibit both *rational* (or *deliberative*) and *reactive* behaviour. The reasoning core of KS-agents is a proof procedure that combines forward and backward reasoning. Backward reasoning is used primarily for planning, problem solving and other deliberative activities. Forward reasoning is used primarily for reactivity to the environment, possibly including other agents. The proof procedure is executed within an observe-think-act cycle that allows the agent to be alert to the environment and react to it as well as think and devise plans. Both the proof procedure and the KS-agent architecture can deal with temporal information. The proof procedure (IFF proof procedure [5]) treats both inputs from the environment and agents' actions as *abducibles* (hypotheses).

Both inputs and agents' actions are treated by the IFF procedure at the goal level of the agent, and the knowledge base of an agent remains unchanged through the iterations of the observe-think-act cycle. Thus, the knowledge of an agent is *static* - it does not evolve with time.

J. J. Alferes et al. [1, 11] present a *dynamic logic programming* framework that allows us to update knowledge bases represented as logic programs. In order to represent *negative* information in logic programs and their updates so that new knowledge may contradict old knowledge, they use generalized logic programs, allowing default negation $not\, A$ not only in the premises of clauses but also in their heads.

Suppose that we are given a set of generalized logic programs, each describing a different state of the world. Different states may represent different time points or different viewpoints. Consequently, logic programs may contain mutually contradictory as well as overlapping information. The role of the dynamic program update is to employ the mutual relationship existing between different programs to determine, at any given program composition stage, the declarative and procedural semantics of the program resulting from the combination of them.

A semantic characterization of updates is given in [1] as a generalization of the stable model semantics of normal logic programs [6]. Such a semantics is generalized to the three-valued case in [2], which enable us to update programs under the well-founded semantics.

In this paper we propose a combination of the dynamic logic programming paradigm and (a version of) KS-agents. In the resulting framework, rational, reactive agents can dynamically change their own knowledge bases as well as their own goals. In particular, at every iteration of an observe-think-act cycle, the agent can make observations, learn new facts and new rules from the environment, and then it can update its knowledge accordingly.

The agent can also receive a piece of information that contrasts with its knowledge. To solve eventual cases of contradiction within the theory of an agent, techniques of contradiction removal and preferences among several sources can be adopted [10].

The actions of an agent are modelled by means of updates, inspired by the approach in [2]. This introduction of updates within logical-based agents opens up a number of research issues to be explored.

2 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. Atoms whose names begin neither with a "*not*" nor with "i:" (for some integer i) nor with "$i\div$" are called **objective atoms**, while the atoms $not\ A$ are called **default atoms**. We represent default negation $not\, A$ as a standard propositional variable (atom). Atoms of the form $i{:}C$ are called **projects**. $i{:}C$ denotes the intention (of some agent j) of updating the theory of agent i with C. Atoms of the form $j \div C$ are called **updates**. $j \div C$ denotes an update (proposed by j) of the current theory of agent i with C. We assume that projects and updates cannot be negated (i.e., we can have neither $not\ i{:}C$ nor $not\ i \div C$) and can only take one of the following forms:

$$
\begin{array}{ll}
i{:}(L_0 \leftarrow L_1 \wedge \ldots \wedge L_n) & j{\div}(L_0 \leftarrow L_1 \wedge \ldots \wedge L_n) \\
i{:}(L_1 \wedge \ldots \wedge L_n \Rightarrow A) & j{\div}(L_1 \wedge \ldots \wedge L_n \Rightarrow A) \\
i{:}(L_1 \wedge \ldots \wedge L_n \Rightarrow \mathit{false}) & j{\div}(L_1 \wedge \ldots \wedge L_n \Rightarrow \mathit{false}) \\
i{:}(?{-}L_1 \wedge \ldots \wedge L_n) & j{\div}(?{-}L_1 \wedge \ldots \wedge L_n).
\end{array}
$$

Objective atoms, default atoms, projects and updates are generically called **atoms**.

[1] For further motivation and intuitive reading of logic programs with default negations in the heads see [7].

- ▶ A **generalized logic program** P is a set of propositional Horn clauses of the form:

$$L_0 \leftarrow L_1 \wedge \ldots \wedge L_n \qquad (n \geq 0)$$

where L_0 is an objective or default atom, and every L_i $(1 \leq i \leq n)$ is a project, or an objective or default atom. If all the atoms appearing in heads of clauses of P are objective atoms, then P is a **normal logic program**. If all the atoms appearing in P are objective atoms, then P is a **definite logic program**.

The following definition introduces rules that are evaluated bottom-up. To emphasize this aspect, we use a different notation for them.

- ▶ Let $L_1, \ldots, L_n$ $(n \geq 1)$ be objective or default atoms. An **integrity constraint** is an implication of the form:

$$L_1 \wedge \ldots \wedge L_n \Rightarrow \mathit{false}$$

and an **action rule** is an implication of the form:

$$L_1 \wedge \ldots \wedge L_n \Rightarrow A$$

where A is a project. Integrity constraints and action rules are called **active rules**.

