I N F S Y S

R E S E A R C H

R E P O R T

INSTITUT FÜR INFORMATIONSSYSTEME

ABTEILUNG WISSENSBASIERTE SYSTEME

DECLARATIVE UPDATE POLICIES FOR NONMONOTONIC KNOWLEDGE BASES

Thomas EITER   Giuliana SABBATINI
Michael FINK   Hans TOMPITS

INFSYS RESEARCH REPORT 1843-02-07

MAY 2002

Institut für Informationssysteme
Abtg. Wissensbasierte Systeme
Technische Universität Wien
Favoritenstraße 9-11
A-1040 Wien, Austria
Tel: +43-1-58801-18405
Fax: +43-1-58801-18493
sek@kr.tuwien.ac.at
www.kr.tuwien.ac.at

TECHNISCHE UNIVERSITÄT WIEN

INFSYS RESEARCH REPORT
INFSYS RESEARCH REPORT 1843-02-07, MAY 2002

DECLARATIVE UPDATE POLICIES FOR NONMONOTONIC KNOWLEDGE BASES

Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits[1]

Abstract. Updating databases, and in particular relational databases, is a central issue which has been well-studied in the database field for many years, and solutions have been incorporated into commercial database systems. As for more advanced databases such as logical databases, and in particular for nonmonotonic knowledge bases, work on this problem is more recent. Various approaches for incorporating a single or a sequence of changes into a nonmonotonic knowledge base have been proposed. However, most of these approaches were concerned with *how* a change should be accomplished at a technical level, but did not address the issue of strategic update behavior, i.e., *which* change should be issued in reaction to new information. In this paper, we present a generic framework for declarative specification of update policies, which addresses this issue. In this framework, the update behavior of an agent maintaining its knowledge base is described in a rule-based language, whose statements describe change actions being dependent on the information received, the currents set of beliefs, and possible other change actions. In a layered approach, update policies are evaluated and compiled to update semantics for sequences of nonmonotonic logic programs using generic components. The framework can be instantiated to obtain different update mechanisms, and thus provides flexible support for the development of adaptive reasoning agents.

---

[1]Institut für Informationssysteme, Abteilung Wissensbasierte Systeme 184/3, Technische Universität Wien, Favoritenstraße 9-11, A-1040 Wien, Austria. E-mail: [eiter, michael, giuliana, tompits]@kr.tuwien.ac.at

Acknowledgements: This work was supported by the Austrian Science Fund (FWF) projects P13871-INF and Z29-INF.

Some of the results in this paper appeared, in preliminary form, in *Proc. 17th Int'l Joint Conference on Artificial Intelligence (IJCAI-01)*, Morgan Kaufmann, 2001, and in *Proc. Symposium on Adaptive Agents and Multi-Agent Systems (AISB'01)*, The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (AISB).

Copyright © 2002 by the authors

Contents

1 Introduction

Since the eighties of the last century, the problem of updating logical data and knowledge bases has been studied by many researchers, and a number of different approaches have been developed (cf. [18, 33] and [16] for accounts of early methods). While the early approaches were devised for classical monotonic knowledge bases, methods for updating nonmonotonic knowledge bases, and in particular for nonmonotonic logic programs, have been proposed only much later, cf. [1, 2, 11, 15, 35, 30, 24].

These methods were concerned with *how to incorporate* a change request, given by a rule or a set of rules, or a sequence of update requests into the knowledge base, and proposed solutions based on different grounds. Only more recently, the issue of *how to specify change requests* for knowledge bases has received growing attention. Here, one is interested in *update specifications*, that is, in formalisms for describing the way in which a knowledge base has to be adjusted in reaction to incoming information in a dynamic and flexible way, based on change primitives. Update requests should be specified in a designated policy language, which provides flexibility by taking information from the environment and the current knowledge base into account. We note that similar approaches have been developed in the area of network management and security [26, 10, 9, 6].

For nonmonotonic knowledge bases represented as (nonmonotonic) logic programs, the LUPS language of dynamic updates [3, 4] has been proposed as a general and flexible language for expressing different kinds of operations for changing the knowledge base, such as adding or removing a clause from it. In LUPS, a sequence $U_1, \ldots, U_n$ of sets U_i of update statements describes how a so-called *dynamic logic program* $P_1, \ldots, P_n$, i.e., a sequence of logic programs P_i expressing subsequent updates [2], is formed step by step. An important feature which is missing in LUPS but indispensable for the usage of nonmonotonic knowledge bases to model the behavior of dynamic agents, is the possibility of modeling flexible reaction to *events* of the environment, making update actions depend on the content of rules or facts which are, in discrete way, communicated by the environment, e.g., by some cooperating agent or by some sensing device.

In this paper, we present a generic framework for the specification of declarative update policies and a language called EPI, which builds upon LUPS and provides a number of novel aspects: the possibility of conditioning update actions over external events, over other update commands in the same policy, and over the current belief set of the agent. Advantages of this approach are, besides its flexibility and expressivity, on the one hand the generality, because of its use of a declarative language which is neither tailored on a specific application domain nor ad hoc for special purposes, and, on the other hand, its modularity, which allows to instantiate the components of the framework in different ways, provided that some conditions are satisfied. Moreover, as we shall see, the EPI language can be easily implemented in the agent platform IMPACT [32]; however, it is by no means platform dependent. Realization in IMPACT opens the way to powerful extensions of the language, in that deontic modalities, available in IMPACT can enrich the decision making of the agent and the expressivity of update policies.

The main contributions of our work can be briefly summarized as follows.

(1) We present a *generic* framework for specifying update policies, which can be instantiated with different underlying update semantics for (sequences of) logic programs. This is realized through a *layered approach*: At the top level, the update policy is evaluated, given an event and the current belief set of the agent, to single out the set U of update commands which are to be performed on the current knowledge base. At the next layer, U is compiled to a set P of rules to be incorporated into the knowledge base, encoding the intended change. At the bottom level, the updated knowledge base is represented as a sequence of logic programs, whose semantics is given by a parametric semantics for such programs and determines the current belief set from the knowledge base.

(2) We define a declarative language for update policies, which is inspired by LUPS. The language offers several additional features which, in our opinion, can be successfully used for modeling dynamic agents. First of all, access to incoming information, which is viewed as event, is provided and can be used for triggering the execution of change actions. For example, the statement

$$\mathbf{retract}(best\_buy(shop_1))[[\mathbf{E} : best\_buy(shop_2)]]$$

expresses that if $best\_buy(shop_2)$ is told, then $best\_buy(shop_1)$ is removed from the knowledge base. Statements may also involve further and more complex conditions on the current belief set, and on other commands in the same update step. The language thus enables a flexible handling of events, such as simply recording changes in the environment, skipping uninteresting updates, or applying default actions. We also discuss some extensions of the language, which do not require new semantic definitions and which could be of use in practical applications, providing more expressive conditions through access to rule structure and means to record the past.

(3) We analyze some properties of the framework, based on properties of its basic components. In particular, we formulate some structural properties which are satisfied by the update answer set semantics of [11] and by dynamic logic programs from [2], as well as by possible similar approaches. Moreover, we identify an *expressive core* of the language EPI, by showing that a subset of EPI (and of LUPS) primitives provides the expressivity of the whole language. Furthermore, we derive complexity results for elementary reasoning tasks in the framework. While unsurprisingly reasoning is intractable in general, due to intractability of nonmonotonic logic programs, we also identify a class of tractable cases.

(4) We describe a possible realization of the framework in the agent platform IMPACT [32], providing evidence that our approach can serve as a viable tool supporting the development of adaptive reasoning agents. We remark at the same time that the features of the framework are by no means platform dependent; on the contrary, the use of a powerful platform like IMPACT would allow to make use of additional features, enriching the policy language with deontic modalities.

The remainder of this paper is structured as follows. The next section contains some preliminaries and fixes notation. Our framework for update policies is then presented in Section 3. After that, we study in Section 4 some properties of the framework, where we consider structural as well as computational aspects. Section 5 briefly describes an implementation of the framework in the context of the IMPACT multi-agent platform [32]. Possible extensions of the EPI language are discussed in Section 6, which is followed by a comparison to LUPS and other related work, in particular to $\mathcal{PDL}$ [26]. The final Section 8 outlines some issues for further research. Some of the results appeared in preliminary form in [12, 13].

2 Preliminaries

We consider knowledge bases represented by extended logic programs under the answer set semantics. An *extended logic program* (ELP) is a finite set P of rules r of form

$$L_0 \leftarrow L_1, \ldots, L_m, not\ L_{m+1}, \ldots, not\ L_n,$$

where each L_i is a literal, i.e., either an atom A or a strongly negated atom $\neg A$, and not denotes *weak* (or *default*) *negation*. The literal L_0 is the *head* of r and is denoted by $H(r)$. In case that L_0 is absent, r

is called a *constraint*. The set $B(r) = \{L_1, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n\}$ is the *body* of r. We define $B^+(r) = \{L_1, \ldots, L_m\}$ and $B^-(r) = \{L_{m+1}, \ldots, L_n\}$. We call r a *fact* if $B(r) = \emptyset$ and $H(r)$ is present.

The complement of a literal L is A if $L = \neg A$ and $\neg A$ if $L = A$; the complement of L is denoted by $\neg L$. For any set S of literals, we denote $\neg S = \{\neg L \mid L \in S\}$. In particular, $Lit_{\mathcal{A}} = \mathcal{A} \cup \neg\mathcal{A}$ is the set of all literals over $\mathcal{A}$. By $\mathcal{L}_{\mathcal{A}}$ we denote the set of all rules constructible from the literals in $Lit_{\mathcal{A}}$. The notion of a *stratified logic program* is defined as usual (view $\mathcal{A} \cup \neg\mathcal{A}$ as atoms and add constraints $\leftarrow A, \neg A$ for all $A \in \mathcal{A}$).

An *update program* is a sequence $\boldsymbol{P} = (P_1, \ldots, P_n)$ of ELPs P_i, where $n \geq 1$, expressing successive updates to the original knowledge base P_1. We adopt an abstract view of the semantics of ELPs and update programs, given as a mapping Bel, which associates with every sequence $\boldsymbol{P}$ a set $Bel(\boldsymbol{P}) \subseteq \mathcal{L}_{\mathcal{A}}$ of rules; intuitively, $Bel(\boldsymbol{P})$ are the consequences of $\boldsymbol{P}$. Different instantiations of Bel are possible, according to various proposals for update semantics in the literature (e.g., [2, 11, 35, 30, 27]). We only assume that Bel has some elementary properties which these or any other "reasonable" semantics satisfy. In particular, the following properties must hold:

1. *Priority of updates:* $P_n \subseteq Bel(\boldsymbol{P})$ for $\boldsymbol{P} = (P_1, \ldots, P_n)$, as it is common in most knowledge-base update approaches (e.g., [2, 11, 30], and cf. [15, 22]).

2. *Invariance w.r.t. satisfied literals:* given $A \leftarrow\ \in Bel(\boldsymbol{P})$ and $A \in B(r)$, then $r \in Bel(\boldsymbol{P})$ iff $H(r) \leftarrow B(r) \setminus \{A\} \in Bel(\boldsymbol{P})$.

As usual, the semantics of programs P and update sequences $\boldsymbol{P}$ with variables is defined in terms of their grounded versions $\mathcal{G}(P)$ and $\mathcal{G}(\boldsymbol{P})$ over the Herbrand universe, respectively. We use in the following for illustration the answer set semantics for update programs from [11], which coincides with the semantics of inheritance programs in [8] over the common fragment.

An *interpretation* is any set $I \subseteq \mathcal{A}$ which contains no complementary pair of literals. A literal L is *true* in I if $L \in I$. As well, $not\, L$ is true in I if $L \notin I$. A set S of literals or weakly negated literals is true in I if any element in S is true in I. A rule r is true in I if either $H(r) \in I$ or some element from $B(r)$ is not true in I. A program P is true in I if any $r \in P$ is true in I (I is called *model* of P). We write $I \models \alpha$ to express that α is true in I, where α is one of the objects above.

An interpretation $S \subseteq Lit_{\mathcal{A}}$ is a (consistent) *answer set* of an ELP P [20] iff it is a minimal model of the *reduct*

$$P^S = \{H(r) \leftarrow B^+(r) \mid r \in P \text{ and } B^-(r) \cap S = \emptyset\}.$$

By $\mathcal{AS}(P)$ we denote the collection of all answer sets of P. Following [11], answer sets for an update program $\boldsymbol{P} = (P_1, \ldots, P_n)$ are defined in terms of the answers sets of a single ELP as follows. The *rejection set*, $Rej(S, \boldsymbol{P})$, of $\boldsymbol{P}$ with respect to the interpretation S is given by $Rej(S, \boldsymbol{P}) = \bigcup_{i=1}^{n} Rej_i(S, \boldsymbol{P})$, where $Rej_n(S, \boldsymbol{P}) = \emptyset$, and, for $n > i \geq 1$, $Rej_i(S, \boldsymbol{P})$ contains every rule $r \in P_i$ such that $H(r') = \neg H(r)$ and $S \models B(r) \cup B(r')$, for some $r' \in P_j \setminus Rej_j(S, \boldsymbol{P})$ with $j > i$. That is, $Rej(S, \boldsymbol{P})$ contains the rules from $\boldsymbol{P}$ which are rejected on the basis of unrejected rules from later updates. Then, $S \subseteq Lit_{\mathcal{A}}$ is an *answer set* of $\boldsymbol{P} = (P_1, \ldots, P_n)$ iff S is an answer set of the program $P = \bigcup_i P_i \setminus Rej(S, \boldsymbol{P})$. We denote the set of all answer sets of $\boldsymbol{P}$ by $\mathcal{AS}(\boldsymbol{P})$. Since $n = 1$ implies $Rej(S, \boldsymbol{P}) = \emptyset$, the semantics properly extends the usual answer set semantics for single programs. A syntactic transformation of $\boldsymbol{P}$ into an ELP realizing this semantics is given in [11].

Example 1 *Let $P_0 = \{b \leftarrow not\, a,\ a \leftarrow\ \}$, $P_1 = \{\neg a \leftarrow, c \leftarrow\ \}$, and $P_2 = \{\neg c \leftarrow\ \}$. Then, P_0 has the single answer set $S_0 = \{a\}$ with $Rej(S_0, P_0) = \emptyset$; the sequence (P_0, P_1) has the answer set $S_1 = \{\neg a, c, b\}$*

$Bel(KS_{i-1})$		Belief set at step $i-1$
$KB \quad E_1 \quad \ldots \quad E_{i-1}$	E_i	Knowledge state KS_i
$\Downarrow \mathcal{U}$		*update policy*
$KB \quad U_1 \quad \ldots \quad U_{i-1}$	U_i	Executable commands
$\Downarrow tr$		*compilation*
$P_0 \quad P_1 \quad \ldots \quad P_{i-1}$	P_i	Update sequence
$\Downarrow Bel$		*update semantics*
$Bel(KS_i) = Bel(P_0, \ldots, P_i)$		Belief set at step i

Figure 1: From knowledge state to belief set at step i.

with $Rej(S_1, (P_0, P_1)) = \{a \leftarrow\}$*; and* (P_0, P_1, P_2) *possesses* $S_2 = \{\neg a, \neg c, b\}$ *as unique answer set with* $Rej(S_2, (P_0, P_1, P_2)) = \{c \leftarrow, a \leftarrow\}$.

The *belief set* $Bel_{\mathcal{A}}(\boldsymbol{P})$ is the set of all rules entailed under cautious answer set semantics, i.e., $Bel_{\mathcal{A}}(\boldsymbol{P}) = \{r \in \mathcal{L}_{\mathcal{A}} \mid S \models r \text{ for all } S \in \mathcal{AS}(\boldsymbol{P})\}$. We shall drop the subscript "$\mathcal{A}$" if no ambiguity can arise. With a slight abuse of notation, for a literal L, we write $L \in Bel_{\mathcal{A}}(\boldsymbol{P})$ if $L \leftarrow \in Bel_{\mathcal{A}}(\boldsymbol{P})$. For a sequence $(P_1, \ldots, P_n)$ we simply write $Bel(P_1, \ldots, P_n)$ instead of $Bel((P_1, \ldots, P_n))$.

