Preliminary exploration on actions as updates

José Júlio Alferes
Univ. Évora, and CENTRIA
R. Romão Ramalho, 59
P-7000 Évora, Portugal
jja@dmat.uevora.pt

Luís Moniz Pereira
Centro de Inteligência Artificial (CENTRIA)
FCT, Univ. Nova de Lisboa
P-2825-114 Caparica, Portugal
lmp@di.fct.pt

Halina Przymusinska
Dept. Computer Science
California State Polytechnic Univ.
Pomona, CA 91768, USA
halina@cs.ucr.edu

Teodor C. Przymusinski
Dept. Computer Science
Univ. of California
Riverside, CA 92521, USA
teodor@cs.ucr.edu

Paulo Quaresma
Univ. Évora, and CENTRIA
R. Romão Ramalho, 59
P-7000 Évora, Portugal
pq@dmat.uevora.pt

Abstract

This paper reports on the use of logic program updates to models actions that bring about changes in knowledge states describing the world and its rules. The LUPS language is employed to specify concurrent update rules to model actions that depend on, and change, rule rich world states, not described simply by propositional fluents. This language and corresponding semantics, whose original stable semantics we generalize to the well-founded setting, is briefly recapitulated here.

The advantages of our approach to modelling actions as updates are brought out by a number of telling examples, especially conceived for the purpose. They have been test run in our implementation of LUPS, though the latter is not described here for lack of space.

1 Introduction

Traditionally, Logic Programming (LP) has allowed us to declaratively specify and query the semantics of non-monotonic Knowledge Base (KB) states. Of late, LP has been pressed into the service of also declaratively specifying KB updates. The updating of one successive KB by another is expressable through a given updating

sequence of generalized LPs acting on an initial KB state [1]. Generalized LPs are normal programs that allow for default negation in the head of rules too, so as to cater for negating a conclusion, and thus possibly updating it.

Most recently, LUPS, a language for updating LPs (which itself translates into LP), has been deployed for describing the rule content of such updating programs. In LUPS, the specification of the actual update rules to be made part of an update program may depend on the semantics of the KB state that is being updated.

In this setting, is natural to explore how actions, linking preconditions in one state to postconditions in the next, might be newly depicted as sets of update rules of an update program. The advantages of doing so are that:

- Actions can act upon and produce KB states, and these are more generally expressed by sets of rules, not just sets of propositional fluents. Consequently, rules themselves can be updated, providing for improved expressivity and naturalness;
- Several actions may be enacted simultaneously, via a single update program;
- The resulting KBs are just generalized LPs that have been implemented by preprocessing into normal LPs;
- Other non-monotonic mechanisms can be combined with actions within the larger LP setting;
- A precise semantics is provided by the LP theoretical setting.

In this paper, we start by recalling the LUPS language of updates, and the semantics of the generalized LPs which it manipulates. The semantics of LUPS, briefly reviewed here, can itself be defined by translating update programs into sequences of generalized LPs. The meaning of the sequences of generalized LPs (called dynamic LPs) is given in [1], under a two-valued stable semantics.

Next in this paper, our concern with defining and implementing LUPS in a three-valued well-founded setting leads us to define a three-valued well-founded semantics (WFS) for generalized LPs. This allows us, namely, to obtain object language level models of sets of actions with undefined outcome, to rely on top-down query procedures, and to plug onto existing efficient implementations of the WFS. Though our preprocessor implementation into normal LPs (running with the XSB system) is not described here for paucity of space[1], the concrete presented examples have all been test run with success.

The final part of the paper proffers seven illustrative examples of how the modelling of actions via LP updates is achieved in LUPS in a natural and expressive way. Some of the examples are based on classical ones, and all are designed to bring out the advantages of the approach. This paper is a preliminary exploration of the subject, as much remains to be done, e.g. in the way of characterizing the class of actions being captured, on comparing LUPS with extant actions languages [7], and on combining actions as updates with abduction to achieve planning. Some forays

[1] The implementation is available at `http://centria.di.fct.unl.pt/~jja/lups.p`

into these topics are well under way by ourselves and others. We thought it useful and find it exciting to report back on the landmarks of our incursion so far.

2 The language LUPS

In this section we briefly review the *"language for updating logic programs"*, LUPS, defined in [2]. As explained in the introduction, LUPS is a declarative language to represent changes in knowledge bases represented by logic programs.