The absence of *not* A in the conclusion of general integrity constraints is possible only in two-valued semantics. *not* A in the head is not the same as A in the body if A can be undefined. That is, a rule of the form $L_1 \wedge \ldots \wedge L_n \Rightarrow \mathit{not}\, A$ can be expressed in a two-valued semantics as a denial $L_1 \wedge \ldots \wedge L_n \wedge A \Rightarrow \mathit{false}$.

Integrity constraints are rules that enforce some condition over the state, and therefore they take the form of denials. Actions rules are rules that modify the current state when executed. Action rules take the form:

$$\mathit{action\_name} \wedge \mathit{Prec} \Rightarrow A$$

where *action_name* is an abducible. If the preconditions *Prec* of the rule are satisfied, then the project (fluent) A must hold in the state obtained by executing the action.

The head of an action 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.

- ▶ An **abductive logic program** is a tuple $\langle P, \mathcal{A}, R\rangle$, where P is a generalized logic program, $\mathcal{A}$ is a set of object and default atoms in the language of P, and R is a set of active rules. The atoms in $\mathcal{A}$ are referred to as the **abducibles**.

The definition of abductive logic program generalizes the definition given in [8] for allowing default negation in the head of clauses. Without loss of generality, we can assume that abducibles have no matching clauses in P.

Abducibles can be thought of as hypotheses that can be used to extend the given logic program in order to provide an "explanation" for given queries. Explanations are required to "satisfy" the integrity constraints. Different notions of explanation and satisfaction have been used in the literature. The simplest notion of satisfaction, which we employ here, is consistency of the explanation with the program and the integrity constraints.

Example 1. Let Igor be an agent whose underlying theory is characterized by the abductive logic program $\langle P, \mathcal{A}, R\rangle$ given as follows:

$$P = \left\{ \begin{array}{l} \texttt{go\_out} \leftarrow \texttt{go\_concert} \\ \texttt{go\_out} \leftarrow \texttt{go\_movie} \\ \texttt{preferred(j,l)} \end{array} \right\}$$

$$\mathcal{A} = \left\{ \texttt{go\_concert}, \texttt{go\_movie} \right\}$$

$$R = \left\{ \begin{array}{l} \texttt{go\_concert} \wedge \texttt{go\_movie} \Rightarrow \texttt{false} \\ \texttt{go\_concert} \wedge \texttt{good\_concert} \Rightarrow \texttt{t:reserve\_concert} \\ \texttt{go\_movie} \wedge \texttt{not good\_concert} \Rightarrow \texttt{t:reserve\_movie} \end{array} \right\}.$$

The last clause in P states that Jane (represented as j) is preferred to Louise (represented as l). The second active rule in R formalizes the action of going to a concert: if the precondition `good_concert` of the rule is satisfied, then make a reservation for the concert to the ticket agent t.

Assume that Igor wants to go out, that is, he has the goal G = `?-go_out`. Then, the sets of abducibles {`go_concert`} and {`go_movie`} are two explanations for G, satisfying the integrity constraint in R, whereas the set {`go_concert`, `go_movie`} is not, because it is inconsistent with R.

► A **multi-agent system** $\mathcal{M} = \{Ag_1, \ldots, Ag_n\}$ is a set of abductive logic programs $Ag_1, \ldots, Ag_n$.

Within logic programs we refer to agents by using the corresponding number. For instance, if we want to update the theory of agent Ag_i with A, we write $i{:}A$. Thus, we abbreviate Ag_i as i.

► Let $\mathcal{M} = \{Ag_1, \ldots, Ag_n\}$ be a multi-agent system. An **updating program** P is a finite set of updates such that if an update $i{\div}C \in P$ then $i \in \mathcal{M}$.

Suppose that $\{P_s \mid s \in S\}$ is a finite or infinite sequence of generalized logic programs, indexed by the set $S = \{1, 2, \ldots, n, \ldots\}$. We call elements s of the set $\{0\} \cup S$ **states** and we refer to 0 as the **initial state**.

The following definition introduces the notion of sequences of updating programs. Given an abductive logic program P_0 (the initial theory[2] of the agent), a sequence of updating programs is defined as a set consisting of P_0 and a number of updating programs containing the updates to be performed over P_0.

▶ Let $s \in S$ be a state. Suppose that P_0 is an abductive logic program and every P_s, with $s \in S$, is an updating program. Then, $\mathcal{P} = \{P_s \mid s \in \{0\} \cup S\}$ is a **sequence of updating programs**. P_0 is called the **initial program**.

The following simple examples illustrate the notion of sequence of updating programs. More interesting and complex examples that show the full power of updates as actions can be found in [2, 10].

Example 2. Let $\mathcal{M} = \{i, l, j, t\}$ and $S = \{0, 1, 2\}$. Suppose that $P_0 = \langle P, \mathcal{A}, R\rangle$ is the abductive logic program given in example 1. Let the updating programs P_1 and P_2 be defined as:

$$P_1 = \{\}$$

$$P_2 = \left\{ \begin{array}{l} \texttt{j}\div\texttt{not good\_concert} \\ \texttt{l}\div\texttt{good\_concert} \end{array} \right\}.$$

Then, $\mathcal{P} = \{P_0, P_1, P_2\}$ is a sequence of updating programs. Assume that the theory of the agent Igor (represented as i) is characterized by $\mathcal{P}$. At state 1 there are no updates (i.e., $P_1 = \{\}$), while at the second state there are two updates. The first one requires to update the theory of Igor with `not good_concert` and it is required by Jane (i.e., in the theory of Jane the project `i:not good_concert` holds), whereas Louise requires an opposite update.