3 Framework and Language

We first describe our generic framework for event-based updating, and afterwards the EPI language ("the language around") for specifying *update policies*.

3.1 General framework

We start with the formal notions of an *event* and of a *knowledge state* of an agent.

Definition 1 *An* event class *is a collection* $\mathcal{EC} \subseteq 2^{\mathcal{L}_{\mathcal{A}}}$ *of finite sets of rules. The members* $E \in \mathcal{EC}$ *are called* events.

Informally, $\mathcal{EC}$ describes the possible events an agent may witness. For example, the collection $\mathcal{F}$ of all sets of facts from a subset $\mathcal{A}' \subseteq \mathcal{A}$ of atoms may be an event class, e.g., current date, weather, or sale information. More complex events in form of logic program rules can also be thought of, e.g., new learned rules, current exchange rates, special offers depending on user profiles, and so on. In what follows, we assume that an event class $\mathcal{EC}$ has been fixed.

Definition 2 *A* knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ *consists of an ELP* KB (*the* initial knowledge base) *and a sequence* $E_1, \ldots, E_n$ *of events* $E_i \in \mathcal{EC}$, $i \in \{1, \ldots, n\}$. *For* $i \geq 0$, $KS_i = \langle KB; E_1, \ldots, E_i \rangle$ *is the projection of* KS *to the first* i *events.*

Intuitively, KS describes a concrete evolution of the agent's knowledge, starting from its initial knowledge base. When a new event E_i occurs, the current knowledge state $KS_{i-1} = \langle KB; E_1, \ldots, E_{i-1} \rangle$ changes to $KS_i = \langle KB; E_1, \ldots, E_{i-1}, E_i \rangle$, which requests the agent to incorporate the event into its knowledge base and adapt its belief set.

The modular procedure for adapting the belief set $Bel(KS_{i-1})$ upon arrival of E_i is illustrated in Figure 1. Informally, at Step i of the knowledge evolution, given the current belief set $Bel(KS_{i-1})$ and the knowledge state $KS_{i-1} = \langle KB; E_1, \ldots, E_{i-1} \rangle$, together with the new event E_i, the new belief set $Bel(KS_i)$ is computed by applying the update policy $\mathcal{U}$ as follows. First, a set U_i of *executable commands*, which have to be actually applied in the current situation, is determined from $\mathcal{U}$. This set depends on the current knowledge state, its belief set, and on the incoming events. Afterwards, given the previously computed sets $U_1, \ldots, U_{i-1}$, the sequence $(KB; U_1, \ldots, U_i)$ is compiled by the transformation tr into the update sequence $\boldsymbol{P} = (P_0, P_1, \ldots, P_i)$. Finally, $Bel(KS_i)$ is given by $Bel(\boldsymbol{P})$.

3.2 Syntax

The language EPI builds upon the update specification language LUPS [3], by allowing update statements to depend on other such statements in the same EPI program, and more complex conditions on both the current belief set and the actual event (note that LUPS has no notion of an external event, which may be crucial for modeling reactive behavior of a system exposed to unforeseen updates).

The syntax of EPI is given in Table 1. In what follows, we use cmd to denote update commands and ρ for rules or rule variables. In general, an EPI statement is of form

$$cmd_1(\rho_1) \textbf{ if } [\textbf{not}]cmd_2(\rho_2), \ldots, [\textbf{not}]cmd_m(\rho_m)[\![c_1, \mathbf{E}:c_2]\!],$$

which expresses conditional assertion or retraction of a rule ρ_1, given by $cmd_1(\rho_1)$, depending on other commands $[\textbf{not}]cmd_2(\rho_2)$, $\ldots$,$[\textbf{not}]cmd_m(\rho_m)$, and conditioned with the proviso whether c_1 belongs to the current belief set and whether c_2 is in the actual event. Each condition in "$[\![\ldots]\!]$", both of the form c_1 and $\mathbf{E}:c_2$, can be substituted by a list of such conditions.

The basic EPI commands are similar to the corresponding ones of LUPS (for differences in their meaning, see [3] and cf. Section 7.1), plus the additional command **ignore**, which allows to skip unintended updates. Intuitively, **assert** and **retract** commands respectively force the insertion and retraction of rules into and from the knowledge base. The insertion or retraction persist by inertia until the rule is retracted or re-asserted. The option **event** indicates that such inertia effect is not intended, therefore the rule is only supposed to be asserted or retracted *in the given step of the evolution*, and to be retracted or reasserted afterwards. So-called *persistent commands*, viz. **always** and **cancel**, are meant to assert a rule in each successive update step (until the **always** command is itself deactivated), and to stop this iterated assertion, respectively. An option **event** for this iterated assertion means that the rule which is inserted in this way is not supposed to persist by inertia, i.e., as soon as the repeated insertion stops, the rule will be retracted. Of

$$
\begin{array}{lcl}
\langle stat \rangle & ::= & \langle comm \rangle \text{ [} \mathbf{if}\ \langle cond1 \rangle \text{] [} [\![\ \langle cond2 \rangle\ ]\!] \text{]};\\
\langle c\_name \rangle & ::= & \mathbf{assert}[\mathbf{\_event}] \mid \mathbf{retract}[\mathbf{\_event}] \mid \mathbf{always}[\mathbf{\_event}] \mid \mathbf{cancel} \mid \mathbf{ignore};\\
\langle r\_id \rangle & ::= & \langle rule \rangle \mid \langle r\_var \rangle;\\
\langle lit\_id \rangle & ::= & \langle literal \rangle \mid \langle lit\_var \rangle;\\
\langle comm \rangle & ::= & \langle c\_name \rangle (\langle r\_id \rangle)\ ;\\
\langle cond1 \rangle & ::= & [\mathbf{not}]\ \langle comm \rangle \mid [\mathbf{not}]\ \langle comm \rangle\ ,\ \langle cond1 \rangle\ ;\\
\langle cond2 \rangle & ::= & \langle kb\_conds \rangle \mid \mathbf{E} : \langle ev\_conds \rangle \mid \langle kb\_conds \rangle,\ \mathbf{E} : \langle ev\_conds \rangle;\\
\langle kb\_conds \rangle & ::= & \langle kb\_cond \rangle \mid \langle kb\_cond \rangle\ ,\ \langle kb\_conds \rangle;\\
\langle kb\_cond \rangle & ::= & \langle r\_id \rangle \mid [not] \langle lit\_id \rangle\ ;\\
\langle ev\_conds \rangle & ::= & \langle ev\_cond \rangle \mid \langle ev\_cond \rangle\ ,\ \langle ev\_conds \rangle;\\
\langle ev\_cond \rangle & ::= & \langle lit\_id \rangle \mid \langle r\_id \rangle\ ;
\end{array}
$$

Table 1: Syntax of an update statement in EPI.

course complex interactions between update commands can be originated, and we shall carefully consider them in our semantic definition, in order to avoid unintended conflicts and "side effects".

Note that in LUPS no conditions on rules and external events can be explicitly expressed, nor dependencies between update commands. We also extend the language by permitting variables for rules and literals in the update commands, ranging over the universe of the current belief set and of the current event (syntactic safety conditions can be easily checked). By convention, variable names start with capital letters.

We can now define what an update policy is:

Definition 3 *An* update policy $\mathcal{U}$ *is a finite set of* EPI *statements.*

For instance, the EPI statement

$$\mathbf{assert}(R)\ \mathbf{if\ not\ ignore}(R)[\![E : R]\!] \tag{1}$$

means that all rules in the event have to be incorporated into the new knowledge base, except if it is explicit specified that the rule is to be ignored.

3.2.1 Examples

Let us now have a look at two examples, which will be used in the rest of this paper.

Example 2 *[Shopping agent] Consider a simple agent selecting Web shops in search for some specific merchandise. Suppose its knowledge base, KB, contains the rules*

$$
\begin{array}{rr}
r_1: & query(S) \leftarrow sale(S), up(S), not\ \neg query(S);\\
r_2: & try\_query \leftarrow query(S);\\
r_3: & notify \leftarrow not\ try\_query;
\end{array}
$$

and a fact $r_0 : date(0)$ as an initial time stamp. Here, r_1 expresses that a shop S, which has a sale and whose Web site is up, is queried by default; r_2 and r_3 serve to detect that no site is queried, which causes *notify to be true. Assume that an event, E, might be any consistent set of facts or ground rules of form $sale(s) \leftarrow date(t)$, stating that shop s has a sale on date t, such that E contains at most one time stamp $date(\cdot)$.*

An update policy $\mathcal{U}$ might be defined as follows. Assume it contains the incorporate-by-default statement (1), as well as:

$$\begin{array}{l}\mathbf{always}(sale(S) \leftarrow date(T)) \ \mathbf{if}\ \mathbf{assert}(sale(S) \leftarrow date(T));\\ \mathbf{cancel}(sale(S) \leftarrow date(T))[\![date(T), T \neq T', \mathbf{E}\!:\!date(T')]\!];\\ \mathbf{retract}(sale(S) \leftarrow date(T))[\![date(T), T \neq T', \mathbf{E}\!:\!date(T')]\!].\end{array}$$

Informally, the first statement repeatedly confirms the information about a future sale, which guarantees that it is effective on the given date, while the second statement revokes this. The third one removes information about a previously ended sale (assuming the time stamps increase). Furthermore, $\mathcal{U}$ includes also the following statements:

$$\begin{array}{l}\mathbf{retract}(date(T))[\![date(T), T \neq T', \mathbf{E} : date(T')]\!];\\ \mathbf{ignore}(sale(s_1))[\![\mathbf{E} : sale(s_1)]\!];\\ \mathbf{ignore}(sale(s_1) \leftarrow date(T))[\![\mathbf{E} : sale(s_1) \leftarrow date(T)]\!].\end{array}$$

The first statement keeps the time stamp $date(t)$ in KB unique, and removes the old value. The other statements simply state that sales information about shop s_1 is ignored.

Let us now consider a more complex example.

Example 3 *[Mailing agent] Consider a mailing agent which has the following initial knowledge base KB, whose rules are instantiated over suitable variable domains:*

$$\begin{array}{lrl} r_1: & type(M, private) & \leftarrow from(M, tom);\\ r_2: & type(M, business) & \leftarrow subject(M, project);\\ r_3: & type(M, other) & \leftarrow not\ type(M, private), not\ type(M, business), msg(M);\\ r_4: & remove(M) & \leftarrow date(M,T), today(T'), not\ save(M), not\ new\_msg(M),\\ & & \quad T' > (T + 30);\\ r_5: & found(M) & \leftarrow search(T), type(M,T), msg(M), not\ remove(M);\\ r_6: & success & \leftarrow found(M);\\ r_7: & failure & \leftarrow search(T), not\ success. \end{array}$$

The knowledge base KB provides a classification of messages types depending on the subject (rules r_1 and r_3). By means of r_3, a default type is assigned to all messages which are neither $private$ nor $business$. Automatic message removal is performed by r_4, when a message is older then thirty days and not saved yet, providing it is not an unread message (signaled by the atom new_msg). Rules r_5, r_6 and r_7 are used to check for all messages of a given type, which have not been asked for removal, and to signal if at least one such message has been found ($success$) or not ($failure$).

An event E might consist in this setting of one or more of the following items, possibly generated by a user:

- *at most one fact $today(d)$, for some date d;*
- *facts $save(m)$, $remove(m)$, $read(M)$ for mail identifiers m, signaling respectively that a message has to be marked as "saved", has to be removed, or has been read (i.e., it is no longer be considered as a "new message");*

- *at most one fact* $search(t)$*, for some mail type* $t \in \{other, business, private\}$*;*
- *zero or more sets of facts* $from(m, n)$*,* $date(m, d)$*, or* $subject(m, s)$ *for mail identifier* m*, name* n*, date* d*, and subject* s*. The first two attributes are mandatory for creating an object* msg *in the knowledge base.*

The update policy $\mathcal{U}$ *might contain, for example, the following statements (we do not claim here to design a complete mailing agent, we only show how the* EPI *language can be used for this purpose; note that different realizations can be easily thought of):*

$$\begin{array}{l}\mathbf{assert}(today(D))\ [\![\mathbf{E} : today(D)]\!];\\ \mathbf{retract}(today(D))\ \mathbf{if}\ \mathbf{assert}(today(D'))\ [\![today(D), D \neq D']\!];\end{array}$$

which provide update of date information, every time a new time stamp is contained in the current event.

Furthermore,

$$\begin{array}{l}\mathbf{assert}(new\_msg(M))\ [\![\mathbf{E} : from(M, N), date(M, D)]\!];\\ \mathbf{assert}(msg(M))\ \mathbf{if}\ \mathbf{assert}(new\_msg(M));\end{array}$$

create a new message instance if corresponding (minimal) $from$ *and* $date$ *information is available in the event, and signaling that the message in question is a new one (i.e., it has not been read yet). If then the information arrives, that an existing message has been read, the following statement records the change (but the* msg *instance remains):*

$$\mathbf{retract}(new\_msg(M))\ [\![new\_msg(M), \mathbf{E} : read(M)]\!].$$

For each new incoming message, the following statements serve to record in the knowledge base the corresponding available information:

$$\begin{array}{c}\mathbf{assert}(from(M, N))\ \mathbf{if}\ \mathbf{assert}(new\_msg(M))\ [\![\mathbf{E} : from(M, N)]\!];\\ \mathbf{assert}(subject(M, S))\ \mathbf{if}\ \mathbf{assert}(new\_msg(M))\ [\![\mathbf{E} : subject(M, S)]\!];\\ \mathbf{assert}(date(M, D))\ \mathbf{if}\ \mathbf{assert}(new\_msg(M))\ [\![\mathbf{E} : date(M, D)]\!].\end{array}$$

The following statements provide management of $save$ *and* $remove$ *commands:*

$$\begin{array}{l}\mathbf{assert}(save(M))\ [\![\mathbf{E} : save(M)]\!];\\ \mathbf{assert}(remove(M))\ [\![not\, new\_msg(M),\ \mathbf{E} : remove(M)]\!];\\ \mathbf{ignore}(remove(M))\ [\![save(M)]\!];\\ \mathbf{ignore}(remove(M))\ \mathbf{if}\ \mathbf{assert}(save(M));\\ \mathbf{retract}(msg(M))\ \mathbf{if}\ \mathbf{assert}(remove(M));\\ \mathbf{retract}(msg(M))\ [\![remove(M)]\!].\end{array}$$

The effect is to save or remove a message if an external request comes in, whereas the removal is only possible if no corresponding save information is available (either in the current event, or recorded in the knowledge base). Observe that in this way, once a message is saved, it cannot be deleted anymore, as far as the **ignore** *statements disable any removal request. Removal of unread messages is also not allowed. A removal causes the corresponding message instance to be retracted. Corresponding to these actions, retraction of available information about subjects and dates could be deleted too, by means of simple commands.*

Some further commands take care of (temporarily) activating a search, and of deleting the results after their use, in order not to affect successive searches:

$$\begin{array}{l}\mathbf{assert\_event}(search(T))\ [\![\mathbf{E} : search(T)]\!];\\ \mathbf{retract}(found(T))\ [\![found(T)]\!];\\ \mathbf{retract}(success)\ [\![success]\!];\\ \mathbf{retract}(failure)\ [\![failure]\!].\end{array}$$

Using the signal atom new_msg, it is possible, for example, to activate a reminder for the user, that he has yet unread messages:

$$\begin{array}{l}\mathbf{always}(new\_mail!)\ \mathbf{if}\ \mathbf{assert}(new\_msg(M));\\ \mathbf{cancel}(new\_mail!)\ \mathbf{if}\ \mathbf{retract}(new\_msg(M)),\ \mathbf{not}\ \mathbf{assert}(new\_msg(M'))\\ \qquad\qquad\qquad [\![not\ new\_msg(M''), M'' \neq M]\!].\end{array}$$

While all policy statements introduced so far for this example concern assertion or retraction of pure facts, it is of course possible to express assertion or retraction of more complex rules, e.g.,

$$\begin{array}{l}\mathbf{assert\_event}(\,remove(M) \leftarrow not\,new\_msg(M),\\ \qquad\qquad\qquad\qquad not\,type(M, business)\,)\ [\![\mathbf{E} : max\_size]\!];\end{array}$$

which determines the cancellation of all messages which have already been read and do not concern business issues, if an event notifies that the maximal size of the mail account is reached. Observe that the option **event** *is required to ensure that the rule is deactivated in the following step (otherwise, private messages would be automatically canceled after reading, even if the account size is right).*

3.3 Semantics

The semantics of a knowledge state is computed in two main steps, as depicted in Figure 1. First of all, the update policy is *evaluated* over the current state, i.e., conditions in "$[\![\ldots]\!]$" are checked for all their ground instances. Only those statements are sorted out for which conditions are satisfied. Viewing these statements as a logic program, stable models are computed, resulting in a set of executable commands. In a second step, a compilation function encodes into an update sequence the effects of update commands over the initial knowledge base, so that the belief set of the resultant sequence is as intended.