In the LUPS framework, knowledge evolves from one state to another as a result of sets of (simultaneous) update commands. In order to represent *negative* information in logic programs and in their updates, the framework resorts to more general logic programs, those allowing default negation *not* A not just in the premises of their rules but in their heads as well. In [1], such programs are dubbed *generalized logic programs*, and their semantics is defined as an extension of the stable model semantics of normal logic programs [5] to this broader class of programs[2]. To facilitate the generalization of this semantics to a 3-valued setting (presented in section 3 below), the usual definition of the stable model semantics of generalized logic programs, presented below, is different from the original one in [1], but their equivalence can easily be shown, given the results in [3].

Definition 1 (Generalized Logic Program) *A* generalized logic program P *in the language* $\mathcal{L}$ *is a (possibly infinite) set of propositional rules of the form:*

$$L \leftarrow L_1, \ldots, L_n \tag{1}$$

where L *and* L_i *are literals. A literal is either an atom* A *or its default negation not* A*. Literals of the form not* A *are called default literals.*

If none of the literals appearing in heads of rules of P *are default literals, then the logic program* P *is* normal*.*

In order to define the semantics of generalized programs, we start by eliminating all default literals in heads.

Definition 2 *Let* $\overline{\mathcal{L}}$ *be the language obtained from the language* $\mathcal{L}$ *of a generalized logic program* P *by adding for each propositional symbol* A *the new symbol* $\overline{A}$*.* $\overline{P}$ *is the normal program obtained from the generalized program* P *through replacing every negative head not* A *by* $\overline{A}$*.*

The definition of the stable models of generalized programs can now be obtained from the stable models of the program $\overline{P}$. The idea is quite simple: since $\overline{P}$ is a normal program, its stable models can be obtained by the usual Gelfond-Lifschitz definition [5]; afterwards, all it remains to do is to interpret the $\overline{A}$ atoms in the stable models of $\overline{P}$ as the default negation of A. Since atoms of the form $\overline{A}$ never appear in the body of rules of $\overline{P}$, this task is quite simple: if $\overline{A}$ is true in a stable

[2]The class of generalized logic programs can be viewed as a special case of a yet broader class of programs introduced earlier in [6].

model then *not* A must also be true in it (i.e. A cannot belong to the stable model); if $\overline{A}$ is false in a stable model, then no rule in P imposes *not* A, and so the valuation of A in the stable model is independent of the existence of rules for $\overline{A}$.

Definition 3 (Stable models of generalized logic programs) *Let P be a generalized logic program, and let I be a stable model of $\overline{P}$ (i.e. I be such that $I = \Gamma_{\overline{P}} I$) such that for no atom A both A and $\overline{A}$ belong to I. The model M, obtained from I by deleting from it all atoms of the form $\overline{A}$, is a stable model of P.*

As proven in [1], the class of stable models of generalized logic programs extends the class of stable models of normal programs [5] in the sense that, for the special case of normal programs both semantics coincide.

In LUPS, knowledge states, each represented by a generalized logic program, evolve due to sets of update commands. By definition, and without loss of generality [2], the initial knowledge state KS_0 is empty and, in it, all predicates are assumed false by default. Given a current knowledge state KS_i, its successor state is produced as a result of the occurence of a non-empty set U of simultaneous update commands. Thus, any knowledge state is solely determined by the sequence of sets of updates commands performed from the initial state onwards. Accordingly, each non-initial state can be denoted by:

$$KS_n = U_1 \otimes \ldots \otimes U_n \qquad n > 0$$

where each U_i is a set of update commands.

Update commands specify assertions or retractions to the current knowledge state (i.e. the one resulting from the last update performed). In LUPS a simple assertion is represented as the command:

$$assert\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{2}$$

Intuitively, its meaning is that if the precondition $L_{k+1}, \ldots, L_m$ is true in the current state, then the rule $L \leftarrow L_1, \ldots, L_k$ is added to its successor state, and persists by inertia, until possibly retracted or overridden by some future update command.

In order to represent rules and facts that do not persist by inertia, i.e. that are one-state events, LUPS includes the modified form of assertion:

$$assert\ event\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{3}$$

The retraction of rules is performed with the update command:

$$retract\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{4}$$

Its meaning is that, subject to precondition $L_{k+1}, \ldots, L_m$ (verified at the current state) rule $L \leftarrow L_1, \ldots, L_k$ is either retracted from its successor state onwards, or just temporarily retracted in the successor state (if governed by *event*).