3 Dynamic Program Updates

In this section we introduce the notion of *dynamic program update* $\bigoplus\{P_s \mid s \in S\}$ over a sequence of updating programs $\mathcal{P} = \{P_s \mid s \in S\}$.

The idea of dynamic updates is simple and quite fundamental. Suppose that we are given a set of program modules P_s, indexed by different states of the world s. Each program P_s contains some knowledge that is supposed to be true at the state s. The individual program modules may contain mutually contradictory as well as overlapping information. The role of the dynamic program update $\bigoplus\{P_s \mid s \in S\}$ is to use the mutual relationships existing between different states (and specified in the form of the ordering relation) to precisely determine, at any given state s, the *declarative* as well as the *procedural* semantics of the combined program, composed of all modules.

Suppose that $\mathcal{K}$ is an arbitrary set of propositional variables. By $\overline{\mathcal{K}}$ we denote the following superset of $\mathcal{K}$:

$$\overline{\mathcal{K}} = \mathcal{K} \cup \{A^-, A_s, A_s^-, A_{i,s}, A_{i,s}^- \mid A \in \mathcal{K},\ i \in \mathcal{M},\ s \in \{0\} \cup S\}.$$

[2] Initial knowledge can be made into permanent background knowledge by including it in every update.

This definition assumes that the original set $\mathcal{K}$ of atoms does not contain any of the newly added symbols of the form $A^-, A_s, A_s^-, A_{i,s}, A_{i,s}^-$ so that they are all disjoint sets of symbols. We denote by $\overline{\mathcal{L}} = \mathcal{L}_{\overline{\mathcal{K}}}$ the extension of the language $\mathcal{L} = \mathcal{L}_{\mathcal{K}}$ generated by $\overline{\mathcal{K}}$.

RPC - rewritten program clauses
Let $s \in \{0\} \cup S$. Let γ be a function defined as follows:

- If C is $A \leftarrow B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn$, then

$$\gamma(s, C) = A_s \leftarrow B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^-.$$

- If C is $\mathit{not}\, A \leftarrow B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn$, then

$$\gamma(s, C) = A_s^- \leftarrow B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^-.$$

- If C is $B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn \Rightarrow \mathit{false}$, then

$$\gamma(s, C) = B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^- \Rightarrow \mathit{false}_s.$$

- If C is $B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn \Rightarrow A$, then

$$\gamma(s, C) = B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^- \Rightarrow A_{s+1}.$$

The rewritten clauses/active rules are simply obtained from the original ones by replacing atoms A (respectively, atoms $\mathit{not}\, A$) occurring in their heads by the atoms A_s (respectively, A_s^-) and by replacing negative premises $\mathit{not}\, C$ by C^-. The project A of an action rule is replaced by the atom A_{s+1} since A is performed in the next state. That is, all the atoms in the body of an action rule are evaluated at the current state and the project A is performed at state $s+1$. Note, crucially, that the atoms in bodies of clauses and active rules are not subscribed with any state and are therefore always evaluated at the current state.

Given a state s and a set U of clauses and active rules, we write $\gamma(s, U)$ to indicate:

$$\gamma(s, U) = \{\gamma(s, C) \mid \text{for every } C \text{ in } U\}.$$

RU - rewritten updates
Let $s \in S$ be a state. Let σ be a function defined as follows:

- If C is $A \leftarrow B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn$, then

$$\sigma(s, i \div C) = A_{i,s} \leftarrow B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^-.$$

- If C is $\mathit{not}\, A \leftarrow B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn$, then

$$\sigma(s, i \div C) = A_{i,s}^- \leftarrow B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^-.$$

– If C is $B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn \Rightarrow \mathit{false}$, then

$$\sigma(s, i \div C) = B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^- \Rightarrow \mathit{false}_{i,s}.$$

– If C is $B1 \wedge \ldots \wedge Bm \wedge \mathit{not}\, C1 \wedge \ldots \wedge \mathit{not}\, Cn \Rightarrow A$, then

$$\sigma(s, i \div C) = B1 \wedge \ldots \wedge Bm \wedge C1^- \wedge \ldots \wedge Cn^- \Rightarrow A_{i,s+1}.$$

The atom $\mathit{false}_{i,s}$ in the third rewritten update means that at state s agent i believes that the knowledge base is inconsistent.

Let P be an updating program. We write $\sigma(s, P)$ to indicate:

$$\sigma(s, P) = \{\sigma(s, i \div C) \mid \text{for every } i \div C \text{ in } P\}.$$

UR - update rules

$$\begin{array}{l} A_s \leftarrow A_{i,s} \wedge \mathit{not}\ \mathit{exception}(A_{i,s}) \\ A_s^- \leftarrow A_{i,s}^- \wedge \mathit{not}\ \mathit{exception}(A_{i,s}^-) \end{array}$$

for all objective atoms $A \in \mathcal{K}$, for all $i \in \mathcal{M}$ and for all $s \in S$. The update rules state that an atom A must be true (respectively, false) in the state $s \in S$ if it is true (respectively, false) in the updating program P_s. The addition of the predicate *exception*, although not strictly needed at this point, allows us to impose later on additional restrictions or preferences on the update rules.