3.3.1 Executable commands

According to the semantics of EPI, as depicted in Figure 1, at Step i, we first determine the set of executable commands U_i given the current knowledge state $KS_{i-1} = \langle KB; E_1, \ldots, E_{i-1}\rangle$ and its associated belief set $Bel(KS_{i-1}) = Bel(\boldsymbol{P}_{i-1})$, where $\boldsymbol{P}_{i-1} = (P_0, \ldots, P_{i-1})$. To this end, we evaluate the update policy $\mathcal{U}$ over the new event E_i and the belief set $Bel(\boldsymbol{P}_{i-1})$. Let $\mathcal{G}(\mathcal{U})$ be the grounded version of $\mathcal{U}$ over the language $\mathcal{A}$ underlying the given update sequence and the received events. Then, the set $\mathcal{G}(\mathcal{U})^i$ of reduced update statements at Step i is given by

$$\begin{array}{ll}\mathcal{G}(\mathcal{U})^i = \{cmd(\rho)\ \mathbf{if}\ C_1 \mid cmd(\rho)\ \mathbf{if}\ C_1[\![C_2]\!] \in \mathcal{G}(\mathcal{U}), & \text{where } C_2 \text{ has the form}\\ & C_2 = c_1, \ldots, c_l,\ \mathbf{E} : r_1, \ldots, r_m, \text{ and}\\ & Bel(\boldsymbol{P}_{i-1}) \models c_1, \ldots, c_l \text{ and } r_1, \ldots, r_m \in E_i\}.\end{array}$$

The update statements in $\mathcal{G}(\mathcal{U})^i$ are thus of form

$$cmd_1(\rho_1)\ \mathbf{if}\ [\mathbf{not}]\ cmd_2(\rho_2), \ldots, [\mathbf{not}]\ cmd_m(\rho_m).$$

Consider again the shopping agent of Example 2. If the initial knowledge base KB is as described in the example, and the first event E_1 contains $date(1)$ and

$$r_4: \; sale(s_1) \leftarrow date(1),$$

then $\mathcal{G}(\mathcal{U})^1$ is made up of the following update statements:

$$\begin{array}{l} \mathbf{assert}(date(1)) \textbf{ if not ignore}(date(1)); \\ \mathbf{assert}(r_4) \textbf{ if not ignore}(r_4); \\ \mathbf{always}(r_4) \textbf{ if assert}(r_4); \\ \mathbf{retract}(date(0)); \\ \mathbf{ignore}(r_4). \end{array}$$

Semantically, we interpret them as ordinary logic program rules

$$cmd_1(\rho_1) \leftarrow [not\,]cmd_2(\rho_2), \ldots, [not\,]cmd_m(\rho_m).$$

The program $\Pi_i^{\mathcal{U}}$ is the collection of all these rules, given $\mathcal{G}(\mathcal{U})^i$, together with the following constraints, which exclude contradictory commands:

$$\begin{array}{l} \leftarrow \mathbf{assert}[\_\mathbf{event}](R), \mathbf{retract}[\_\mathbf{event}](R); \\ \leftarrow \mathbf{always}[\_\mathbf{event}](R), \mathbf{cancel}(R). \end{array}$$

Definition 4 *Let $KS = \langle KB; E_1, \ldots, E_n \rangle$ be a knowledge state and $\mathcal{U}$ an update policy. Then, U_i is a set of* executable update commands at Step i $(i \leq n)$ *iff U_i is an answer set of the grounding $\mathcal{G}(\Pi_i^{\mathcal{U}})$ of $\Pi_i^{\mathcal{U}}$.*

Following our example, the set of executable update commands at Step 1 is $U_1 = \{\mathbf{ignore}(r_4), \mathbf{retract}(date(0)), \mathbf{assert}(date(1))\}$.

Since update statements do not contain strong negation, executable update commands are in fact *stable models* of $\mathcal{G}(\Pi_i^{\mathcal{U}})$ [19]. Furthermore, since programs may in general have more than one answer set, or no answer set at all, we assume a suitable *selection function*, $Sel(\cdot)$, returning a particular U_i if an answer set exists, or, otherwise, returning $U_i = \{\mathbf{assert}(\bot_i \leftarrow)\}$, where $\bot_i$ is a special atom not occurring elsewhere. These atoms are used for signaling that the update policy encountered inconsistency. They can easily be filtered out from Bel, if needed, restricting the outcomes of the update to the original language.

3.3.2 Compilation to update sequences

Next, the executable-command sets $U_1, \ldots, U_i$ have to be compiled into an update sequence $(P_0, \ldots, P_i)$, serving as input for the function Bel. This can be realized in several ways, as far as different transformations tr can be defined serving different purposes, and up to abstract mappings between knowledge states and update sequences. We choose here to give a syntactical transformation, which is a refinement of the one given for LUPS [3], in that it avoids some unintended conflicts between commands, and which is more readable (cf. Section 7.1). We could have given a definition in terms of sets of rules to be added to or deleted from the current knowledge base, but the approach we introduce offers an insight in the underlying mechanisms.

In what follows, we assume a suitable naming function for rules in the update sequence, enforcing that each rule r is associated with a unique name n_r. Additional temporary sets are used, denoted Add_i, $Enable_i$, and $Disable_i$, to make the transformation easier to understand. These sets contain rules which are to be inserted, activated, and deactivated respectively in the resulting knowledge base.

Definition 5 *Let* $KS = \langle KB; E_1, \ldots, E_n \rangle$ *be a knowledge state and* $\mathcal{U}$ *an update policy. Then, for* $i \geq 0$, $tr(KB; U_1, \ldots, U_i) = (P_0, P_1, \ldots, P_i)$ *is inductively defined as follows, where* $U_1, \ldots, U_i$ *are the executable commands according to Definition 4:*

$i = 0:$ *Set* $P_0 = \{H(r) \leftarrow B(r), on(n_r) \mid r \in KB\} \cup \{on(n_r) \leftarrow \mid r \in KB\}$, *where* $on(\cdot)$ *are new atoms. Furthermore, initialize the sets* PC_0 *of* persistent commands *and* EC_0 *of* effective commands *to* $\emptyset$.

$i \geq 1:$ EC_i *and* PC_i *are as follows:*

$$EC_i = \{cmd(r) \mid cmd(r) \in U_i \wedge \mathbf{ignore}(r) \notin U_i\};$$

$$\begin{aligned} PC_i = PC_{i-1} &\cup \{\mathbf{always}(r) \mid \mathbf{always}(r) \in EC_i\} \\ &\cup \{\mathbf{always\_event}(r) \mid \mathbf{always\_event}(r) \in EC_i \\ &\qquad \wedge \mathbf{always}(r) \notin EC_i \cup PC_{i-1}\} \\ &\setminus (\{\mathbf{always\_event}(r) \mid \mathbf{always}(r) \in EC_i\} \\ &\quad \cup \{\mathbf{always[\_event]}(r) \mid \mathbf{cancel}(r) \in EC_i\}). \end{aligned}$$

P_i *is computed based on the content of the sets* Add_i, $Enable_i$ *and* $Disable_i$, *as follows:*

$$\begin{aligned} P_i = \{on(n_r) \leftarrow\ , H(r) \leftarrow B(r), on(n_r) \mid r \in Add_i\} \\ \cup \{on(n_r) \leftarrow \mid r \in Enable_i\} \\ \cup \{\neg on(n_r) \leftarrow \mid r \in Disable_i\}, \end{aligned}$$

where Add_i, $Enable_i$ *and* $Disable_i$ *are respectively:*

$$r \in Add_i \Leftrightarrow \mathbf{assert[\_event]}(r) \in EC_i \ \vee\ \mathbf{always[\_event]}(r) \in PC_i;$$

$$\begin{aligned} r \in Enable_i \Leftrightarrow\ & \mathbf{retract\_event}(r) \in EC_{i-1} \ \wedge\ \mathbf{retract}(r) \notin EC_{i-1} \wedge \\ & \mathbf{retract[\_event]}(r) \notin EC_i \ \wedge\ (\mathbf{always\_event}(r) \notin PC_{i-1} \\ & \vee \mathbf{cancel}(r) \notin EC_i); \end{aligned}$$

$$\begin{aligned} r \in Disable_i \Leftrightarrow\ & \textit{one of the following holds:} \\ & (1)\ \mathbf{retract[\_event]}(r) \in EC_i \ \wedge\ \mathbf{always[\_event]}(r) \notin PC_i \\ & (2)\ \mathbf{cancel}(r) \in EC_i \ \wedge\ \mathbf{always\_event}(r) \in PC_{i-1} \\ & \qquad \wedge \mathbf{assert[\_event]}(r) \notin EC_i \\ & (3)\ \mathbf{assert\_event}(r) \in EC_{i-1} \ \wedge\ \mathbf{assert}(r) \notin EC_{i-1} \\ & \qquad \wedge \mathbf{always[\_event]}(r) \notin PC_i \ \wedge\ \mathbf{assert[\_event]}(r) \notin EC_i. \end{aligned}$$

Observe that, for optimization purposes, the set Add_i could be reduced by moving the rules which already exist in KB into the set $Enable_i$, thus avoiding unnecessary repeated insertion of rules.

The transformation introduces in the body of each rule r a control atom, $on(n_r)$ (where on is a reserved predicate), which is used to (de-)activate the rule as requested. Given the set U_i of executable commands, first those commands are filtered out which affect some rule r which is explicitly to be ignored, and the subset EC_i of *effective commands* is constructed. In this way, the effect of $\mathbf{ignore}$ commands is simply to *skip* unintended updates. E.g., if a certain rule serves as a *tick of the clock* for the agent, updates to this rule can be avoided in order to preserve the integrity of behavior.

The *persistent commands* set, PC_i, contains those update commands which affect more than one evolution step. In the construction of PC_i, all occurrences of $\mathbf{always\_event}(r)$ are deleted as soon as $\mathbf{always}(r)$ is stated. The idea is that the $\mathbf{always}(r)$ command decides the persistence by inertia of the rule, independent of other previous non-inertial insertions.

Then, while building P_i, simple syntactical checks avoid some possible conflicts, which may arise either between commands in successive updates or between persistent and simple commands. Conflicts caused by successive updates are solved in favor of new update commands, while conflicts between simple and persistent commands are solved in favor of the latter ones. Different and more complex conflict avoidance strategies can of course be imagined, and corresponding changes can be encoded in the definition of tr. Based on our definitions, a conflict would occur if one of the following conditions holds:

- $\mathbf{retract\_event}(r) \in EC_{i-1} \wedge \mathbf{retract}[\mathbf{\_event}](r) \in EC_i$:

  The option **event** for the **retract** command at Step $i-1$ requires r to be retracted *only* at that step, thus the rule should be re-activated at Step i, by means of $on(n_r)$. If a **retract** command for the same rule is present in EC_i, then a conflict would arise. This is avoided by assigning higher priority to the commands in EC_i, and thus $on(n_r)$ is not added.

- $\mathbf{retract}[\mathbf{\_event}](r) \in EC_i \wedge \mathbf{always}[\mathbf{\_event}](r) \in PC_i$:

  The $\mathbf{retract}(r)$ command requires adding $\neg on(n_r)$. The **always**[**_event**] command for the same rule r in PC_i requires on the other hand adding $on(n_r)$, thus a conflict would arise. This is avoided by assigning higher priority to the command in PC_i, and thus $\neg on(n_r)$ is not added.

- $\mathbf{always\_event}(r) \in PC_{i-1} \wedge \mathbf{assert}[\mathbf{\_event}](r) \in EC_i \wedge \mathbf{cancel}(r) \in EC_i$:

  The option **event** for the **always** command at Step $i-1$, together with the **cancel** command at Step i, would require to deactivate r at Step i, by means of $\neg on(n_r)$. If an **assert**[**_event**] command for the same rule is present in EC_i, then a conflict would arise. This is avoided by assigning higher priority to the last incoming command in EC_i, and thus $\neg on(n_r)$ is not added.

- $\mathbf{assert\_event}(r) \in EC_{i-1} \wedge \mathbf{assert}[\mathbf{\_event}](r) \in EC_i$:

  The option **event** for the **assert** command at Step $i-1$ requires r to be asserted *only* at that step, thus r should be deactivated again at Step i, by means of $\neg on(n_r)$. If an **assert** command for the same rule is present in EC_i, then a conflict would arise. This is avoided by assigning higher priority to the command in EC_i, and thus $\neg on(n_r)$ is not added.

- $\mathbf{assert\_event}(r) \in EC_{i-1} \wedge \mathbf{always}[\mathbf{\_event}](r) \in PC_i$:

  The option **event** for the **assert** command at Step $i-1$ requires r to be asserted *only* at that step, thus r should be deactivated again at Step i, by means of $\neg on(n_r)$. If an **always** command for the same rule is present in PC_i, then a conflict would arise. This is avoided by assigning higher priority to the command in PC_i, and thus $\neg on(n_r)$ is not added.

Observe that rule insertion due to an **always** command could be further optimized to avoid unnecessary copies of rules, i.e., if $\mathbf{always}(r)$ is already present in PC_{i-1}, then adding $on(n_r)$ is enough, since the rule is already in the knowledge base.

On the basis of this compilation, the belief set for a knowledge state KS can be defined:

Definition 6 *Let KS and $\mathcal{U}$ be as in Definition 5, and let $U_1, \ldots, U_n$ be the corresponding executable commands obtained from Def. 4. Then, the* belief set *of KS is given by $Bel(KS) = Bel(tr(KB; U_1, \ldots, U_n))$.*

Example 4 *Reconsider Example 2 and suppose the event $E_1 = \{sale(s_0), date(1)\}$ occurs at $KS = \langle KB \rangle$. Then,*

$$\mathcal{G}(\mathcal{U})^1 = \{\mathbf{assert}(sale(s_0))\ \mathbf{if\ not\ ignore}(sale(s_0)),\\ \mathbf{assert}(date(1))\ \mathbf{if\ not\ ignore}(date(1)),\ \mathbf{retract}(date(0))\}.$$

The corresponding program $\Pi_1^{\mathcal{U}}$ *has the single answer set*

$$S = \{\mathbf{assert}(sale(s_0)),\ \mathbf{assert}(date(1)),\ \mathbf{retract}(date(0))\},$$

which is compiled via tr *to* $PC_1 = PC_0 \setminus \{\mathbf{assert\_[event]}(date(0))\} = \emptyset$ *and* $P_1 = \{sale(s_0) \leftarrow on(r'_1),\ date(1) \leftarrow on(r'_2),\ on(r'_1) \leftarrow,\ on(r'_2) \leftarrow,\ \neg on(r_0) \leftarrow\}$*. The belief set* $Bel(\langle KB, E_1\rangle) = Bel(P_0, P_1)$ *contains thus* $sale(s_0)$ *and* $query(s_0)$*.*

4 Properties

In this section, we consider some properties of the framework for update policies that we have introduced above.