Normally assertions represent newly incoming information. Although its effects remain by inertia (until countervened or retracted), the assert command itself does

not persist. However, some update commands may desirably persist in the successive consecutive updates. This is especially the case of laws which, subject to some preconditions, are always valid, or of rules describing the effects of an action. In the former case, the update command must be added to all sets of updates, to guarantee that the rule remains indeed valid. In the latter case, the specification of the effects must be added to all sets of updates, to guarantee that, when the action takes place, its effects are enforced.

To specify such persistent update commands, LUPS introduces the commands:

$$always\ [event]\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{5}$$

$$cancel\ (L \leftarrow L_1, \ldots, L_k)\ when\ (L_{k+1}, \ldots, L_m) \tag{6}$$

The first states that, from the current state onwards, in addition to any newly arriving set of commands, whenever the preconditions are verified, the persistent rule is added too. The second command cancels this persistent update.

Definition 4 (LUPS language[2]) *An update program in LUPS is a finite sequence of updates:*

$$U_1 \otimes \ldots \otimes U_n$$

each update U_i being a non-empty set of (simultaneous) commands of the forms (2)-(5).

Any knowledge state KB_q $(0 \leq q \leq n)$ resulting from an update program $U_1 \otimes \ldots \otimes U_n$ can be queried via:

$$holds(L_1, \ldots, L_m)\ at\ q?$$

The query is true iff the conjunction of its literals holds at KB_q.

The semantics of LUPS [2] is defined by translating update programs into sequences of generalized logic programs. The meaning of such sequences of programs (called dynamic logic programs) is determined by the semantics defined in [1]. Alternatively, [2] also defines the semantics of update programs as the stable models of a generalized logic program (written in a meta-language) obtained from the update program. Here we simply recap the first alternative.

A dynamic program [1] is a sequence $P_0 \oplus \ldots \oplus P_n$ (also denoted $\bigoplus \mathcal{P}$, where $\mathcal{P}$ is a set of generalized logic programs indexed by $1, \ldots, n$ and $P_0 = \{\}$). The notion of *"Dynamic program update at a given state s"*, represented by $\bigoplus_s \mathcal{P}$, precisely characterizes a generalized logic program whose stable models correspond to the meaning of the dynamic program when queried at state s. If some literal or conjunction of literals ϕ holds in all stable models of $\bigoplus_s \mathcal{P}$, we write $\bigoplus_s \mathcal{P} \models_{sm} \phi$.

The translation into a dynamic program is made by induction, starting from the empty program P_0, and for each update U_i, given the already built dynamic program $P_0 \oplus \ldots \oplus P_{i-1}$, determining the resulting program $P_0 \oplus \ldots \oplus P_{i-1} \oplus P_i$. To cope with persistent update commands we will further consider, associated with every dynamic program in the inductive construction, a set containing all currently active persistent commands, i.e. all those that were not cancelled until that point

in the construction, from the time they were introduced. To be able to retract rules, we need to uniquely identify each such rule. This is achieved by augmenting the language of the resulting dynamic program with a new propositional variable "$rule(L \leftarrow L_1, \ldots, L_n)$" for every rule $L \leftarrow L_1, \ldots, L_n$ appearing in the original LUPS program[3].

Definition 5 (Translation into dynamic programs) *Let $\mathcal{U} = U_1 \otimes \ldots \otimes U_n$ be an update program. The corresponding dynamic program $\Upsilon(\mathcal{U}) = \mathcal{P} = P_0 \oplus \ldots \oplus P_n$ is obtained by the following inductive construction, using at each step i an auxiliary set of persistent commands PC_i:*