URAR - update rules for active rules

$$A_{i,s} \wedge \mathit{not}\ \mathit{exception\_ar}(A_{i,s}) \Rightarrow A_s$$

for all projects $A \in \mathcal{K}$ or $A = \mathit{false}$, for all $i \in \mathcal{M}$ and for all $s \in S$. The update rules for active rules say that a project A is true in state $s \in S$ if it is true in the updating program P_s.

IR - inheritance rules

$$\begin{array}{l} A_s \leftarrow A_{s-1} \wedge \mathit{not}\ \mathit{reject}(A_{s-1}) \\ A_s^- \leftarrow A_{s-1}^- \wedge \mathit{not}\ \mathit{reject}(A_{s-1}^-) \\ \\ \mathit{reject}(A_{s-1}) \leftarrow A_s^- \\ \mathit{reject}(A_{s-1}^-) \leftarrow A_s \end{array}$$

for all objective atoms $A \in \mathcal{K}$ (with A possibly *false*), for all $i \in \mathcal{M}$ and for all $s \in S$. The inheritance rules say that an atom A is true (respectively, false) in the state $s \in S$ if it is true (respectively, false) in the previous state $s-1$ and it is not *rejected*, i.e., forced to be false (respectively, true) by the updating program.

The addition of the predicate *reject* allows us to impose later on additional restrictions on this inheritance by inertia.

CSR(x) - current state rules

$$A \leftarrow A_x \qquad A^- \leftarrow A_x^-$$

for all objective atoms A in $\mathcal{K}$[3]. Current state rules specify the current state x in which the updated program is being evaluated and determine the values of the atoms A and *not* A[4].

In order to introduce the notion of dynamic program update we need the next definition. Note first that an updating program P for an agent i can contain updates of the form $i \div (?{-}L_1 \wedge \ldots \wedge L_n)$. Such updates are intended to add $L_1 \wedge \ldots \wedge L_n$ to the current query of the agent i. Thus, those updates have to be differently treated. To achieve this, we introduce a function β defined over updating programs.

▶ Let P be an updating program and $i \in \mathcal{M}$ an agent.

- If P does not contain any update starting with "$i \div (?-$", then

$$\beta(i, P) = (\mathit{true}, P).$$

- (Otherwise) Suppose that P contains n updates starting with "$i \div (?-$", say $i \div (?{-}C_1), \ldots, i \div (?{-}C_n)$ for n conjunctions $C_1, \ldots, C_n$ of atoms. Then

$$\beta(i, P) = (C_1 \wedge \ldots \wedge C_n, P - \{i \div (?{-}C_1), \ldots, i \div (?{-}C_n)\}).$$

Given a set Q of clauses and active rules, we write $\Omega_1(Q)$ and $\Omega_2(Q)$ to indicate:

$$\Omega_1(Q) = \{C \mid \text{for every clause } C \text{ in } Q\}$$
$$\Omega_2(Q) = \{R \mid \text{for every active rule } R \text{ in } Q\}.$$

▶ Let $s \in S$ be a state, $i \in \mathcal{M}$ an agent and $\mathcal{P} = \{P_s \mid s \in \{0\} \cup S\}$ a sequence of updating programs. Suppose that the abductive logic program $P_0 \in \mathcal{P}$ is $\langle P, \mathcal{A}, R \rangle$. The **dynamic program update at the state s for agent i**, written as $\bigoplus_s^i \mathcal{P}$, over $\mathcal{P}$ is the tuple which consists of the following clauses in the extended language $\overline{\mathcal{L}}$, for every s belonging to S:

$$\bigoplus\nolimits_s^i \mathcal{P} = \langle P', \mathcal{A}, R' \rangle \text{ with}$$
$$P' = \gamma(0, P) \cup \mathit{UR} \cup \mathit{IR} \cup \mathit{CSR}(\mathit{max}(S)) \cup \bigcup\nolimits_{s \in S}(\Omega_1(Q_s))$$
$$R' = \gamma(0, R) \cup \mathit{URAR} \cup \bigcup\nolimits_{s \in S}(\Omega_2(Q_s))$$

[3] The original definition of CSR(x) contains also the axiom $\mathit{not}\, A \leftarrow A^-$. This axiom is not needed here due to the dynamic update transformation and the dual transformation.

[4] Note that the semantics of agent i can be inconsistent since we allow *false* without subscripts via current state rules. We may specify a semantics for agent i by adapting the semantics of generalized logic programs [1, 2] and by then using techniques for removing contradictions arising from A_x and A_x^-.

where $\beta(i, P_s) = (\ldots, U_s)$ and $\sigma(s, U_s) = Q_s$, for every $s \in S$.

Note that P' is a normal logic program, that is, default atoms can only occur in bodies of clauses. The following example illustrates the use of dynamic program updates.