4.1 Structural properties

We discuss here some properties of $Bel(KS)$ for particular update policies, using the definition of Bel based on the update answer sets approach of [11], as explained in Section 2. We stress that properties similar to those given below are also satisfied by similar instantiations of Bel, like e.g., dynamic logic programming [2]. First, we note some basic properties. Recall that we suppose that the two properties stated in Section 2 are satisfied by any "reasonable" belief function in this setting.

Definition 7 (Incremental compilation) *A compilation function,* tr*, is incremental iff for each* $KS = \langle KB; E_1, \ldots, E_n\rangle$*,* $tr(KB; U_1, \ldots, U_n) = (P_0, \ldots, P_n)$ *such that for all* $0 \leq i \leq n$*,* $tr(KB; U_1, \ldots, U_i) = (P_0, \ldots, P_i)$*.*

Proposition 1 *The compilation* tr *introduced above is incremental.*

A similar result applies to the compilation suggested for LUPS [3].

Definition 8 (Contracting belief function) *A belief function,* Bel*, is contracting iff the following conditions hold:*

1. $Bel(\boldsymbol{P} + \emptyset + \boldsymbol{P}') = Bel(\boldsymbol{P} + \boldsymbol{P}')$*, for all* $\boldsymbol{P}, \boldsymbol{P}'$ *update sequences, where* $+$ *denotes the concatenation of sequences; and*

2. $Bel(\boldsymbol{P}) = Bel(P_0, \ldots, P_{i-1}, P_i \setminus \{r\}, P_{i+1}, \ldots, P_n)$*, for all update sequences* $\boldsymbol{P} = (P_0, \ldots, P_n)$ *and all* $r \in P_i \cap P_j$ *with* $i < j$*.*

A belief function satisfying only the first condition will be called contracting under empty updates*; as well, if only the second condition holds, the belief function will will be called* contracting under rule repetition.

Proposition 2 *The belief function,* Bel*, obtained from the semantics of update answer sets is contracting.*

Proof. For proving Property 1, we have to show that, given two update sequences $(P_0, \ldots, P_m)$ and $(P_{m+2}, \ldots, P_n)$, it holds that $Bel(P_0, \ldots, P_m, \emptyset, P_{m+2}, \ldots, P_n) = Bel(P_0, \ldots, P_m, P_{m+2}, \ldots, P_n)$. Let us denote the first sequence by $\boldsymbol{P}^{\emptyset}$, and the second one simply by $\boldsymbol{P}$. The rejection sets for the two sequences coincide; in fact, given a set S, $Rej(S, \boldsymbol{P}^{\emptyset}) = \bigcup_{i=0}^{m} Rej_i(S, \boldsymbol{P}^{\emptyset}) \cup Rej_{m+1}(S, \boldsymbol{P}^{\emptyset}) \cup \bigcup_{i=m+2}^{n} Rej_i(S, \boldsymbol{P}^{\emptyset})$. Now, in $P_{m+1} = \emptyset$ there is no rule which could be chosen for rejection, therefore $Rej_{m+1}(S, \boldsymbol{P}^{\emptyset}) = \emptyset$. Moreover, for no rule $r \in Rej_i(S, \boldsymbol{P}^{\emptyset})$, with $0 \leq i \leq m$, it can be the case that the rejection condition is satisfied by a rule $r' \in P_j$ with $j = m+1$, since $P_{m+1} = \emptyset$, so $\bigcup_{i=0}^{m} Rej_i(S, \boldsymbol{P}^{\emptyset}) = \bigcup_{i=0}^{m} Rej_i(S, \boldsymbol{P})$ holds. The rejection sets for $i \geq m+1$ are anyway not affected by P_{m+1}, and $\bigcup_{i=m+2}^{n} Rej_i(S, \boldsymbol{P}^{\emptyset}) = \bigcup_{i=m+2}^{n} Rej_i(S, \boldsymbol{P})$ holds as well. Therefore, we have $Rej(S, \boldsymbol{P}^{\emptyset}) = Rej(S, \boldsymbol{P})$, and, for $0 \leq i \leq n$, $\bigcup_{i \neq m+1} P_i = \bigcup_{i=0}^{n} P_i$ as well; thus, the answer sets of the two sequences coincide.

Concerning Property 2, let us denote the two occurrences of rule r in P_i and in P_j by r_i and r_j, respectively. Rule r_i can affect the semantics of the update sequence only if $r_i \notin Rej(S, \boldsymbol{P})$ for some $S \in \mathcal{AS}(\boldsymbol{P})$, that is, only if $r_i \notin Rej_i(S, \boldsymbol{P})$. Based on the condition for rejection, this means that there does not exist some $r' \in P_k$, $k \geq i$, with conflicting head and with S satisfying both the body of r_i and of r'. This condition holds for r_j as well, since $j \geq i$. Therefore, $r_j \notin Rej(S, \boldsymbol{P})$. This means, the occurrence r_j would not be rejected as well, and the semantics is not affected by deleting r_i. □

The same holds for dynamic logic programming. We can now state some structural properties of the framework.

Proposition 3 (Empty policy) *Let $KS = \langle KB; E_1, \ldots, E_n \rangle$ be a knowledge state, and let $\mathcal{U} = \emptyset$. If Bel is contracting under empty updates, then $Bel(KS_i) = Bel(KB)$ for any $i \in \{1, \ldots, n\}$.*

Proof. For any $i \in \{1, \ldots, n\}$, $\mathcal{G}(\mathcal{U})^i = \emptyset$ and thus $U_i = \emptyset$. By means of tr clearly $P_i = \emptyset$ as well, leading to $Bel(KS_i) = Bel(P_0, \emptyset, \ldots, \emptyset) = Bel(P_0)$ in view of the assumption that Bel is contracting under empty updates, and thus, by the definition of tr, $Bel(KS_i) = Bel(KB)$. □

Proposition 4 (Unconditional assert policy) *For any knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ and $\mathcal{U} = \{\mathbf{assert}(r)\ [\![\mathbf{E} : r]\!]\}$, we have that $Bel(KS_i) = Bel(KB, E_1, \ldots, E_i)$, for any $i \in \{1, \ldots, n\}$.*

Proof. For any $i \in \{1, \ldots, n\}$, $\mathcal{G}(\mathcal{U})^i = \{\mathbf{assert}(r) \mid r \in E_i\} = U_i$, and thus, by means of tr, $P_i = \{r,\ on(n_r) \leftarrow \mid r \in E_i\}$. Therefore, $Bel(KS_i) = Bel(P_0, P_1, \ldots, P_i) = Bel(KB, E_1, \ldots, E_i)$. □

4.1.1 Physical removal of rules

An important issue is the growth of the agent's knowledge base, as the modular construction of the update sequence through tr causes some rules and facts to be repeatedly inserted. This is addressed next, where we discuss the physical removal of rules from the knowledge base.

Lemma 1 *Let $\boldsymbol{P} = (P_0, \ldots, P_n)$ be an update sequence. Let $r \in P_i, r' \in P_j$ with $i < j$, such that one of (i)-(iii) holds, where L is ground:*

(*i*) $r = r'$;

(*ii*) $r = L \leftarrow$, $r' = \neg L \leftarrow$ *and either L is not unifiable with any rule head in $P_{j+1} \cup \cdots \cup P_n$, or $L \leftarrow\ \in P_{j+1} \cup \cdots \cup P_n$;*

(iii) $\neg L \in B(r)$*,* $r' = L \leftarrow$ *and no rule head in* $P_{j+1} \cup \cdots \cup P_n$ *is unifiable with* $\neg L$*.*

Then, $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, \ldots, P_{i-1}, P_i \setminus \{r\}, P_{i+1}, \ldots, P_n)$*.*

Proof. In case (i), the result follows by Proposition 2. In case (ii), suppose for simplicity that $\boldsymbol{P}$ is ground (the non-ground case is easily reduced to this one), and consider any update answer set S of $\boldsymbol{P}$ (resp., of $\boldsymbol{P}' = P'_0, \ldots, P'_n$ where $P'_h = P_h$, for all $h \in \{0, \ldots, n\} \setminus \{i\}$, and $P'_i = P_i \setminus \{r\}$). If r' is not rejected, i.e., $r' \notin Rej_j(S, \boldsymbol{P})$, then $r \in Rej(S, \boldsymbol{P})$ and thus $\bigcup_i P_i \setminus Rej(S, \boldsymbol{P}) = \bigcup_i P'_i \setminus Rej(S, \boldsymbol{P}')$. Hence, S is an update answer set of $\boldsymbol{P}'$ (resp., $\boldsymbol{P}$). Otherwise, if r' is rejected, i.e., $r' \in Rej_j(S, \boldsymbol{P})$ (= $Rej_j(S, \boldsymbol{P}')$), then by hypothesis some r'' of form $L \leftarrow$ is in $P_{j'}$ where $j' > j$. Then, $r'' \notin Rej_{j'}(S, \boldsymbol{P})$, (=$Rej_{j'}(S, \boldsymbol{P}')$), which means $\bigcup_i P_i \setminus Rej(S, \boldsymbol{P}) = \bigcup_i P'_i \setminus Rej(S, \boldsymbol{P}')$. Therefore, S is an answer set of $\boldsymbol{P}'$ (resp. $\boldsymbol{P}$). Finally, in case (iii), r' can not be rejected in any case, and L belongs, by means of r', to the belief set of $\boldsymbol{P}$. Thus, the body of r is unsatisfiable, and its removal does not affect the update answer sets. □

The following property now holds:

Theorem 1 (Physical removal of rules) *Let* KS *be a knowledge state such that* $Bel(KS) = Bel(\boldsymbol{P})$*, where* $\boldsymbol{P} = (P_0, \ldots, P_n)$ *is compiled from* KS *as aboves. Let* $\boldsymbol{P}^j$*, for* $0 \leq j$*, denote the result of applying* j *times on* $\boldsymbol{P}$ *the rule removal procedure as in Lemma 1. Then:*

1. *For all* $j \geq 0$*,* $Bel_{\mathcal{A}}(\boldsymbol{P}^j) = Bel_{\mathcal{A}}(\boldsymbol{P})$*.*
2. *Let* $\boldsymbol{P}^*$ *result from* $\boldsymbol{P}$ *after applying as long as possible the rule removal procedure as in Lemma 1, and let* $\boldsymbol{P}^- = (P_0^-, \ldots, P_n^-)$*, where*

$$\begin{aligned} P_i^- \;=\; & \{H(r) \leftarrow B(r) \setminus \{on(n_r)\} \mid r \in P_i^*,\ on(n_r) \leftarrow\ \in \boldsymbol{P}^*\} \\ & \setminus \{(\neg)on(n_s) \leftarrow\ \mid (\neg)on(n_s) \leftarrow\ \in \boldsymbol{P}^*\}. \end{aligned}$$

   Then, $Bel_{\mathcal{A}}(KS) = Bel_{\mathcal{A}}(\boldsymbol{P}^-)$*.*

Proof. We show Property 1 inductively on $j \geq 1$ (for $j = 0$, Lemma 1 is not applied at all). For $j = 1$, the theorem amounts to Lemma 1, thus it holds. Let us now suppose $Bel(\boldsymbol{P}^{j-1}) = Bel(\boldsymbol{P})$ for all $j > 1$ (we omit subscript '$\mathcal{A}$'), and let us prove $Bel(\boldsymbol{P}^j) = Bel(\boldsymbol{P})$. For the application of Lemma 1, we have $Bel(\boldsymbol{P}^{j-1}) = Bel(\boldsymbol{P}^j)$, and by the inductive hypothesis, $Bel(\boldsymbol{P}^j) = Bel(\boldsymbol{P})$ follows.

Based on Lemma 1, to prove Property 2 we must show that $Bel_{\mathcal{A}}(\boldsymbol{P}^*) = Bel_{\mathcal{A}}(\boldsymbol{P}^-)$. First of all, observe the following: each $S \in \mathcal{AS}(\boldsymbol{P}^*)$ is made up of two parts, which we denote by S^* and S_{on}, respectively; the former containing literals from the original language and possibly special atoms $\perp_i$ for $0 \leq i \leq n$, and the latter consisting of literals $(\neg)on(n_r)$ for rules r. Now, as far as literals $(\neg)on(n_r)$ can occur in the head of facts only, and since $\boldsymbol{P}^*$ does not contain contradictory facts (due to Condition (ii) in Lemma 1), the part S_{on} is exactly the same for each $S \in \mathcal{AS}(\boldsymbol{P}^*)$.

Let us call a set S^* *completing* for $\boldsymbol{P}^*$ if and only if $S^* = S \setminus S_{on}$ for some $S \in \mathcal{AS}(\boldsymbol{P}^*)$. Then, S^* is completing for $\boldsymbol{P}^*$ iff S^* is an answer set of $(P_0^{*\prime}, \ldots, P_n^{*\prime})$, where $P_i^{*\prime} = \{H(r) \leftarrow B(r) \setminus \{on(n_r)\} \mid r \in P_i^*, on(n_r) \in \boldsymbol{P}^*\} \setminus \{(\neg)on(n_r) \leftarrow\mid (\neg)on(n_r) \in \boldsymbol{P}^*\}$, which is exactly the construction of $\boldsymbol{P}^-$. Thus, the equivalence of the belief sets over $\mathcal{A}$ follows. □

Thus, we can purge the knowledge base and remove duplicates of rules, as well as all deactivated (retracted) rules. We remark that the update sequence $\boldsymbol{P}^*$ is unique: as can be seen, the rewriting is always terminating and confluent. Note that if further updates are expected, then $\boldsymbol{P}^*$ instead of $\boldsymbol{P}^-$ should be kept.

4.1.2 History contraction

Another relevant issue is the possibility, for some special case, to *contract the agent's update history*, and compute its belief set at Step i merely based on information at Step $i-1$. Let us call $\mathcal{U}$ a *factual assert policy* if all **assert[_event]** and **always[_event]** statements in $\mathcal{U}$ involve only facts. In this case, the compilation tr for a knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ can be simplified: (1) $P_0 = KB$, and (2) the construction of each P_i involves facts $L \leftarrow$ and $\neg L \leftarrow$ instead of $on(n_r) \leftarrow$ and $\neg on(n_r) \leftarrow$, respectively. For such sequences, the following holds:

Lemma 2 *Let* $\boldsymbol{P} = (P_0, \ldots, P_n)$ *be an update sequence such that* P_i *contains only (ground) facts, for* $1 \leq i \leq n$. *Then,* $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, P_{u_n})$, *where* $P_{u_1} = P_1$, *and* $P_{u_{i+1}} = P_{i+1} \cup (P_{u_i} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P_{i+1}\})$.

Proof. This can be proved by induction on $n \geq 0$. For $n = 0$, no update takes place. For $n \geq 1$, let us suppose that the lemma holds for n, i.e., $Bel(P_0, \ldots, P_n) = Bel(P_0, P_{u_n})$, for all sequences of length n, and let us show it for $n + 1$. Now, $Bel(P_0, \ldots, P_n, P_{n+1}) = Bel(P_0, \ldots, P_{n-1}, P'_n)$, where, based on the semantics of update answer sets, $P'_n = P_{n+1} \cup (P_n \setminus \{L \leftarrow \mid \neg L \leftarrow \in P_{n+1}\})$. The new sequence $(P_0, \ldots, P_{n-1}, P'_n)$ consists of n programs, thus the inductive hypothesis can be applied. We get $Bel(P_0, \ldots, P_{n-1}, P'_n) = Bel(P_0, P'_{u_n})$, with $P'_{u_n} = P'_n \cup (P_{u_{n-1}} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P'_n\})$. Substituting in this equation the definition of P'_n, and combining it with the general construction of P_{u_i}, we finally get that $Bel(P_0, P'_{u_n}) = Bel(P_0, P_{u_{n+1}})$, thus $Bel(P_0, \ldots, P_n, P_{n+1}) = Bel(P_0, P_{u_{n+1}})$, as intended. $\Box$

We can thus establish the following property for history contraction.