Base step: *$P_0 = \{\}$ with $PC_0 = \{\}$.*

Inductive step: *Let $\mathcal{P}_i = P_0 \oplus \ldots \oplus P_i$ with set of persistent commands PC_i be the translation of $\mathcal{U}_i = U_1 \otimes \ldots \otimes U_i$. The translation of $\mathcal{U}_{i+1} = U_1 \otimes \ldots \otimes U_{i+1}$ is $\mathcal{P}_{i+1} = P_0 \oplus \ldots \oplus P_{i+1}$ with set of persistent commands PC_{i+1}, where PC_{i+1} is:*

$$
\begin{array}{r}
PC_i \cup \{assert\ [event]\ (R)\ when\ (C) : always\ [event]\ (R)\ when\ (C) \in U_{i+1}\} \\
-\{assert\ [event]\ (R)\ when\ (C) : cancel\ (R)\ when\ (D) \in U_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} D\} \\
-\{assert\ [event]\ (R)\ when\ (C) : retract\ (R)\ when\ (D) \in U_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} D\}
\end{array}
$$

$NU_{i+1} = U_{i+1} \cup PC_{i+1}$, and P_{i+1} is:

$$
\begin{array}{rl}
 & \{R,\ rule(R) : assert\ [event]\ (R)\ when\ (C) \in NU_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} C\} \\
\cup & \{not\, rule(R) : retract\ [event]\ (R)\ when\ (C) \in NU_{i+1} \wedge \bigoplus_i \mathcal{P}_i \models_{sm} C\} \\
\cup & \{not\, rule(R) : assert\ event\ (R)\ when\ (C) \in NU_i \wedge \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C\} \\
\cup & \{rule(R) : retract\ event\ (R)\ when\ (C) \in NU_i \wedge \bigoplus_{i-1} \mathcal{P}_{i-1} \models_{sm} C, rule(R)\}
\end{array}
$$

where R denotes a generalized logic program rule, and C and D a conjunction of literals. In the inductive step, if $i = 0$ the last two lines are ommitted. In that case NU_i does not exist.

Definition 6 (LUPS semantics) *Let $\mathcal{U}$ be an update program. A query*

$$holds(L_1, \ldots, L_n)\ at\ q$$

is true in $\mathcal{U}$ iff $\bigoplus_q \Upsilon(\mathcal{U}) \models_{sm} L_1, \ldots, L_n$.

From the results on dynamic programs in [1], it is clear that LUPS generalizes the language of updates of "revision programs" defined in [9].

[3]Note that, by definition, all such rules are ground and thus the new variable uniquely identifies the rule, where rule/1 is a reserved predicate.

3 Generalization to the 3-valued case

In order to implement LUPS and run our examples, as explained in the introduction, we first generalize the semantics of update programs to a 3-valued setting. Since the semantics may be defined in terms of a translation into a single generalized logic program, whose semantics is the (2-valued) stable models one, all that needs doing is to generalize the semantics of logic programs with default negation in the heads to a 3-valued setting. The resulting update program semantics is established on the well-founded semantics [4] instead of on stable models.

In fact, like in the alternative semantics definition for update programs, the whole point is to provide a meaning to sequences of update commands by means of successive program transformations (that reflect the changes due to commands). The meaning is then obtained by applying the stable models semantics to the successive resulting generalized logic programs. In order to assign a different meaning, based on another object level semantics, all is required is to apply another semantics to the obtained programs. However, it is essential that this other semantics be a semantics for generalized programs (not the case of the well-founded semantics as defined by [4], which applies to normal programs only). So, below we generalize the well-founded semantics to the class of generalized logic programs, via an alternating fixpoint operator, based on the Gelfond-Lifschitz Γ-operator.

A naïve generalization of the well-founded semantics would simply be to consider the fixpoints of the compound operator Γ^2 for the transformed programs $\overline{P}$, and then removing all fixpoints where, for some atom, both A and $\overline{A}$ held. In fact, for normal programs, the least fixpoint of Γ^2 (where Γ is the Gelfond-Lifschitz operator) provides the well-founded semantics. However this naïve definition does not engender intuitive results:

Example 1 *Consider the generalized program P:*

$$a \leftarrow not\, b \qquad\qquad b \leftarrow not\, a \qquad\qquad not\, a \leftarrow$$

According to the naïve semantics, the well-founded model would be $\{not\, a\}$. In this case, since $not\, a$ is true, one would expect b to be true as well.