Example 3. (Conscientious objector – a rule update example)
Consider the situation where someone is conscripted if he is draftable and healthy. Moreover a person is draftable when he has a specific age. However, after some time, the law changes and a person is not conscripted if he is a conscientious objector.

Let $\mathcal{M} = \{a, b, e, l\}$, where e and l play the role of the environment and the law, and $S = \{0, 1, 2, 3\}$. Below we write rules with variables and we assume that they stand for the ground instances wrt. the set $\{a, b\}$. Let $P_0 = \langle P, \{\}, \{\}\rangle$, where:

$$P = \left\{ \begin{array}{l} \texttt{draftable(X)} \leftarrow \texttt{of\_age(X)} \\ \texttt{conscripted(X)} \leftarrow \texttt{draftable(X)} \wedge \texttt{healthy(X)} \end{array} \right\},$$

be the initial knowledge of agent a, and let P_1 be the updating program:

$$P_1 = \left\{ \begin{array}{l} \texttt{e}\div\texttt{healthy(a)} \\ \texttt{e}\div\texttt{conscientious\_objector(a)} \\ \texttt{e}\div\texttt{healthy(b)} \\ \texttt{e}\div\texttt{conscientious\_objector(b)} \\ \texttt{e}\div\texttt{of\_age(b)} \end{array} \right\}.$$

We assume for simplicity that the update rules UR of agent a have no exceptions, i.e., there are no clauses defining the predicate `exception`. Then, in state 1, a believes that b is conscripted and a is not. In fact, `conscripted(b)` holds in $\bigoplus_1^a \mathcal{P}$ (whereas `conscripted(b)` does not), with $\mathcal{P} = \{P_0, P_1\}$.

However, after some time, the law changes and the theory of a is updated with the following two programs:

$$P_2 = \left\{ \texttt{l}\div\texttt{(not conscripted(X)} \leftarrow \texttt{draftable(X)} \wedge \texttt{conscientious\_objector(X))} \right\}$$

$$P_3 = \left\{ \texttt{e}\div\texttt{of\_age(a)} \right\}.$$

Then, after the updating program P_3 the situation of b changes and a believes that b is not conscripted. On the other hand, a remains not conscripted.

4 The Dual Transformation

In this section we sketch a transformation that adjoins to each rule R of a propositional normal logic program a new rule that is true if and only if R is false. We call the resulting program a *dual* program. Such a transformation allows us to remove default negation in the body of clauses of normal logic programs[5]. For a comprehensive overview of this transformation and the corresponding evaluation method see Alferes et al. [3]. Given an objective atom A and an atom $not(B)$, we write $conj_D(A)$ to indicate $not(A)$ and a $conj_D(not(B))$ to indicate B.

▶ Let P be a normal logic program and P' a definite logic program obtained by substituting every occurrence of $notA$ in P with $not(A)$. The **dual transformation** of P creates a dual program, written as $dual(P)$, defined as the union of P' with the smallest program containing the following set S of rules.

- If O is an objective atom for which there are no facts in P, and all of its rules are:

$$\begin{array}{ll} O \leftarrow L_{1,1} \wedge \ldots \wedge L_{1,n_1} & (n_1 \geq 1) \\ \vdots & \\ O \leftarrow L_{m,1} \wedge \ldots \wedge L_{m,n_m} & (n_m \geq 1) \end{array}$$

  with $m \geq 1$, then S contains the rule:

$$not(O) \leftarrow \mathit{fold\_O}_1 \wedge \ldots \wedge \mathit{fold\_O}_m$$

  together with the rules:

$$\begin{array}{c} \mathit{fold\_O}_i \leftarrow conj_D(L_{i,1}) \\ \vdots \\ \mathit{fold\_O}_i \leftarrow conj_D(L_{i,n_i}) \end{array}$$

  where $\mathit{fold\_O}_i$ is assumed not to occur in $\mathcal{L}_P$, for $1 \leq i \leq m$.

- (Otherwise) If O is in $literals(P)$, O is not an abducible and there is no rule with head O in P, then S contains the rule $not(O)$.

In the dual form, if O is not an abducible, then $not(O)$ is true iff O is false in P. For instance, if there is a fact in P for some objective literal O, then the dual has no rule for $not(O)$.

Example 4. Consider the abductive logic program $\langle P, \emptyset, \emptyset \rangle$, in which the set of abducibles and the set of active rules are both empty, and

$$P = \left\{ \begin{array}{l} \mathtt{s} \leftarrow \mathtt{not(p)} \wedge \mathtt{not(q)} \wedge \mathtt{not(r)} \\ \mathtt{p} \leftarrow \mathtt{not(s)} \wedge \mathtt{not(r)} \wedge \mathtt{q} \\ \mathtt{q} \leftarrow \mathtt{not(p)} \wedge \mathtt{r} \\ \mathtt{r} \leftarrow \mathtt{not(q)} \wedge \mathtt{p} \end{array} \right\}$$

The dual program $dual(P)$ of P is shown in Figure 1.

Note that in the dual program, rules can have atoms of the form $not(A)$ in their heads, and a rule for $not(A)$ is designed to be true if and only if A is false[6].