Theorem 2 *Let* $\mathcal{U}$ *be a factual assert policy and let* $\boldsymbol{P} = (P_0, \ldots, P_n)$ *be the compiled sequence obtained from* KS *by the simplified method described above. Then,* $Bel_{\mathcal{A}}(KS) = Bel_{\mathcal{A}}(KB, P_{u_n})$, *where* P_{u_n} *is as in Lemma 2.*

Proof. By Lemma 2, $Bel_{\mathcal{A}}(KS) = Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, P_{u_n})$, and, for the construction of P_0 by tr, we get $Bel_{\mathcal{A}}(P_0, P_{u_n}) = Bel_{\mathcal{A}}(KB, P_{u_n})$. $\Box$

4.2 Expressive core

An *expressive core* of the language EPI can be identified, that is, a set of *primitives* can be sorted out which possesses the same expressivity as the whole language. We describe this core and show that every update policy over the original language can be transformed into an *equivalent* update policy making use of only these primitives. We denote the core language by EPI_{core}, to distinguish it from the full one.

The actual core of the language consists only of commands **assert** and **retract**, i.e., no persistent command is available. Moreover, the way in which an EPI policy is encoded in an EPI_{core} one is standard and modularized. The proposed transformation also enjoys some structural properties which allow to derive equivalence and complexity results. The equivalence result means that actually the evolution only depends on the current belief set and on the incoming event, not on past steps.

Theorem 3 *Given an update policy* $\mathcal{U}$ *over* EPI, *there exists an update policy* $\mathcal{U}'$ *over* EPI_{core} (*in an extended alphabet*) *such that, for any knowledge state* $KS = \langle KB; E_1, \ldots, E_n \rangle$, *it holds that*

$$Bel_{\mathcal{A}}(tr(KB, U_1, \ldots, U_n)) = Bel_{\mathcal{A}}(tr(KB, U'_1, \ldots, U'_n)),$$

where $U_1, \ldots, U_n$ and $U'_1, \ldots, U'_n$ are the sets of executable commands for KS computed by $\mathcal{U}$ and $\mathcal{U}'$, respectively.

Here, we describe how such a $\mathcal{U}'$ is obtained; the proof of Theorem 3 can be found in [29]. The idea is to rewrite the update policy $\mathcal{U}$ to its new version $\mathcal{U}'$ which is made up of three different subprograms, which yields a modular policy, in which each module provides the input for the next one:

- Module **A** simulates the computation of the set U_i of executable commands;
- Module **B** simulates the computation of the sets PC_i and E_i of persistent and effective commands, respectively, based on the sets PC_{i-1} and E_{i-1} which are *recorded* in the current belief set;
- Module **C** computes P_i (actually, an equivalent program over the original alphabet).

We remark that the logic program $\Pi_i^{\mathcal{U}'}$, resulting at each step from policy $\mathcal{U}'$, consists of a *hierarchy* of three modules as well, thus its semantics can be obtained by combining the semantics of the modules in a bottom-up way (cf. the modularity properties of logic programs as discussed in [25, 17]).

The semantics of knowledge states is defined in the usual way, applying tr to the executable command sets $U'_1, \ldots, U'_n$ resulting from $\mathcal{U}'$, as $\mathcal{U}'$ is a usual policy over an extended language; no special semantics is needed.

Let us analyze in detail the content of $\mathcal{U}'$. The extended alphabet of $\mathcal{U}'$ contains, for each command occurring in $\mathcal{U}$, two new atoms as follows:

- for each command $\mathbf{assert}[\_\mathbf{event}](r)$, atoms $ASR_0(r)$ and $ASR(r)$ (respectively, $ASR\_EV_0(r)$ and $ASR\_EV(r)$ for the option **event**);
- for each command $\mathbf{retract}[\_\mathbf{event}](r)$, atoms $RET_0(r)$ and $RET(r)$ (respectively, $RET\_EV_0(r)$ and $RET\_EV(r)$);
- for $\mathbf{always}[\_\mathbf{event}](r)$, $ALW_0(r)$ and $ALW(r)$ (resp. $ALW\_EV_0(r)$ and $ALW\_EV(r)$);
- for $\mathbf{cancel}(r)$, $CAN_0(r)$ and $CAN(r)$;
- finally, for $\mathbf{ignore}(r)$, atoms $IGN_0(r)$ and $IGN(r)$.

Intuitively, as it will become clear by analyzing the content of Module **A**, the new atoms with subscript '0' (let us denote them by α_0) serve to simulate the computation of the set U_i by $\mathcal{U}$. Given a rule $r \in R_{\mathcal{A}}$, $\alpha_0(r)$ is only asserted in $\mathcal{U}'$ in case the corresponding command belongs to U_i. Moreover, these special atoms, which are new and do not appear elsewhere in the original program and policy, are then used by Module **B** as signal atoms to simulate the effects of the compilation tr for $\mathcal{U}$.

The policy $\mathcal{U}'$ is made up of the following modules over this extended alphabet:

(Module A) To sort out the executable commands:

(A.1) In each statement of $\mathcal{U}$, each occurrence of a command $\mathbf{cmd}(r)$ is replaced by the command $\mathbf{assert}(atom\_cmd_0(r))$, where $atom\_cmd$ is the new atom corresponding to $\mathbf{cmd}$, as introduced above.

(A.2) Additional statements are added to preserve consistency, playing the role of the integrity constraints; for each pair of commands $\alpha_1(r), \alpha_2(r)$ satisfying one of the two conditions:

- $\alpha_1 \in \{\mathbf{assert}(ASR_0(r)), \mathbf{assert}(ASR\_EV_0(r))\}$ and $\alpha_2 \in \{\mathbf{assert}(RET_0(r)), \mathbf{assert}(RET\_EV_0(r))\}$, or
- $\alpha_1 \in \{\mathbf{assert}(ALW_0(r)), \mathbf{assert}(ALW\_EV_0(r))\}$ and $\alpha_2 = \mathbf{assert}(CAN_0(r))$,

the following statements are also added (where $dummy$ is a reserved name):

$$\begin{array}{r}\mathbf{assert}(dummy)\ \mathbf{if}\ \alpha_1(r), \alpha_2(r);\\ \mathbf{retract}(dummy)\ \mathbf{if}\ \alpha_1(r), \alpha_2(r).\end{array}$$

(Module B) The persistent commands PC_{i-1} and the effective commands EC_{i-1} from the previous step are recorded in the knowledge base by means of the corresponding special atoms. These are used to check conditions for the sets PC_i and EC_i.

(B.1) For each atom $\alpha \in \{ASR, ASR\_EV, RET, RET\_EV, CAN\}$, the following statements are added to $\mathcal{U}'$:

$$\begin{array}{l}\mathbf{retract}(\alpha(r))\ \mathbf{if\ not\ assert}(\alpha_0(r))\ [\![\alpha(r)]\!];\\ \mathbf{assert}(\alpha(r))\ \mathbf{if\ assert}(\alpha_0(r)),\ \mathbf{not\ assert}(IGN_0(r)).\end{array}$$

(B.2) For each atom $\beta \in \{ALW, ALW\_EV\}$, add the following statement to $\mathcal{U}'$:

$$\mathbf{assert}(\beta(r))\ \mathbf{if\ not\ retract}(\beta(r))\ [\![\beta(r)]\!].$$

(B.3) The following statements are also added:

$$\begin{array}{rl}\mathbf{assert}(ALW(r))\ \mathbf{if} & \mathbf{assert}(ALW_0(r)),\\ & \mathbf{not\ retract}(ALW(r)),\\ & \mathbf{not\ assert}(IGN_0(r));\\ \mathbf{assert}(ALW\_EV(r))\ \mathbf{if} & \mathbf{assert}(ALW\_EV_0(r)),\\ & \mathbf{not\ retract}(ALW\_EV(r)),\\ & \mathbf{not\ assert}(ALW_0(r)),\\ & \mathbf{not\ assert}(IGN_0(r))\ [\![not\ ALW(r)]\!];\\ \mathbf{retract}(ALW\_EV(r))\ \mathbf{if} & \mathbf{assert}(ALW_0(r)),\ \mathbf{not\ assert}(IGN_0(r));\\ \mathbf{retract}(ALW\_EV(r))\ \mathbf{if} & \mathbf{assert}(CAN_0(r)),\ \mathbf{not\ assert}(IGN_0(r));\\ \mathbf{retract}(ALW(r))\ \mathbf{if} & \mathbf{assert}(CAN_0(r)),\ \mathbf{not\ assert}(IGN_0(r)).\end{array}$$

(Module C) Special statements encode sets of rules to be, respectively, inserted into the knowledge base, activated or deactivated. This amounts to computing P_i:

(C.1) Set Add_i:

$$\begin{array}{l}\mathbf{assert}(r)\ \mathbf{if\ assert}(ASR(r));\\ \mathbf{assert}(r)\ \mathbf{if\ assert}(ASR\_EV(r));\\ \mathbf{assert}(r)\ \mathbf{if\ assert}(ALW(r));\\ \mathbf{assert}(r)\ \mathbf{if\ assert}(ALW\_EV(r)).\end{array}$$

(C.2) Set $Enable_i$:

$$\begin{array}{l}\mathbf{assert}(r)\ \mathbf{if\ not\ assert}(RET(r)),\ \mathbf{not\ assert}(RET\_EV(r))\\ \qquad [\![RET\_EV(r), not\ RET(r), not\ ALW\_EV(r)]\!];\end{array}$$

$$\begin{array}{l}\mathbf{assert}(r)\ \mathbf{if\ not\ assert}(RET(r)),\ \mathbf{not\ assert}(RET\_EV(r)),\\ \qquad \mathbf{not\ assert}(CAN(r))\ [\![RET\_EV(r), not\ RET(r)]\!].\end{array}$$

(C.3) Set $Disable_i$:

$$\begin{aligned}
&\mathbf{retract}(r) \textbf{ if } \mathbf{assert}(RET(r)),\\
&\qquad\qquad \mathbf{not\ assert}(ALW(r)),\ \mathbf{not\ assert}(ALW_EV(r));\\
&\mathbf{retract}(r) \textbf{ if } \mathbf{assert}(RET_EV(r)),\\
&\qquad\qquad \mathbf{not\ assert}(ALW(r)),\ \mathbf{not\ assert}(ALW_EV(r));\\
&\mathbf{retract}(r) \textbf{ if } \mathbf{assert}(CAN(r)),\ \mathbf{not\ assert}(ASR(r)),\\
&\qquad\qquad \mathbf{not\ assert}(ASR_EV(r))\ [\![ALW_EV(r)]\!];\\
&\mathbf{retract}(r) \textbf{ if } \mathbf{not\ assert}(ALW(r)),\ \mathbf{not\ assert}(ASR(r)),\\
&\qquad\qquad \mathbf{not\ assert}(ASR_EV(r)),\ \mathbf{not\ assert}(ALW_EV(r))\\
&\qquad\qquad [\![ASR_EV(r), not\ ASR(r)]\!].
\end{aligned}$$

This rewriting has some advantages. First of all, it gives a new policy which is efficiently computable. The number of rules in $\mathcal{U}'$ is linear in the rules of the original one, and the number of new atoms is limited as well. Furthermore, the new policy is factual if the original policy was factual, and also possible stratifiability (defined in Subsection 4.3 below) of the original policy is preserved. These are, due to complexity properties, desirable properties, as will become clear from the results discussed next.

4.3 Computational complexity

In this section, we address the computational complexity of reasoning about a knowledge state KS in the ground (propositional) case. Observe that the complexity of reasoning strongly depends on the complexity of computing the set Bel, determined by the chosen underlying update semantics. We consider in the following the update answer sets of [11, 15], whereas this also implies that the same results hold for dynamic logic programming [2] as well. For other approaches, e.g., the abductive updates by Inoue and Sakama [30], the results change, as far as the underlying semantics lies at the next level of the polynomial hierarchy. Other results on the complexity of reasoning over a general update framework have been derived in [14].

Definition 9 *An update policy $\mathcal{U}$ is* stratified *if, for all* EPI *statements $cmd(\rho)$* **if** *C_1 $[\![C_2]\!] \in \mathcal{U}$, the associated rules $cmd(\rho) \leftarrow C_1'$ form a locally stratified logic program, where C_1' results from C_1 by replacing the* EPI *declaration* **not** *by the default negation not.*

For stratified $\mathcal{U}$, any $\Pi_i^{\mathcal{U}}$ has at most one answer set, thus the selection function $Sel(\cdot)$ is redundant. Otherwise, the complexity cost of $Sel(\cdot)$ must be taken into account. We assume here that $Sel(\cdot)$ is computable in polynomial time with an NP oracle; e.g., lexicographic preference has this property. If $Sel(\cdot)$ is unknown, we cautiously must consider all possible realizations for $Sel(\cdot)$, which emerge from the different answer sets of the programs $\Pi_i^{\mathcal{U}}$). The following result on reasoning from an update program under update answer set semantics is easily established (cf. the definitions and Lemma 2).

Proposition 5 *Given a ground (or propositional) update program $\boldsymbol{P} = (P_0, \ldots, P_m)$ and a ground rule r', deciding whether $r' \in Bel(P_0, \ldots, P_m)$ is in* coNP*; moreover, if P_0 is stratified and each P_i, $1 \leq i \leq m$, contains only facts, the problem is solvable in polynomial time.*

We can thus establish the following upper complexity bounds.

Lemma 3 *Let Bel be the update answer set semantics. Then, given a ground rule r, a ground knowledge state $KS = \langle KB; E_1, \ldots, E_n\rangle$ and a ground update policy $\mathcal{U}$, deciding whether $r \in Bel(KS)$ is in* P^{NP}, *assuming that* $Sel(\cdot)$ (*which may be part of the input*) *is polynomial-time computable with an* NP *oracle. If* KB *is stratified and* $\mathcal{U}$ *is factual and stratified, then deciding* $r \in Bel(KS)$ *is polynomial.*

Proof. Each U_i and P_i as in Figure 1 can be computed iteratively, where at Step i polynomially many problems $r' \in Bel(P_0, \ldots, P_{i-1})$ must be solved to construct $\Pi_i^{\mathcal{U}}$. From $U_i = Sel(\Pi_i^{\mathcal{U}})$ and previously computed results, P_i can be computed in polynomial time. Since P_i contains at most $|\mathcal{U}|$ rules, Step i is thus feasible in polynomial time with an NP oracle. Therefore, by the hypothesis on $Sel(\cdot)$, $\boldsymbol{P} = (P_0, \ldots, P_n)$ is polynomially computable with an NP oracle, and $r \in Bel(\boldsymbol{P})$ is then decided by Proposition 5 with another oracle call. Thus, deciding whether $r \in Bel(KS)$ is in P^{NP}. If KB is stratified and $\mathcal{U}$ is factual and stratified, $U_i = Sel(\Pi_i^{\mathcal{U}})$ is computable in polynomial time and in $Bel(P_0, \ldots, P_{i-1})$ P_0 is stratified and each P_j, $j \in \{1, \ldots, i-1\}$ contains only facts. Thus, by Proposition 5, the NP oracle can be replaced by a polynomial-time procedure, and each $\Pi_i^{\mathcal{U}}$ is computable in polynomial time. This proves the result. $\Box$

Let us now state precise complexity results.