In the definition of stable models for generalized programs, whenever an atom $\overline{A}$ is true in some interpretation I (in the extended language $\overline{\mathcal{L}}$), and since by definition $A \notin I$, it is guaranteed that when applying the Γ-operator all occurences of $not\, A$ are removed from rule bodies. In other words, whenever $\overline{A}$ is true, A is assumed false by default in rule bodies. In the well-founded semantics, one has to make sure that, whenever $\overline{A}$ belongs to a fixpoint of Γ^2, all literals $not\, A$ in the bodies must be ensured true. In other words, whenever $\overline{A}$ belongs to a fixpoint I of Γ^2, A must not belong to $\Gamma(I)$. This is easily guaranteed by using a semi-normal version of the program:

Definition 7 (Semi-normal program) *The semi-normal version $\overline{Ps}$ of a normal program $\overline{P}$ is obtained by adding to the body of each rule in $\overline{P}$ with head A (resp. $\overline{A}$) the literal $not\, \overline{A}$ (resp. $not\, A$).*

Definition 8 (Partial stable models of generalized programs) *Let I be a set of atoms in the language $\overline{\mathcal{L}}$ such that:*

1. $I = \Gamma_{\overline{P}}(\Gamma_{\overline{Ps}}(I))$
2. $I \subseteq \Gamma_{\overline{Ps}}(I)$

Then the 3-valued model $M = T \cup not\, F$ is a partial stable model of the generalized program P, where $not\,\{A_1, \ldots, A_n\}$ stands for $\{not\, A_1, \ldots, not\, A_n\}$, and:

- *T is obtained from I by deleting all atoms of the form $\overline{A}$;*
- *F is the set of all atoms A that do not belong to $\Gamma_{\overline{Ps}}(I)$;*

With this definition there is no need to explicitly discard interpretations comprising both A and $\overline{A}$ for some atom A. These are already filtered by condition 2. Indeed, if both A and $\overline{A}$ belong to I then, because in $\overline{Ps}$ all rules with head A (resp. $\overline{A}$) have $not\, \overline{A}$ (resp. $not\, A$) in the body, neither A nor $\overline{A}$ belong to $\Gamma_{\overline{Ps}}(I)$, and thus condition 2 fails to hold.

Definition 9 (Well-founded model of generalized programs) *The well founded model of a generalized program P is the set-inclusion least partial stable model of P. The well-founded model can be obtained by iterating the (compound) operator $\Gamma_{\overline{P}}\Gamma_{\overline{Ps}}$ starting from empty, and constructing M from the so obtained least fixpoint.*

Example 2 *The well-founded model of the program in example 1 is $\{b, not\, a\}$. In fact: $\Gamma_{\overline{Ps}}(\{\}) = \{a, b, \overline{a}\}$, $\Gamma_{\overline{P}}(\{a, b, \overline{a}\}) = \{\overline{a}\}$, $\Gamma_{\overline{Ps}}(\{\overline{a}\}) = \{b, \overline{a}\}$, $\Gamma_{\overline{P}}(\{b, \overline{a}\}) = \{b, \overline{a}\}$. Accordingly, by definition, its well-founded model is $\{b, not\, a\}$. Note in the 3rd application of the operator above, how the semi-normality of the program is instrumental to guarantee the truth of b.*

4 Examples

In this section we muster a variety of examples of application of LUPS to the representation of actions. In fact actions can be described by persistent updates (laws) which relate the state where the actions happened with the successor state where the consequences hold. In the examples we are assuming all persistent commands not explicitly linked to a knowledge state to belong to U_1. Moreover whenever is referred a knowledge state U_i which was not defined previously it is assumed to be equal to $assert\ (true)$.

4.1 Twice fined

A car-driver looses his licence after a second fine. He can regain the licence if he goes for a refresh course at the drivers school.

In order to represent this problem it is necessary to introduce a new predicate, "*probation*", which holds after the first fine takes place. The $fine$ action is represented by two persistent update commands:

$$always\ (probation)\ when\ (fined)$$
$$always\ (not\ licence)\ when\ (fined,\ probation).$$

The first rule means *probation* becomes true after a fine and the second rule means the driver looses his licence after a fine if he is on probation.

The *attend_school* action is represented by the two persistent update commands:

$$always\ (licence)\ when\ (attend_school).$$
$$always\ (not\ probation)\ when\ (attend_school).$$

These rules mean that after attending school the driver has a licence and he is not in a probation state.

Imagine the sequence of non-persistent update commands:

$$\begin{aligned} U_1 &: assert\ event\ (attend_school) \\ U_2 &: assert\ event\ (fine) \\ U_3 &: assert\ event\ (fine) \\ U_4 &: assert\ event\ (attend_school) \end{aligned}$$

As intended, *licence* holds in state $1, 2, 4$ and *not licence* holds in state 3 (after two fines).