---

[5] Using this technique to remove negation from normal logic programs allows specifying a semantics for agents capturing both the well-founded semantics and stable model semantics [3].

[6] In the ABDUAL proof procedure [3], a literal L is undefined iff neither L nor $not(L)$ are true.

`s ← not(p)∧ not(q)∧ not(r)`	`not(s) ← p`
	`not(s) ← q`
	`not(s) ← r`
`p ← not(s)∧ not(r)∧ q`	`not(p) ← s`
	`not(p) ← r`
	`not(p) ← not(q)`
`q ← not(p)∧ r`	`not(q) ← p`
	`not(q) ← not(r)`
`r ← not(q)∧ p`	`not(r) ← q`
	`not(r) ← not(p)`

Fig. 1. *Dual program for P*

5 The IFF proof procedure

In this section we present a simplified version of the IFF proof procedure proposed by Kowalski et al. [5]. Since we map a generalized logic program into a normal logic program via the dynamic program update transformation, and then into a definite logic program via the dual transformation, the IFF proof procedure is simplified here to take into consideration only propositional definite logic programs.

The IFF procedure presupposes that there is some underlying proof procedure (in this case ABDUAL, but an agent could use several) that can rewrite an iff-definition into another equivalent one. A major issue for building agents is how to control the extent of this autonomous rewriting. It could be by allocating resources or defining the number of computations steps, whatever they are, but one natural solution is as follows, that can be implemented with ABDUAL. The iff-definitions are rewriten by the underlying procedure to the point where all its literals are abducibles. By varying the abducibles used at each stage, the IFF procedure can have control over the extent to which the iff-definitions are rewritten. A literal considered abducible in a previous step, may next be allowed to be rewritten into abducible subgoals. Thus, by varying the abducibles of interest one can control the level of detail and the expansions of the iff-definitions. These definitions can be used as a representation language for expressing ongoing problems on which different problem solving procedures may be brought to bear by the agent. The problem appears similar to the one faced by hybrid theorem provers.

In order to introduce the IFF proof procedure we need the following two definitions.

- ▶ Given an atom A defined in a definite logic program P by the clauses:

$$\begin{array}{c} A \leftarrow D_1 \\ \dots \\ A \leftarrow D_k \end{array} \qquad\qquad (k \geq 1)$$

the **iff-definition** of A is:

$$A \leftrightarrow D_1 \vee \ldots \vee D_k.$$

- ▶ The Σ transformation of a normal logic program P, written as $\Sigma(P)$, is the set of the iff-definitions of all the atoms defined in $dual(P)$.

The IFF proof procedure is a rewriting procedure, consisting of a number of inference rules, each of which replaces a formula by one which is equivalent to it in a theory T of iff-definitions. We assume $T = \Sigma(P)$, for some given abductive logic program $\langle P, \mathcal{A}, R \rangle$. The inference rules are:

1. **Unfolding**: given an atom A and an iff-definition
   $A \leftrightarrow D_1 \vee \ldots \vee D_k$ in T,
   A is replaced by $D_1 \vee \ldots \vee D_k$.
2. **Propagation**: given an abducible atom A and an active rule
   $L_1 \wedge \ldots \wedge L_i \wedge \ldots \wedge L_n \Rightarrow B$ such that $L_i = A$,
   a new active rule $L_1 \wedge \ldots \wedge L_{i-1} \wedge L_{i+1} \wedge \ldots \wedge L_n \Rightarrow B$ is added.
3. **Logical simplification**:
   $[B \vee C] \wedge E$ is replaced by $[B \wedge E] \vee [C \wedge E]$ (**splitting**)
   $B \wedge \texttt{false}$ is replaced by $\texttt{false}$,
   $B \vee \texttt{false}$ is replaced by B, and so on.

- ▶ Let $\langle P, \mathcal{A}, R \rangle$ be an abductive logic program and $T = \Sigma(P)$ a theory of iff-definitions. Given an **initial goal** G (a conjunction of objective atoms), a **derivation** for G (wrt. T) is a sequence of formulae:

$$\begin{array}{c} F_1 = G \wedge R \\ \vdots \\ F_n = N_1 \vee \ldots \vee N_m \end{array}$$

such that each **derived goal** F_{i+1} is obtained from F_i by applying one of the inference rules, as follows:
  - Unfolding is applied to atoms that are either conjuncts in F_i or conjuncts in bodies of active rules in F_i, and to iff-definitions in T.
  - Propagation is applied to abducibles that are conjuncts in F_i and to active rules in F_i.
  - Logical simplification is applied to subexpressions of F_i.

Every derivation relies upon some control strategy. Some strategies are preferable to others, e.g., splitting should always be postponed as long as possible, because it is an explosive operation.

- ▶ Let $F_1 = G \wedge R, \ldots, F_n = N_1 \vee \ldots \vee N_m$ be a derivation for G such that $N_j \neq \texttt{false}$ (for some $1 \leq j \leq m$), N_j is some conjunction of atoms and active rules, and no inference rule can be applied to N_j. Then, $F_1, \ldots, F_n$ is a **successful derivation**. An **answer extracted from** N_j is a set $\mathcal{D}$ of all abducibles that are conjuncts in N_j.