Theorem 4 (General KB with factual and stratified $\mathcal{U}$) *Given a ground rule* r*, a ground knowledge state* $KS = \langle KB; E_1, \ldots, E_n \rangle$*, and a ground stratified update policy* $\mathcal{U}$*, deciding whether* $r \in Bel(KS)$ *is* P^{NP}*-complete.*

Proof. By Lemma 3, it remains to show hardness. We can encode the problem of finding the last bit (denoted α_n^*) of the lexicographically maximal model of a formula ϕ in CNF, which is well-known to be P^{NP}-hard [28], into a reasoning task in the framework of update policies. Let ϕ be defined by clauses c_j, $j \in \{1, \ldots, m\}$, on variables $x_1, \ldots, x_n$. For our purpose, let us define an initial program KB as consisting of the following (sets of) rules:

(α) $unsat \leftarrow \tau(c_j)\ \ \forall j \in \{1, \ldots, m\}$, where $\tau(c_j)$ is obtained from $\neg c_j$ by replacing each x_i by $not\ \hat{x}_i$ for $i \in \{1, \ldots, n\}$;

(β) $x_i \leftarrow not\ \hat{x}_i$ and $\hat{x}_i \leftarrow not\ x_i, \forall i \in \{1, \ldots, n\}$;

(γ) the fact $x_1 \leftarrow .$

Note that in this way $KB \models unsat$ if and only if ϕ is unsatisfiable with $x_1 = true$ (then, the optimal assignment for x_1 is $x_1^* = false$). In the context of update policies, we consider as events the variables to be set, and the following update statements building up $\mathcal{U}$:

$$\begin{array}{l}\mathbf{retract}(x_i)\ [\![\ unsat, \mathbf{E}: x_i\ ]\!],\ \forall 1 \le i \le n;\\ \mathbf{assert}(x_{i+1})\ [\![\ \mathbf{E}: x_i\ ]\!],\ \forall 1 \le i < n.\end{array}$$

Thus, $\mathcal{U}$ is factual and, trivially, stratified.

Let us then consider $KS = \langle KB; E_1, \ldots, E_n \rangle$, where $E_i = \{x_i\}$. Each event E_i *checks* whether ϕ with the (lexicographically maximal) prefix $x_1^*, \ldots, x_{i-1}^*$ and $x_i = true$ is satisfiable. This is the case if and only if $unsat \notin Bel(KS_{i-1})$. In particular, $x_n \in Bel(KS)$ if and only if the last bit α_n^* of $x_1^*, \ldots, x_n^*$ is $true$, thus proving the P^{NP}-hardness for a general KB and a factual and stratified $\mathcal{U}$. $\Box$

Theorem 5 (Stratified KB' with stratified $\mathcal{U}$) *Given a ground rule* r *and a ground knowledge state* $KS = \langle KB'; E_1, \ldots, E_n \rangle$ *where* KB' *is stratified, and a stratified ground update policy* $\mathcal{U}$*, deciding whether* $r \in Bel(KS)$ *is* P^{NP}*-complete.*

Proof. By Lemma 3, it remains to show hardness. The proof is by a reduction similar to the one in the proof of Theorem 4. However, KB' is stratified; it is obtained from KB (as defined above) by removing clauses $\hat{x}_i \leftarrow not\, x_i$. The first event in the sequence, which we call E_0, serves to add exactly these clauses to the knowledge base (we denote it by $E_0 = \{add\_clauses\}$). The update policy $\mathcal{U}$ contains to this purpose the additional statements $\mathbf{assert}(\hat{x}_i \leftarrow not\, x_i)$ $[\![$ $\mathbf{E}$:*add_clauses* $]\!]$, for each $1 \leq i \leq n$. Thus, after the first evolution step, $KS_1 = \langle KB'; E_0 \rangle$ amounts to the same program as KB as defined above, and P^{NP}-hardness follows for the case of a stratified initial knowledge base with a stratified update policy. □

Theorem 6 (General $\mathcal{U}$ and unknown Sel) *Given a ground rule r, a ground knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ and a general ground policy $\mathcal{U}$, deciding whether $r \in Bel(KS)$ holds cautiously for all selection functions Sel computable in polynomial time with an* NP *oracle is Π^P_2-complete. Hardness holds even if KB is stratified.*

Proof. Let $U_1, \ldots, U_n$ be any sequence of update commands such that U_i is an answer set of $\Pi^{\mathcal{U}}_i$, $1 \leq i \leq n$ and $U_i = U_j$ if $\Pi^{\mathcal{U}}_i = \Pi^{\mathcal{U}}_j$, for all $1 \leq i < j \leq n$. The properties can be checked in polynomial time with an NP oracle, by computing $\Pi^{\mathcal{U}}_i$, $1 \leq i \leq n$, and checking the conditions.

Any such sequence describes the results $U_i = Sel(\Pi^{\mathcal{U}}_i)$ of some selection function Sel computable in polynomial time with an NP oracle, and vice versa each such Sel gives rise to such a sequence. Thus, $r \in Bel(KS)$ can be disproved by guessing and checking a sequence $U_1, \ldots, U_n$ as above such that $r \notin Bel(P_0, \ldots, P_n)$ where $(P_0, \ldots, P_n)$ is the update program constructed from KS and $U_1, \ldots, U_n$ in polynomial time with an NPoracle. Thus, deciding whether $r \notin Bel(KS)$ for some selection function Sel is in $\mathrm{NP}^{\mathrm{NP}} = \Sigma^P_2$. The membership result follows.

For the hardness part, let us consider a QBF $\forall X_1 \exists X_2 \varphi$, where φ is a CNF on atoms $X_1 \cup X_2$. We define KB containing the sets of rules (α) and (β) as defined in the proof of Theorem 4 over $X = X_1 \cup X_2$, plus the fact $sat \leftarrow$. We further define $\mathcal{U}$ as made up of the following sets of statements:

$$\begin{array}{l}
\mathbf{assert}(x_{1,j}) \ \mathbf{if\ not\ assert}(\hat{x}_{1,j}) \ [\![\ \mathbf{E}:guess\ ]\!],\ \forall x_{1,j} \in X_1;\\
\mathbf{assert}(\hat{x}_{1,j}) \ \mathbf{if\ not\ assert}(x_{1,j}) \ [\![\ \mathbf{E}:guess\ ]\!],\ \forall x_{1,j} \in X_1;\\
\mathbf{retract}(sat) \ [\![\ unsat, \mathbf{E}:check\ ]\!].
\end{array}$$

Let then $KS = \langle KB; E_1, E_2 \rangle$, with $E_1 = \{guess\}$ and $E_2 = \{check\}$. The question is then whether $sat \in Bel(KS)$ for all $Sel(\cdot)$. Note that E_1 nondeterministically selects a truth assignment for $x_{1,j}$, thus at the first evolution step $KS_1 = \langle KB, E_1 \rangle$ the program reduces essentially to the CNF $\varphi[\sigma(X_1)]$ with the truth assignment $\sigma(X_1)$ plugged in. Then, the event $E_2 = \{check\}$ tests whether $unsat \in Bel(KS_1)$, which is the case if and only if $\varphi[\sigma(X_1)]$ is unsatisfiable. In this case, sat is removed, else it remains. Thus, $sat \in Bel(KS)$ under any implementation of Sel if and only if for every $\sigma(X_1)$, formula $\varphi[\sigma(X_1)]$ is satisfiable; in other words, if and only if $\forall X_1 \exists X_2 \varphi$ is true. This proves the Π^P_2-hardness for a general KB. As in the previous case, we can start from a stratified subset KB' of KB and add the remaining clauses by means of a preliminary event $E_0 = \{add\_clauses\}$, thus proving that the Π^P_2-hardness also holds for initial stratified KB. □

5 Implementation and Update Agents

It is possible to give a straightforward implementation of our framework for update policies in terms of programs for *update agents*, as explained in the following. Two possible multi-agent scenarios are interesting

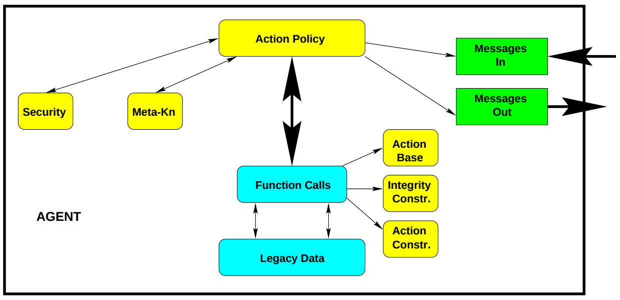

Figure 2: IMPACT Agent Architecture

to our purpose. In the first one, several cooperative agents share a common knowledge base, which has to be dynamically updated by a *knowledge management agent*, which receives inputs from the environment and update information from the single agents. In the second one, each agent in the system has to maintain its own knowledge base, depending on external updates and on system-internal communications. In both cases, the agent has to implement a given update policy, as defined in the corresponding framework. We call such agents *update agents*. We describe how such update agents can be implemented in an existing agent architecture. More specifically, we discuss the realization of such agents in the IMPACT multi-agent system platform [32].

5.1 IMPACT

IMPACT [32], the *Interactive Maryland Platform for Agents Collaborating Together*, is an agent framework which allows for existing code and data sources to be “agentized”. The behavior of an IMPACT agent, i.e., which actions it takes upon a state change, is specified declaratively by a set of rules. The possibility to employ existing implementations, together with the possibility of declarative agent specifications, makes the realization of update agents as IMPACT agents straightforward. Figure 2 shows the general architecture of an IMPACT agent. All agents have the same components, but the *content* of these components can be different, leading to different behaviors and capabilities.

Basically, the behavior of the agent is driven by an *action policy*: on the basis of the content of the message box for the agent and of queries to background data (performed by means of function calls which abstract from the structure both of the data and of the information sources), the actions to be performed are selected under a declarative semantics. Constraints ensure security and integrity of data and behavior. Actions may themselves be changes to available data, posting of messages to other agents, and so on.

Agents are built “on top” of some existing body of code. Thus, to every agent, a set of types can be assigned, which contains all data types or data structures that the agent manipulates. The set of data structures is manipulated by a set of functions that are callable by external programs via *code calls*.

Each agent has access to a message box, together with some function calls for it. At any given point in time, the actual set of objects in the data structures (and message box) managed by the agent constitutes the *state* of the agent. The set of ground code calls which are true in it are identified as an agent’s state $\mathcal{O}$.

The agent can execute a set of *actions*, e.g., reading a message from the message box, executing a request, updating the agent data structures, or even doing nothing. A set of *action rules*, called the *agent program*, specifies the principles under which the agent is operating. These rules specify, using deontic modalities, what the agent may do, must do, may not do, etc.

Each agent program has a *formal semantics* which is defined in semantical structures called *status sets*. In particular, the semantics of an agent is defined in terms of *feasible status sets*. A feasible status set is a set of ground action status atoms, which satisfies various conditions. E.g., it is closed under the rules of the agent program and complies to certain deontic axioms. Additionally, stronger semantical notions than feasible status sets have been introduced for IMPACT agents, namely, *rational status sets* and *reasonable status sets*. A detailed description of further components of an IMPACT agent, which are not relevant for our purposes here, can be found in [32].

5.2 Update agents

The idea is to integrate the features of the declarative IMPACT action language on the one hand, and the declarative EPI formalism for updating knowledge bases on the other hand in a suitable way, leading to a rule-based language for specifying update behavior to a knowledge base. A formal semantics for update policies, given in this language, is inherited from the semantics for agent programs in IMPACT, combined with the semantics of the EPI language. The current implementation covers the basic EPI framework, without extensions (cf. Section 6). This implementation could be easily adapted to realize some extensions and the LUPS semantics as well.

An update policy consists of an IMPACT agent program, which is build over the following agent data structures and actions:

1. Suppose a software package $\mathcal{SP}$ for updating and querying a knowledge base KB is available, and that KB can be accessed through a function $bel()$ which returns the current belief set $Bel(KS)$. Moreover, assume that an event is a finite set of rules and $\mathcal{SP}$ (or another package) has a function $event()$ which lists all rules in the current event. Then, code call atoms, i.e., atoms of the form $in(t, bel())$, $not\_in(t, bel())$, and $in(t, event())$, where t is either a specific rule r or a variable R, can be used to express conditions on the belief set and the event[1].

2. IMPACT actions model the operations of the update language by means of their effects, which are specified in terms of *add lists* and *delete lists*. In particular, the actions $always(R)$, $assert(R)$, $retract(R)$, $cancel(R)$, $always\_event(R)$, $assert\_event(R)$, $retract\_event(R)$, and $ignore(R)$, where R is a rule identifier, are required.

3. An update policy $\mathcal{U}$ is represented as a set of IMPACT action rules of form

$$\mathbf{Do}\, cmd_1(\bar{t}_1) \leftarrow \quad [\neg]\mathbf{Do}\, cmd_2(\bar{t}_2), \ldots, [\neg]\mathbf{Do}\, cmd_n(\bar{t}_n), CC(\mathtt{cond}),$$

   where $cmd_i(\bar{t}_i)$, $1 \leq i \leq n$, are action atoms as introduced above, and $CC(\mathtt{cond})$ is a list of code call atoms expressing conditions on the belief set and the event in $\mathtt{cond}$ as described above. Intuitively, such a rule expresses that the update agent will perform action cmd_1 (with the given arguments), if it also performs actions cmd_i, $1 \leq i \leq n$, (or does not perform those actions, if the action atoms are negated) and if the current belief set and event satisfy $CC(\mathtt{cond})$. E.g., the action rule $\mathbf{Do}\, assert(R) \leftarrow \neg\mathbf{Do}\, ignore(R), in(R, event())$ encodes a simple *incorporate-by-default* policy, i.e., event rules are incorporated in the knowledge base, except if it is explicitly specified that the rule has to be ignored.

Consider the shopping agent from Example 2. Its update policy might be defined as an IMPACT agent program as follows. Assume it contains the incorporate-by-default rule from above, as well as:

[1]The available current implementation covers the propositional case, i.e., rule variables are not allowed.

$$\begin{aligned}
&\mathbf{Do}\, always(sale(S) \leftarrow date(T)) \leftarrow in(S, shops()),\ in(T, dates()),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{Do}\, assert(sale(S) \leftarrow date(T));\\
&\mathbf{Do}\, cancel(sale(S) \leftarrow date(T)) \leftarrow in(date(T), bel()), in(date(T'), event()),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad in(T, dates()), in(T', dates()), T \neq T',\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad in(S, shops());\\
&\mathbf{Do}\, retract(sale(S) \leftarrow date(T)) \leftarrow in(date(T), bel()), in(date(T'), event()),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad T \neq T', in(T, dates()), in(T', dates()),\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad in(S, shops()).
\end{aligned}$$

Note that we assumed the existence of two code calls, $shops()$ and $dates()$, which we used for grounding, i.e., they return all possible assignments for shops and time stamps. Furthermore, $\mathcal{U}$ includes also the following rules:

$$\begin{aligned}
&\mathbf{Do}\, retract(date(T)) \leftarrow in(date(T), bel()), in(date(T'), event()),\\
&\qquad\qquad\qquad\qquad\quad T \neq T', in(T, dates()), in(T', dates());\\
&\mathbf{Do}\, ignore(sale(s_1)) \leftarrow in(sale(s_1), event());\\
&\mathbf{Do}\, ignore(sale(s_1) \leftarrow date(T)) \leftarrow in(sale(s_1) \leftarrow date(T), event()), in(T, dates()).
\end{aligned}$$

Intuitively, an update agent follows its update policy by calculating a new (reasonable) status set on each event it receives, and by executing exactly those (ground) actions $cmd(\bar{t})$ such that the action status atom $\mathbf{Do}\, cmd(\bar{t})$ is in the status set. For instance, if the update agent given above were triggered by the event $E_1 = \{sale(s_2), date(1)\}$, its reasonable status set would contain the action status atoms $assert(sale(s_2))$, $assert(date(1))$, and $retract(date(0))$. By the agent performing the corresponding actions, the knowledge base would be updated accordingly.

Obviously, not all actions are compatible with each other, e.g., the actions $assert(r)$ and $retract(r)$ should not be executed simultaneously. This can be handled naturally in IMPACT by formulating action constraints, which disable joint execution of actions, subject to an optional code call condition.

Another benefit that comes for free by using IMPACT programs to specify policies for update agents is the availability of deontic modalities. By allowing for arbitrary modalities in the action rules of an update policy, its expressiveness can be further enhanced. This would make it possible to state under which conditions the assertion or retraction of a rule is possible, allowed, forbidden, etc.

6 Extensions

The EPI framework can be extended with additional features, offering valuable capabilities for information agents. Observe that such straightforward extensions do not require new semantic definitions, they just fit in the usual ones, providing the evaluation of the condition part in each statement is suitably adapted.