4.2 Timers

In this example we represent timers using LUPS. Timers are started by the action $on_for(N)$ which means the timer will be on for the next N states. This notion can be captured by the three persistent rules:

$$always\ event\ (on_for(M))\ when\ (on_for(N), N > 0, minus(N, 1, M))$$

This rule states the timer is on for M states if it was on for $N = M + 1$ states in the previous state. $minus(A, B, C)$ is a auxiliary predicate meaning that C is equal to A minus B.

Timers are *on* if they are $on_for(N)$, where $N > 0$:

$$always\ (on\ \leftarrow\ on_for(N), N > 0)$$

and switched off when $N = 0$

$$always\ (not\ on\ \leftarrow\ on_for(0))$$

Consider the sequence of updates:

$$U_1\ :\ assert\ event\ (on_for(2))$$

As intended, *on* holds in state $1, 2$ and *not on* holds in state 3.

4.3 Suitcase

There is a suitcase, with two latches, which opens whenever both latches are up, and there is an action of toggling applicable to each latch [8]. This situation is represented by the three persistent commands:

$$always\ (open \leftarrow up(l1), up(l2))$$
$$always\ (up(L))\ when\ (not\ up(L), toggle(L))$$
$$always\ (not\ up(L))\ when\ (up(L), toggle(L))$$

In the initial situation $l1$ is down, $l2$ is up, and the suitcase is closed:

$$U_1 = \{assert\ (not\ up(l1)), assert\ (up(l2)), assert\ (not\ open)\}$$

Suppose there are now two simultaneous toggling actions:

$$U_2 = \{assert\ event\ (toggle(l1)), assert\ event\ (toggle(l2))\}$$

and afterwards there is still another $l2$ toggling action:

$$U_3 = \{assert\ event\ (toggle(l2))\}$$

In the knowledge state 2 we'll have $up(l1), not\ up(l2)$ and the suitcase is not open. Only after U_3 will latch $l2$ be up and the suitcase open.

4.4 Lift

A simple lift allowing simultaneous requests will be modeled.

The basic action is the *push* button one which starts a *request*:

$$always\ (request(X))\ when\ (push(X))$$

The request holds until it is satisfied by a non-inertial *go* action:

$$always\ event\ (go(F))\ when\ (request(F), preferred(F))$$
$$always\ (not\ request(F) \leftarrow go(F))$$

The first rule means that a non-inertial *go* action to a specific floor is performed when there is a request for that floor and the floor is the preferred one to go next. Moreover, the *go* action cancels the *request*.

The preferred predicate singles out the preferred floor from requested ones:

$$always\ (preferred(F) \leftarrow request(F), not\ better_request(F)).$$
$$always\ (better_request(F) \leftarrow request(F1), better(F1, F)).$$

These rules mean that a floor is the preferred one if there aren't better requested floors. We are assuming the existence of a partial order between the floors.

The $go(X)$ action causes the lift to be at the intended floor at the next state:

$$always\ (at(F))\ when\ (go(F))$$
$$always\ (not\ at(X))\ when\ (go(F), diff(X,F))$$

We conjure the following situation (assuming floor 1 is better than floor 2 which is better than floor 3):

$$\begin{aligned} U_1 \ &: \ assert\ (at(1)). \\ &\quad\ assert\ event\ (push(2)) \\ &\quad\ assert\ event\ (push(3)) \\ U_2 \ &: \ assert\ event\ (push(1)) \end{aligned}$$

At knowledge state 1 there are two pending requests: $request(2)$ and $request(3)$. Request 2 is the preferred one and is served at the state 3: $go(2)$. At state 2 there are also two requests: $request(3)$ and $request(1)$. As $request(1)$ is the preferred one, it is first served: $go(1)$. Finally, $request(3)$ is acted on: $go(3)$.

4.5 Dialogue

LUPS can be used to model dialogue interchange between agents. First, it is necessary to describe the different speech acts and some behaviour rules, namely how the information is transferred between the agents [10].

In this example we exhibit a simple dialogue where an agent informs another about a property P contradictory with his previous beliefs. We assume cooperative agents.