Example 5. Let $\mathcal{P}$ be the sequence of updating programs in example 2. The following is a (successful) derivation for the goal $G = \texttt{?-go\_out}$ wrt. the theory $T = \Sigma(P)$, where $\bigoplus_2^i \mathcal{P} = \langle P, \mathcal{A}, R\rangle$. We assume that the atom `exception` of the update rules is defined by:

$$\texttt{exception(good\_concert}_{l,s}) \leftarrow \texttt{good\_concert}^-_{j,s} \wedge \texttt{preferred(j,l)}.$$

The derivation starts with:

$$F_1 = G \wedge R.$$

By using the rules in CSR(2) and in IR, the rule $\texttt{go\_out}_0 \leftrightarrow \texttt{go\_concert} \vee \texttt{go\_movie}$ in $\Sigma(P)$, by unfolding and logical simplification we obtain:

$$F_2 = [\texttt{go\_concert} \wedge R] \vee [\texttt{go\_movie} \wedge R]$$

and by propagating the active rules in R

$$\begin{aligned} F_3 = {} & [\texttt{go\_concert} \wedge R \wedge (\texttt{go\_movie} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{good\_concert} \Rightarrow \texttt{t:reserve\_concert}_1)] \vee \\ & [\texttt{go\_movie} \wedge R \wedge (\texttt{go\_concert} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{good\_concert}^- \Rightarrow \texttt{t:reserve\_movie}_1)] \end{aligned}$$

Assume that the first disjunct of F_3 is selected, and CSR(2) and unfolding are applied to the action rule by using the update rule:
$\texttt{good\_concert}_2 \leftarrow \texttt{good\_concert}_{l,2} \wedge \texttt{not(exception(good\_concert}_{l,2}))$

$$\begin{aligned} F_4 = {} & [\texttt{go\_concert} \wedge R \wedge (\texttt{go\_movie} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{good\_concert}_{l,2} \wedge \texttt{not(exception(good\_concert}_{l,2})) \Rightarrow \texttt{t:reserve\_concert}_1)] \vee \\ & [\texttt{go\_movie} \wedge R \wedge (\texttt{go\_concert} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{good\_concert}^- \Rightarrow \texttt{t:reserve\_movie}_1)] \end{aligned}$$

Since $\texttt{exception(good\_concert}_{l,2})$ is provable, F_4 is rewritten to:

$$\begin{aligned} F_5 = {} & [\texttt{go\_concert} \wedge R \wedge (\texttt{go\_movie} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{false} \Rightarrow \texttt{t:reserve\_concert}_1)] \vee \\ & [\texttt{go\_movie} \wedge R \wedge (\texttt{go\_concert} \Rightarrow \texttt{false}_0) \wedge \\ & (\texttt{good\_concert}^- \Rightarrow \texttt{t:reserve\_movie}_1)] \end{aligned}$$

Thus, the action rule of the first disjunct cannot be triggered. It is easy to see that by unfolding the atom $\texttt{good\_concert}^-$ in the action rule of the second disjunct, we obtain:

$$F_6 = [\texttt{go\_concert} \wedge R \wedge (\texttt{go\_movie} \Rightarrow \texttt{false}_0) \wedge (\texttt{false} \Rightarrow \texttt{t:reserve\_concert}_1)] \vee$$

$$[\texttt{go\_movie} \wedge R \wedge (\texttt{go\_concert} \Rightarrow \texttt{false}_0) \wedge \texttt{t:reserve\_movie}_1]$$

from which the answer $\mathcal{D} = \{\texttt{go\_movie}\}$ can be extracted and the project $\texttt{t:reserve\_movie}_1$ can be propagated via inheritance rules and executed at the next state (i.e., $s = 3$).

6 The Agent's Cycle

Every agent can be thought of as an 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, executed via the following cycle. Assume that $\beta(i, P_s) = (L_1 \wedge \ldots \wedge L_n, U_s)$, $\sigma(s, U_s) = Q_s$ and $\mathcal{P}' = \mathcal{P} \cup \{P_s\}$.

Cycle(i,r,s,$\mathcal{P}$,G)

1. Observe and record any input in the updating program P_s.
2. Resume the IFF procedure by propagating the inputs wrt. the program $\bigoplus_s^i \mathcal{P}'$ and the goal $G \wedge L_1 \wedge \ldots \wedge L_n \wedge \Omega_2(Q_s)$.
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' a project which can be executed.
5. Execute the selected project.
6. Cycle with (i,r,$s+1$,$\mathcal{P}'$,G').

Fig. 2. *The agent's cycle*

The cycle of an agent i 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 P_s. Then, the proof procedure is applied for r units of time (steps 2 and 3) with respect to the abductive logic program $\bigoplus_s^i \mathcal{P}'$ obtained by updating the current one with the updating program P_s. The new goal is obtained by adding the goals $L_1 \wedge \ldots \wedge L_n$ and the active rules in $\Omega_2(Q_s)$ received from P_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.* Propagation is applied first (step 2) in order to allow for an appropriate reaction to the inputs.

Afterwards (steps 4 and 5), a project is selected from G', the last formula of the derivation, and executed. Only projects with state tag s can be selected. If the selected project takes the form $j : C_s$ (meaning that agent i intends to update the theory of agent j with C at state s), then (once executed) the update $i \div C$ 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.