6.1 Access to the structure of rules

The general form of an update statement is like in the basic framework, but the *structure* of rules is accessible: conditions of form $\rho := H(\rho') \leftarrow A, B^+(\rho'')$ (*extended conditions*) are allowed, where ρ, ρ', and ρ'' are rule variables, A is a variable for a literal, and $H(\rho')$, $B^+(\rho'')$ denote the head and positive body of rules ρ', and ρ'', respectively (cf. also Table 2). This possibility can be very useful when expressing, e.g., retraction of a whole set of rules satisfying some condition. In the context of information agents, assuming, e.g., that a certain information source s_1 is and will be no more available (signaled by the event $out(s_1) \leftarrow$), then the following update statements can serve to retract all rules of the knowledge base which would start a query to or require the availability of this source:

$$\mathbf{retract}(\rho)\ [\![(\rho := query(s_1) \leftarrow \#),\ \mathbf{E} : out(s_1) \leftarrow]\!];$$
$$\mathbf{retract}(\rho)\ [\![(\rho : \# \leftarrow up(s_1), \#),\ \mathbf{E} : out(s_1) \leftarrow]\!].$$

where $\#$ denotes *don't care*. In a similar way, the update policy from Example 2 can be simplified, by substituting the last two rules with the single update statement $\mathbf{ignore}(\rho)\ [\![(\rho := sale(s_1) \leftarrow \#)]\!]$. Additionally, atoms $\mathbf{in}(\cdot)$ and $\mathbf{in_a}(\cdot)$ over rules (where **in** and $\mathbf{in_a}$ are reserved predicates) can be evaluated over the current knowledge base (in the first part of the "$[\![\ldots]\!]$" condition). Intuitively, these atoms record if a certain rule r is physically present respectively activated in the current knowledge base; the second condition is checked by means of the corresponding atom $on(n_r)$ if it is in the belief set of a given knowledge state, after the compilation tr. We denote by $AR(KS_i)$, for each Step i, the set of activated rules in the knowledge base at that step, and say that $Bel(KS) \models \mathbf{in_a}(r)$ for some rule r iff $Bel(KS) \models on(n_r)$.

Again, this feature can be useful in the situation of Example 2, adding to the update policy the following statement:

$$\mathbf{ignore}(sale(S) \leftarrow \#)\ [\![\mathbf{in_a}(sale(pref\_shop) \leftarrow date(T)), \mathbf{E} : date(T)]\!].$$

In this way, possible sales in other shops are ignored, as soon as a sale is running in the favorite shop. Or, reconsidering Example 3, the extended conditions could be useful to express something like the following: suppose in events rules are allowed, which possibly specify the classification of new types of messages, i.e., rules with head $type(M, t_{new})$ for some new type attribute t_{new}, and corresponding conditions over message attributes in the body. Then the update policy could contain the following statements:

$$\begin{array}{l}\mathbf{assert}(r_{new})\ [\![\mathbf{E} : r_{new} := type(M, t_{new}) \leftarrow \#]\!];\\ \mathbf{retract}(r_3)\ \mathbf{if\ assert}(r_{new});\\ \mathbf{assert}(r' := \mathbf{H}(r_3) \leftarrow \mathbf{B}(r_3), not\, \mathbf{H}(r_{new}))\ \mathbf{if\ retract}(r_3).\end{array}$$

The result is not only the incorporation of the new rule, but also the insertion of an additional default negated atom in the old rule r_3, assigning $other$ as type to a message if none of available types applies.

The syntax of the extended language is shown in Table 2. The introduction of the extended conditions requires a corresponding extension of the semantics of EPI programs, which anyway affects only the evaluation part resulting in the executable update commands. In particular, the notion of $\mathcal{G}(\mathcal{U})^i$, the set of reduced update statements at Step i, has to be redefined for the new syntax. In fact, it is now possible for conditions c_i or r_j ($i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, m\}$) in $\mathcal{G}(\mathcal{U})^i$ to describe and access the structure of rules. For each update statement involving conditions of this form, all possible instantiations of rule variables (correct as to the requirements over corresponding head and body) have to be considered, possibly generating multiple grounded statements in $\mathcal{G}(\mathcal{U})^i$.

First of all, safety requirements over rule variables have to be checked. To this purpose, let us consider a relation $\subset_{\mathcal{U}}$ between rule variables, such that $R_1 \subset_{\mathcal{U}} R_2$ if and only if R_1 occurs in $\mathcal{U}$ in the right-hand side of an extended condition describing the structure of rule R_2. The safety of the policy $\mathcal{U}$ is ensured, if the relation $\subset_{\mathcal{U}}$ is a partial order.

Given an update policy, $\mathcal{U}$, and a knowledge state, $KS = \langle KB; E_1, \ldots, E_n \rangle$, the set $\mathcal{G}(\mathcal{U})^i$ is the result of the following reduction. Let $Bel(KS_{i-1})$ be the belief set at Step $i-1$ and E the new event. Statements in $\mathcal{U}$ have the form

$$cmd(\sigma)\ \mathbf{if}\ C_1\ [\![c_1, \ldots, c_l, \mathbf{E} : r_1, \ldots, r_m]\!],$$

$$
\begin{array}{lll}
\langle stat\rangle & ::= & \langle c\_name\rangle(\langle r\_info\rangle)\ [\ \mathbf{if}\ \langle if\_cond\rangle\ ]\ [\ [\![\ \langle wh\_cond\rangle\ ]\!]\ ]\ ; \\
\langle c\_name\rangle & ::= & \mathbf{assert\_[event]} \mid \mathbf{retract\_[event]} \mid \mathbf{always[\_event]} \mid \mathbf{cancel} \mid \mathbf{ignore}\ ; \\
\langle r\_info\rangle & ::= & \langle rule\rangle \mid \langle r\_id\rangle \mid \langle r\_name\rangle{:=}\langle r\_str\rangle; \\
\langle r\_id\rangle & ::= & \langle r\_name\rangle \mid \langle r\_var\rangle; \\
\langle if\_cond\rangle & ::= & \langle cond1\rangle \mid \langle cond1\rangle\ ,\ \langle if\_cond\rangle; \\
\langle wh\_cond\rangle & ::= & \langle kb\_conds\rangle \mid \mathbf{E} : \langle ev\_conds\rangle \mid \langle kb\_conds\rangle\ ,\ \mathbf{E} : \langle ev\_conds\rangle; \\
\langle cond1\rangle & ::= & [\mathbf{not}]\ \langle c\_name\rangle(\langle r\_info\rangle)\ ; \\
\langle lit\_id\rangle & ::= & \langle literal\rangle \mid \langle lit\_var\rangle; \\
\langle r\_str\rangle & ::= & \langle head\rangle{\leftarrow}\langle body\rangle; \\
\langle body\rangle & ::= & \langle b\_cond\rangle,\langle body\rangle \mid \epsilon\ ; \\
\langle head\rangle & ::= & \langle lit\_id\rangle \mid \# \mid \mathbf{H}(\langle r\_id\rangle) \mid \epsilon; \\
\langle b\_cond\rangle & ::= & \# \mid [not]\langle lit\_id\rangle \mid \mathbf{B}(\langle r\_id\rangle) \mid \mathbf{B}^{+}(\langle r\_id\rangle) \mid \mathbf{B}^{-}(\langle r\_id\rangle) \mid \mathbf{H}(\langle r\_id\rangle); \\
\langle kb\_conds\rangle & ::= & \langle kb\_cond\rangle \mid \langle kb\_cond\rangle\ ,\ \langle kb\_conds\rangle; \\
\langle kb\_cond\rangle & ::= & \langle r\_info\rangle \mid [not]\ \mathbf{in}[_\mathbf{a}](\langle r\_info\rangle) \mid [not]\langle lit\_id\rangle; \\
\langle ev\_conds\rangle & ::= & \langle ev\_cond\rangle \mid \langle ev\_cond\rangle\ ,\ \langle ev\_conds\rangle; \\
\langle ev\_cond\rangle & ::= & \langle lit\_id\rangle \mid \langle r\_info\rangle;
\end{array}
$$

Table 2: EPI extended syntax

where each $c_1, \ldots, c_l$ and $r_1, \ldots, r_m$ can possibly be an extended condition. Then $\mathcal{G}(\mathcal{U})^i$ is obtained performing the following steps:

1. For each statement s and for each r_i in s, $1 \leq i \leq m$, such that r_i is not an extended condition, if $r_i \in E$ then delete r_i from s. Otherwise, delete s.

2. For each statement s and for each extended condition r_i in s of form $\rho : L \leftarrow L_1, \ldots, L_k$, if no rule $r' \in E$ exists such that $H(r') = L$ and $B(r') = \{L_1, \ldots, L_k\}$ (if $L = \#$ or $L_1, \ldots, L_k = \#$, then the head, resp. body, of a rule needs not be considered), then delete s. Otherwise, if such a rule r' exists, substitute each occurrence of the variable ρ in $c_1, \ldots, c_l$, σ, and C_1 of the same statement s with the name of the rule, $n_{r'}$, and then delete r_i from s.

3. For each statement s and for each c_j in s, $1 \leq j \leq l$, such that c_j is not an extended condition, if $Bel(KS_{i-1}) \models c_j$ then delete c_j from s. Otherwise, delete s.

4. For each statement s and for each extended condition c_j in s of form $\rho : L \leftarrow L_1, \ldots, L_k$, if no rule r' exists such that $H(r') = L$ and $B(r') = \{L_1, \ldots, L_k\}$ (if $L = \#$ or $L_1, \ldots, L_k = \#$, then the head, resp. body, of a rule needs not to be considered) and such that $Bel(KS_{i-1}) \models r'$, then delete s. Otherwise, if such a rule r' exists, substitute each occurrence of the variable ρ in σ and in C_1 of the same statement s with the name of the rule, $n_{r'}$, and then delete c_j from s. We obtain a set of statements s_h, $1 \leq h \leq p$, of form

$$cmd_{h,0}(\sigma_{h,0})\ \mathbf{if}\ [\mathbf{not}]\ cmd_{h,1}(\sigma_{h,1}), \ldots, [\mathbf{not}]\ cmd_{h,n}(\sigma_{h,n}),$$

where each $\sigma_{h,1}, \ldots, \sigma_{h,n}$ is either a rule, or a rule name, or an extended condition of the form $\rho : L \leftarrow L_1, \ldots, L_k$.

5. For each statement s_h, $1 \leq h \leq p$, for each extended condition in $cmd_{h,i}(\sigma_{h,i})$ of s_h, $1 \leq i \leq n$, of form $\rho : L \leftarrow L_1, \ldots, L_k$, if there is no statement $s_{h'}$, $1 \leq h' \leq p$ and $h' \neq h$, such that $cmd_{h',0} = cmd_{h,i}$ and such that $\sigma_{h',0}$ is a rule (name) with $H(\sigma_{h',0}) = L$ and $B(\sigma_{h',0}) = \{L_1, \ldots, L_k\}$ (if $L = \#$ or $L_1, \ldots, L_k = \#$, then the head, resp. body, of a rule needs not to be considered), then delete s_h. Otherwise, if such a statement $s_{h'}$ exists, substitute each occurrence of the variable ρ in s_h with the name of the rule in $\sigma_{h',0}$.

At the end of this procedure, we obtain a set of statements of form

$$cmd(\sigma) \textbf{ if } [\textbf{not}]\ cmd_1(\sigma_1), \ldots, [\textbf{not}]\ cmd_n(\sigma_n),$$

where each σ, σ_i is a rule or a rule name. This set is $\mathcal{G}(\mathcal{U})^i$ for an update policy $\mathcal{U}$ over the extended language. The semantics is as in the basic framework.

6.2 Conditions over the past

An interesting feature for several application domains is the possibility for updates to depend not only on the current event and belief set, but also on past belief sets (cf., e.g., [5]). This can be realized in essentially two ways, one requiring an extension of the language, and one including ad hoc statements in the update policy, with special atoms (reserved predicates) which serve to record the past. The two methods are briefly described in the following. Similar mechanisms can be thought of in order to allow queries over the content of past events.

- **Extending the language:** Dependency over the content of past belief sets can be obtained by means of two (similar) extensions of the language. In both cases, special constructs of form "$\mathbf{Past}$:" are allowed inside "$[\![\ldots]\!]$", which intuitively *check* the content of the belief set in past states. Two forms of this construct can be thought of. Let us consider the i-th step of the evolution. In the first form (more generic), conditions like $\mathbf{Past}$:r for some rule r are allowed, and are satisfied iff there is some j, $0 \leq j < i$, such that $Bel(KS_j) \models r$. In the second form (more specific), the construct has the form $\mathbf{Past}$:(j, r) for some rule r and some $0 \leq j < i$, and it is satisfied iff $Bel(KS_j) \models r$. These two alternatives are suitable for different modeling tasks, the one checking if a rule has been in the belief set at any state, the other one checking a given particular step in the evolution.

- **Ad hoc update policy:** With this approach several solutions are possible as well, again addressing different modeling tasks. The idea is to extend the update policy with one of the following special sets of update statements, based on new reserved atoms of the form $past(\cdot)$, the choice depending on what kind of past conditions are of interest to the problem in question:

  1. $\mathbf{assert}(past(n_r) \leftarrow)\ [\![\ on(n_r)\ ]\!]$, where n_r is a variable over rule names; in this way, rules which have been in the belief set in some past state are recorded, and *queries* on the presence of such atoms in the belief set are always possible (in form of conditions checking at Step i the content of the belief set of Step $i-1$).
  2. $\mathbf{retract}(past(n_r) \leftarrow)\ [\![\ past(n_r)\ ]\!]$ and $\mathbf{assert}(past(n_r) \leftarrow)\ [\![\ on(n_r)\ ]\!]$, where n_r is a variable over rule names; in this way, only one additional state at a time is available to the *memory* of the agent, that is, executing Step i the content of the belief set at Step $i-1$ (as usual) and at Step $i-2$ can be queried.

3. $\mathbf{assert}(past(1, n_r) \leftarrow)$ $[\![\ on(n_r)\ ]\!]$ together with the retraction command $\mathbf{retract}(past(1, n_r) \leftarrow)$ $[\![\ past(1, n_r), not\ on(n_r)\ ]\!]$ and, for $1 \leq j \leq k-1$ (k pre-defined), the commands:

$$\mathbf{assert}(past(j+1, n_r) \leftarrow)\ [\![\ past(j, n_r)\ ]\!];$$
$$\mathbf{retract}(past(j+1, n_r) \leftarrow)\ [\![\ past(j+1, n_r), not\, past(j, n_r)\ ]\!].$$

These statements make recording and querying the content of the belief sets for the last k steps possible (observe that checking at Step i the presence of the atom $past(l, n_r)$ means checking if $Bel(KS_{i-1-l}) \models on(n_r)$).

Returning to Example 3, and supposing to make use of the first of the above two solutions, i.e., extending the language, this feature could be useful to express statements such as

$$\mathbf{assert}(system\_date\_error)\ [\![\mathbf{Past}\ {:}today(D), \mathbf{E}\ {:}today(D)]\!],$$

which raises an exception if a date is repeated.

7 Related Work

7.1 LUPS

Alferes *et al.* [3] proposed a language called LUPS for update specifications, which comprises statements of form

$$\texttt{cmd}\ rule\ \texttt{when}\ cond, \tag{2}$$

where `cmd` is one of `assert`[`_event`], `retract`[`_event`], `always`[`_event`], or `cancel`, $rule = L \leftarrow L_1, \ldots, L_k$, and $cond = L_{k+1}, \ldots, L_m$ over the original alphabet. This language can be embedded into our framework, that is, an update policy $\mathcal{U}_{LUPS}$ can be defined which, starting from an initial empty knowledge base KB and receiving as events LUPS programs $LU_1, \ldots, LU_n$, acts as a *meta-interpreter* for the corresponding LUPS sequence and produces the intended knowledge evolution.