This rule represents the $inform$ speech act:

$$always\ (bel(H, bel(S,P)))\ when\ (inform(S,H,P))$$

It means that, after an inform speech act, the hearer starts to believe the speaker believes in the so informed proposition.

Next we describe how information is transferred between agents:

$$always\ (bel(A,P))\ when\ (bel(A, bel(B,P)))$$
$$always\ (not\ bel(A,Q) \leftarrow incompatible(P,Q))\ when\ (bel(A,Q), bel(A, bel(B,P)))$$

These rules mean that a cooperative agent always accepts the transmitted information and cancels his previous beliefs if they are incompatible with the new ones. For instance, it is incompatible for an agent to be at two different places simultaneously:

$$always\ (incompatible(at(P1,A), at(P2,A)) \leftarrow P1 \neq P2)$$

Suppose in state 1 agent a believes agent b is at the hospital:

$$U_1 : assert\ (bel(a, at(hospital, b)))$$

But agent c informs a that b is at home:

$$U_2 : assert\ event\ (inform(c,a, at(home,b)))$$

In state 2 $bel(a, at(home, b))$ will hold as a consequence of the inform speech act and the information transference rule. $bel(a, at(hospital, b))$ will not hold, assuming incompatibility between $at(home, X)$ and $at(hospital, X)$.

As shown, with this approach it is possible to model the acquisition of information from other agents in a dialogue situation.

4.6 Contradictory advice – 3-valued updating

This example models a situation where an agent receives advice from two sources: father and mother. The agent's expected behaviour is to perform an action unless it is contradicted by other advice.

The following persistent update commands can be used to model the desired behavior (for simplicity, we are assuming fathers usually give positive advice and mothers negative advice):

$$\begin{aligned} always\ (do(A) &\leftarrow father_advises(A), not\ dont(A)) \\ always\ (dont(A) &\leftarrow mother_advises(noA), not\ do(A)) \end{aligned}$$

Suppose the father advises buying stocks and the mother advises not to do so:

$$U_1 = \{assert\ event\ (father_advises(buy)), assert\ event\ (mother_advises(no\ buy))\}$$

In this situation $do(buy)$ and $do(no\ buy)$ become undefined and the agent does not perform any action.

4.7 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.

$$\begin{aligned} U_1 &: always\ (draftable(X))\ when\ (of_age(X)) \\ &\quad assert\ (conscripted(X) \leftarrow draftable(X), healthy(X)) \\ U_2 &: assert\ (healthy(a)). \\ &\quad assert\ (healthy(b)). \\ &\quad assert\ (of_age(b)). \\ &\quad assert\ (conscientious_objector(a)). \\ &\quad assert\ (conscientious_objector(b)). \\ U_4 &: assert\ (of_age(a)). \\ U_5 &: assert\ (not\ conscripted(X) \leftarrow conscientious_objector(X)) \end{aligned}$$

In state 3, b is conscripted but after the U_5 assertion his situation changes and not conscripted takes hold. On the other hand, a is never conscripted.

Acknowledgements

This work was partially supported by PRAXIS XXI project MENTAL, by JNICT project ACROPOLE, and a NATO scholarship while L. M. Pereira was on sabbatical leave at the Department of Computer Science, University of California, Riverside.

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.

[2] J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS – a language for updating logic programs. Technical report, June 1999.

[3] C. V. Damásio and L. M. Pereira. Default negated conclusions: why not ? In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proc. of the 5th International Workshop on Extensions of Logic Programming (ELP'96)*, number 1050 in LNAI, pages 103–117, 1996.

[4] A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[5] 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.

[6] 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.

[7] M. Gelfond and V. Lifschitz. Action languages. *Linkoping Electronic Articles in Computer and Information Science*, 3(16), 1998.

[8] F. Lin. Embracing causality in specifying the indirect effects of actions. In *IJCAI'95*, pages 1985–1991. Morgan Kaufmann, 1995.

[9] V. Marek and M. Truszczynski. Revision specifications by means of programs. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *JELIA '94*, volume 838 of *LNAI*, pages 122–136. Springer-Verlag, 1994.

[10] L. M. Pereira and P. Quaresma. Modelling agent interaction in logic programming. In Osamu Yoshie, editor, *INAP'98 - The 11th International Conference on Applications of Prolog*, pages 150–156, Tokyo, Japan, September 1998. Science University of Tokyo.