7 Conclusions

We have presented 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. In a multi-agent system, agents can interact with each other by means of action rules which allow one to express the intention to update the theory of another agent. In turn, the receiving agent will decide whether or not to update his own knowledge (i.e., its current abductive logic program $\bigoplus_s^i \mathcal{P}$) depending on the requiring agent. Preferences among agents can also be formalized in order to remove possible contradictory information coming in from different agents.

The proposed approach results from the combination (of a version) of the approach to agents in [9] (KS-agents) and the approach to updating logic programs in [1, 11]. The contribution of the paper is twofold. On one side, the paper presents an extension of the framework for updating logic programs to cope with multiple agents and to treat active rules as well. On the other side, the paper proposes a starting point for an agent architecture more elaborate than the one of KS-agents. For simplicity, we have considered propositional generalized logic programs and we have not considered time within the agent's cycle.

Even though our approach to agents resembles KS-agents, they differ in a number of ways. First, we use generalized logic programs (rather than normal logic programs as in KS-agents) to express negative information, and we distinguish integrity constraints and action rules. The main difference between the two approaches regards the agent's cycle. In Kowalski and Sadri's approach, the theory of an agent is static: it remains fixed through iterations of the cycle. In particular, observations and agents' actions are treated as abducibles. In contrast, our agents can update their knowledge bases and active rules (both integrity constraints and action rules) as well as their own goals at every iteration of the cycle.

Our approach is a first attempt to characterize a kind of rational, reactive and updating agent. The approach can be extended in several ways. For example, the agent cycle's can be further elaborated in order to have a better interrelationship between the reactive and the rational part of the cycle. We may want to build agents that are aware of this interrelationship, and are therefore able to decide when to be reactive and when to be rational. We may also give the agents the ability to update the updating mechanism itself (rather than just to define it at the cycle level). Furthermore, the language of the agents can be extended to take into consideration the language of (fully) generalized logic programs rather than propositional ones. To this aim, we have to equip our agent with some form of meta-level capability in such a way that the agents can reason about their own beliefs as well as beliefs of other agents and that they can communicate with each other along the lines proposed, for example, in [4]. In addition, we are considering the possibility to incorporate other rational abilities (e.g., learning) in our approach to agents and to introduce preferences among agents and meta-preferences to empower the choice ability of agents.

We are working to an implementation of updating agents based on a system for ABDUAL and dynamic program updates. The later system, running under XSB-Prolog, is available at http://centria.di.fct.unl.pt/~jja.

Acknowledgements

We thank J. J. Alferes and J. A. Leite for their helpful comments on previous versions of this paper, and acknowledge the support of PRAXIS project MENTAL, and of a PRAXIS scholarship to P. Dell'Acqua.

References

1. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic logic programming. In A. Cohn and L. Schubert, editors, *KR'98*, pages 98–109. Morgan Kaufmann, 1998. Extended version forthcoming in the J. of Logic Programming (in press).
2. J. J. Alferes, L. M. Pereira, H. Przymusinska, T. C. Przymusinski, and P. Quaresma. Preliminary exploration on actions as updates. Proc. of Joint Conf. on Declarative Programming (APPIA-GULP-PRODE'99).
3. J. J. Alferes, L. M. Pereira, and T. Swift. Well-founded Abduction via Tabled Dual Programs. To appear in: ICLP'99.
4. P. Dell'Acqua, F. Sadri, and F. Toni. Combining introspection and communication with rationality and reactivity in agents. In J. Dix, F.L. Del Cerro, and U. Furbach, editors, *Logics in Artificial Intelligence*, LNCS 1489, pages 17–32, Berlin, 1998. Springer-Verlag.
5. T. H. Fung and R. Kowalski. The IFF proof procedure for abductive logic programming. *J. Logic Programming*, 33(2):151–165, 1997.

6. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *ICLP'88*, pages 1070–1080. MIT Press, 1988.
7. M. Gelfond and V. Lifschitz. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *KR'92*, pages 603–614. Morgan-Kaufmann, 1992.
8. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. Gabbay, C. Hogger, and A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
9. R. A. Kowalski and F. Sadri. Towards a unified agent architecture that combines rationality with reactivity. In D. Pedreschi and C. Zaniolo, editors, *Logic in Databases, Intl. Workshop LID'96*, LNCS 1154, pages 137–149, Berlin, 1996. Springer-Verlag.
10. E. Lamma, F. Riguzzi, and L. M. Pereira. Agents learning in a three-valued logical setting. In A. Panayiotopoulos, editor, *Workshop on Machine Learning and Intelligent Agents, in conjunction with Machine Learning and Applications, Advanced Course on Artificial Intelligence (ACAI'99)*, Chania (Greece), 1999. (Also available at http://centria.di.fct.unl.pt/~lmp/).
11. J. A. Leite and L. M. Pereira. Iterated logic program updates. In J. Jaffar, editor, *Proc. Joint Intl. Conf. Symp. on Logic Programming 1998*, pages 265–278, Cambridge, Mass., 1998. MIT Press.