By this point of view, the following differences between the LUPS and the EPI semantics have to be considered and managed by the meta-interpreter (for a description of the LUPS syntax and semantics, cf. [3]):

1. `retract` and `retract event` have in LUPS not only the effect of retracting a given rule, but also of deleting possible persistent commands. A LUPS `retract` [`event`] corresponds thus in EPI to a **retract**[**_event**] plus a **cancel** for the same rule;

2. Inconsistencies may occur in a sequence of LUPS programs due to the combined effects of commands in successive updates, which are not managed or avoided by the transformation. The following pairs of consecutive commands may generate such a conflict at Step i:

   (a) (`assert event` $R \in LU_{i-1}$ or `always event` $R \in LU_{i-1}$) together with (`assert` [`event`] $R \in LU_i$ or `always` [`event`] $R \in LU_i$);

   (b) `retract event` $R \in LU_{i-1}$ and `retract` [`event`] $R \in LU_i$.

Condition 2 is better explained with an example: if assert event $R \in LU_{i-1}$, the transformation for LUPS will activate rule R at Step $i-1$ (as tr does for EPI); at Step i, it would then attempt to deactivate R, by means of a negative control atom for it ($\neg on(n_R)$ in our notation), as far as the rule R is not supposed to persist by inertia. But if LU_i contains a new command assert R, this new assertion should be stronger than a deactivation simulating non-inertial effects; nevertheless, the transformation for LUPS performs no check and simply inserts at Step i the activation atom ($on(n_R)$ in our notation). Thus, a conflict at Step i arises, making the whole knowledge inconsistent.

These differences have to be accounted for and simulated by the meta-interpreter, i.e., by $\mathcal{U}_{LUPS}$. Let us suppose that LUPS statements of form (2) are slightly rewritten to rules cmd $(rule) \leftarrow cond$.

The update policy $\mathcal{U}_{LUPS}$ which acts as meta-interpreter can then be made up of the following groups of update statements:

1. Basic statements:

$$\alpha(rule)[\![cond, \mathbf{E} : \alpha(rule) \leftarrow cond\,]\!],$$

   where $\alpha \in \{assert[\_event], retract[\_event], cancel, always[\_event]\}$. This set of statements simply translate LUPS into EPI commands with a one-to-one correspondence, provided that the condition part of the LUPS rule is satisfied in the belief set.

2. Additional statements to express the effect of retract[event] commands over persistent commands:

$$\mathbf{cancel}(rule)[\![cond, \mathbf{E} : \texttt{retract}\ [\texttt{event}](rule) \leftarrow cond]\!].$$

   These statements simply express the additional effect of LUPS retract commands, provided the corresponding condition is satisfied.

3. Statements raising a conflict by successive updates, where $AE(\cdot)$ is a reserved predicate:

$$\mathbf{assert\_event}(AE(n_{rule}))[\![cond, \mathbf{E} : \alpha(rule) \leftarrow cond]\!],$$

   where $\alpha \in \{$assert event, always event$\}$; and

$$\mathbf{retract}(rule)[\![cond, AE(n_{rule}), \mathbf{E} : \alpha(rule) \leftarrow cond]\!],$$

   where $\alpha \in \{$assert [event], always [event]$\}$. As far as the transformation tr of our framework avoids conflicts between successive updates by giving preference to new information, a conflict has to be artificially generated. This is done by means of special atoms which are temporarily added to the knowledge base, and through additional rules containing special atoms, thus not firing in other situations, which generate a conflicting pair of commands for the given rule.

4. Statements raising a conflict by successive updates, where $RE(\cdot)$ is a reserved predicate:

$$\mathbf{assert\_event}(RE(n_{rule}))[\![cond, \mathbf{E} : \texttt{retract event}(rule) \leftarrow cond]\!],$$
$$\mathbf{retract}(rule)[\![cond, RE(n_{rule}), \mathbf{E} : \alpha(rule) \leftarrow cond]\!],$$

   where $\alpha =$ retract [event]. These statements work similar as the previous ones.

The application of tr to the knowledge state $KS = \langle \emptyset; LU_1, \ldots, LU_n \rangle$ under the update policy $\mathcal{U}_{LUPS}$ provides the intended result, i.e., the sequence of logic programs resulting from applying the LUPS sequence is the same one as obtained by application of $\mathcal{U}_{LUPS}$ in our framework. This can be seen by applying the corresponding semantic transformation, but the detailed argumentation is very involved due to the complex operational definition.

In [23], a modified version of LUPS, called LUPS*, is proposed, which extends and modifies the syntax and semantics of the language. Basically, the changes amount to two main issues. First, the handling of non-inertial commands is modified, in order to come to more intuitive behavior in some cases (e.g., by empty updates), as the authors argue, and to ensure that the language is *stable* with respect to non-inertial commands; to this purpose, special atoms are introduced which signal at which step a non-inertial command holds, and used to compute the interaction between inertial and non-inertial commands. We do not address this issue here. We only observe that EPI has, by this point of view, the same behavior as LUPS, and that what is more or less intuitive strongly depends on the examples one chooses and on the way in which knowledge is encoded.

Moreover, an additional kind of persistent command is introduced, allowing to define persistent retraction of a rule, i.e., the analogue of the `always` and `always event` commands for retraction. Of course, the transformation is adapted accordingly. This can be done, if desired, for EPI as well, introducing the necessary changes in tr. Anyway, it does not further affect semantics and properties and, as far as the existence of an expressive core has been shown, it also has no effects on the expressivity of the language.

7.2 $\mathcal{PDL}$

Motivated by applications in the domain of network managements, a general purpose language for describing policies called $\mathcal{PDL}$ has been introduced in [26], and was further elaborated and studied in [10, 9, 31]. $\mathcal{PDL}$ has been strongly influenced by traditional action description languages and composite temporal event languages, and offers the possibility of specifying actions for reactive behavior in response to sequences and complex combinations of events. A $\mathcal{PDL}$ policy description is a collection of rules of forms e_1 **causes** a **if** c and e_1 **triggers** e_2 **if** c, which intuitively state that the occurrence of the event e_1 entails that the action a is taken resp. event e_2 is generated provided that condition c holds. Such policies are given a declarative semantics which associates with a history of sets of primitive events (called *epochs*) a history of sets of actions to be taken. Roughly, epochs play in $\mathcal{PDL}$ the role of events in EPI, and actions correspond to update commands.

Despite some similarities, $\mathcal{PDL}$ and EPI actually differ in several respects.

First of all, $\mathcal{PDL}$ is much more action-oriented than EPI, and takes general user-defined actions into account. Contrariwise, EPI employs a predefined set of actions for modifying the knowledge base. The generality of $\mathcal{PDL}$ leads to the necessity of conflict management between actions. This problem has been extensively addressed for $\mathcal{PDL}$ by the introduction of an additional control layer, called *policy monitor* [10], which tackles conflict detection and removal. In the EPI framework, as pointed out earlier, some (restricted) form of conflict avoidance is *implicitly* performed by the transformation tr; further conflict handling or action restriction may be hard-coded in the update policy, but no extra control level over the actions of the update policy exists.

Different from $\mathcal{PDL}$'s rich event ontology and structure, EPI employs only a single event which signals the arrival of new information. This, of course, permits a much simpler definition of the semantics.

On the other hand, rules in EPI are more expressive than rules in $\mathcal{PDL}$ with respect to action execution, in the sense that they allow to refer to the execution and non-execution of other actions as a pre-condition for the execution of an action. Thus, as for decision making, EPI provides a more elaborated language for modeling a deliberative agent than $\mathcal{PDL}$, which models simple non-reflective reactive behavior.

Finally, $\mathcal{PDL}$ pays no special attention to an internal state or knowledge base of an agent, but to properties of events and event sequences, which play a prominent role in rule conditions. In our EPI language, the current belief set, which contains facts and rules derived from the current knowledge base, is a fundamental part of the input to the update policy.

7.3 Access control policies

In [32, 7], the problem of how to derive a main access control policy combining in different ways several partial ones is addressed. The single policies could, e.g., be generated by different authorities regulating access to (areas of) the intranet of an institution. To solve conflicts and to solve some inconsistencies, an algebraic approach is proposed, which allows the use of several algebra-like operators realizing combination of the single policies in different ways. It is obviously assumed that a central authority can decide in which way the set of available policies has to be composed.

Though the chosen approach is modeled on the particular application domain and is not directly suitable for extension to areas such as knowledge-base update, it is generally a good question if and how different policies could be combined. A first solution would be adding a control level to the stratified framework of EPI, but this would cause additional complexity and intractability even for the most simple reasoning tasks.

7.4 Meta-rules in active databases

In [21], the use of meta-rules is discussed, in order to solve potential conflicts between active rules. The idea is to introduce a set of declarative meta-rules which together control the interaction and the execution order of active rules.

Meta-rules may express (i) *positive requirements* over the execution of rules, i.e., if a rule is to be executed, then some other rule has to be executed too; (ii) *disabling* conditions, i.e., the execution of a rule prevents the execution of another one; (iii) *preference* information, i.e., a rule is given priority over another one; (iv) *scheduling* criteria, i.e., control information about the relative order in which rules are to be executed.

In the framework it is also possible to perform some static reasoning tasks, i.e., to determine if a rule will ever execute, if two rules can be executed together, whether the existence of a unique execution set is guaranteed.

Though not directly comparable, some features of the approach could be simulated by a suitable EPI policy, e.g., positive and negative requirements. The treatment of preference and scheduling information would require some careful extension, as far as explicit priority handling is not available in our framework.

8 Conclusion

In this paper, we discussed a generic framework for specifying general update policies for nonmonotonic knowledge bases represented as logic programs containing the negation-as-failure operator. The framework is realized in terms of a layered approach and addresses the issue of strategic update behavior, i.e., which change should be invoked in reaction to new information. This topic is arguably central for the realization of flexible information systems, providing access to a set of heterogenous, dynamic data sources.

Our framework is generic in that it can be instantiated with different underlying update semantics for (sequences of) logic programs. For defining concrete update policies, we introduced a declarative specification language, EPI, which extends previous work by several additional features which are important for modeling dynamic agents. Among these features, the language allows a flexible handling of external events, making it possible that update actions depend on the content of the incoming information. The relevance of the proposed language concerning the realization of adaptive reasoning agents is further strengthened by the fact that EPI allows a direct implementation within the agent platform IMPACT [32].

An important aspect not discussed here concerns temporal reasoning over an evolving knowledge base maintained by an update policy. For instance, it would be important to know whether a fact is always true in a knowledge base, or whether it is never true. Analogous issues, called *maintenance* and *avoidance*, have recently been studied in the agent community [34]. Other properties may involve temporal relationships such as whether $message\_to(tom)$ is true in the given knowledge base at some point, meaning that a message should be sent to Tom, then $sent\_message\_to(tom)$ will become true in the evolving knowledge base at some point, representing that a message to Tom was sent. A suitable logical language for expressing properties of this form has been defined in [14], where the complexity of its associated reasoning problem has also been analyzed.

Possible issues for further research include a coupling of the current update framework with planning or learning modules, providing enhanced reasoning capabilities for dynamic agents. Such additional features would be a further step towards a more comprehensive architecture for building an information infrastructure for dealing with dynamic, heterogenous information sources.

References

[1] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic logic programming. In *Proc. 6th Int'l Conf. on Principles of Knowledge Representation and Reasoning* (*KR'98*), pp. 98–109. Morgan Kauf., 1998.

[2] J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *J. Logic Programming*, 45(1-3):43–70, 2000.

[3] J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS - A language for updating logic programs. In *Proc. LPNMR'99*, LNAI 1730, pp. 162–176. Springer, 1999.

[4] J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS - A language for updating logic programs. *Artificial Intelligence*, in press, 2002.

[5] C. Baral, J. Llopis, J. Lobo, and G. Mendez. Temporal logic and reasoning about actions. In *Common Sense 96*, 1996.

[6] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. A modular approach to composing access control policies. In *Proc. 7th ACM Conf. on Computer and Communications Security (CCS 2000)*, pp. 164–173, 2000.

[7] P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An algebra for composing access control policies. *ACM Trans. on Information and System Security*, 5(1):1–35, 2002.

[8] F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In *Proc. ICLP'99*, pp. 79–93. MIT Press, 1999.

[9] J. Chomicki and J. Lobo. Monitors for history-based policies. In *Proc. Int'l Workshop on Policies for Distributed Systems and Networks (POLICY 2001)*, LNCS 1995, pp. 57–72. Springer, 2001.

[10] J. Chomicki, J. Lobo, and S. Naqvi. A logic programming approach to conflict resolution in policy management. in *Proc. KR 2000*, pp. 121–132. Morgan Kaufmann, 2000.

[11] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Proc. JELIA 2000*, LNAI 1919, pp. 2–20. Springer, 2000.

[12] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. in *Proc. 17th Int'l Joint Conf. on Artificial Intelligence (IJCAI-01)*, pp. 649–654. Morgan Kauf., 2001.

[13] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Declarative knowledge updates through agents. In *Proc. Symposium on Adaptive Agents and Multi-Agent Systems (AISB'01), York, UK, 22-23 March 2001*, pp. 79–84, UK, 2001. The Society for the Study of Artificial Intelligence and the Simulation of Behaviour (AISB). ISBN 1902956170.

[14] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Reasoning about evolving nonmonotonic knowledge bases. In *Proc. 8th Int'l Conf. on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2001)*, LNCS 2250, pp. 407–421. Springer, 2001.

[15] T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of update sequences based on causal rejection. *Theory and Practice of Logic Programming*, 2002. To appear. See also Tech. Rep. INFSYS RR-1843-00-08, Institut für Informationssysteme, TU Wien.

[16] T. Eiter and G. Gottlob. On the complexity of propositional knowledge base revision, updates, and counterfactuals. *Art. Intell.*, 57(2-3):227–270, 1992.

[17] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive datalog. *ACM Trans. on Database Systems*, 22(3):364–418, 1997.

[18] R. Fagin, J. D. Ullman, and M. Y. Vardi. On the semantics of updates in databases. In *Proc. PODS-83*, pp. 352–365. ACM Press, 1983.

[19] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. ICSLP'88*, pp. 1070–1080. MIT Press, 1988.

[20] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3-4):365–386, 1991.

[21] H. Jagadish, A. Mendelzon, and I. Mumick. Managing rule conflicts in active databases. *J. of Computer & System Sciences*, 58(1):13–28, 1999.

[22] H. Katsuno and A. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proc. KR'91*, pp. 387–394. Morgan Kauf., 1991.

[23] J. Leite. A modified semantics for LUPS. In *Proc. 10th Portuguese Conf. on Artificial Intelligence (EPIA 2001)*, LNCS 2258, pp. 261–275. Springer, 2001.

[24] J. Leite and L. Pereira. Generalizing updates: From models to programs. In *Logic Programming and Knowledge Representation, Third Int'l Workshop* (*LPKR'97*), LNCS/LNAI 1471, pp. 224–246. Springer, 1998.

[25] V. Lifschitz and H. Turner. Splitting a logic program. In *Proc. ICLP'94*, pp. 23–38. MIT Press, 1994.

[26] J. Lobo, R. Bhatia, and S. Naqvi. A policy description language. In *Proc. AAAI/IAAI'99*, pp. 291–298. MIT Press, 1999.

[27] V. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science*, 190(2):241–277, 1998.

[28] C. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[29] G. Sabbatini. *Declarative Logic Programming for Representing Evolving Nonmonotonic Knowledge Bases*. Doctoral dissertation, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, 2002.

[30] C. Sakama and K. Inoue. Updating extended logic programs through abduction. In *Proc. LPNMR'99*, LNCS/LNAI 1730, pp. 147–161. Springer, 1999.

[31] T. Son and J. Lobo. Reasoning about policies using logic programs. In *Proc. AAAI 2001 Spring Symp. on Answer Set Programming: (Workshop Technical Report SS-01-01)*, pp. 210–216. AAAI Press, 2001.

[32] V.S. Subrahmanian, J. Dix, T. Eiter, P. Bonatti, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.

[33] M. Winslett. *Updating Logical Databases*. Cambridge University Press, 1990.

[34] M. Wooldridge. The computational complexity of agent design problems. In *Proc. 4th Int'l Conf. on Multi-Agent Systems (ICMAS 2000)*. IEEE Press, 2000.

[35] Y. Zhang and N. Foo. Updating logic programs. In *Proc. ECAI'98*, pp. 403–407. Wiley, 1998.