Specifying Update Policies for Nonmonotonic Knowledge Bases

Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits

Institut und Ludwig Wittgenstein Labor für Informationssysteme,
Technische Universität Wien, Favoritenstraße 9–11, A-1040 Wien, Austria
{eiter,michael,giuliana,tompits}@kr.tuwien.ac.at

Abstract. Recently, several approaches for updating knowledge bases represented as logic programs have been proposed. We present a generic framework for declarative specifications of update policies, which is built upon such approaches. It extends the LUPS language for update specifications, and incorporates the notion of events into the framework. An update policy allows an agent to flexibly react upon new information, arriving as an event, and perform suitable changes of its knowledge base. The framework compiles update policies to logic programs by means of generic translations, and can be instantiated in terms of different concrete update approaches. It thus provides a flexible tool for designing adaptive reasoning agents.

1 Introduction

Updating knowledge bases is an important issue for the realization of intelligent agents, since, in general, an agent is situated in a changing environment, and has to adjust its knowledge base when new information is available. While for classical knowledge bases this issue has been well-studied, approaches to update nonmonotonic knowledge bases, like logic programs [1, 6, 18, 10] or default theories [17], are more recent.

The problem of updating logic programs, on which we focus here, deals with the incorporation of an update P, given by a rule or a set of rules, into the current knowledge base KB. Accordingly, sequences $P_1, \ldots, P_n$ of updates lead to sequences $(KB, P_1, \ldots, P_n)$ of logic programs which are given a declarative semantics. To broaden this approach, Alferes *et al.* [2] have proposed the LUPS update language, in which updates consist of sets of *update commands*. Such commands permit to specify changes to KB in terms of adding or removing rules from it. For instance, a typical command is `assert` $a \leftarrow b$ `when` c, which says that rule $a \leftarrow b$ should be added to KB if c is currently true in it. Similarly, `retract` b states that b must be eliminated from KB, without any further condition.

However, a certain limitation of LUPS and the above formalisms is that while they handle *ad hoc* changes of KB, they are not conceived for handling a *yet unknown* update, which will arrive as the environment evolves. In fact, these approaches lack the possibility to specify how an agent should react upon the arrival of such an update. For example, we would like to express that, on arrival of the fact $best_buy(shop_1)$, this should be added to KB, while best-buy information about other shops is removed from KB.

In this paper, we address this issue and present a declarative framework for specifying the update behavior of an agent. The agent receives new information in terms of a set of rules, viewed as *event*, and adjusts its KB according to an *update policy*, which consists of statements in a declarative language.

Our main contributions are briefly summarized as follows:

(1) We present a *generic* framework for specifying update behavior, which can be instantiated with different update approaches to logic programs. This is facilitated by a *layered approach*: At the top level, the update policy is evaluated, given an event and the agent's current belief set, to single out the update commands U which need to be performed on KB. At the next layer, U is compiled to a set P of rules to be incorporated to KB; at the bottom level, the updated knowledge base is represented as a sequence of logic programs, serving as input for the underlying update semantics for logic programs, which determines the new current belief set.

(2) We define a declarative language for update policies, which generalizes LUPS by various features. Most importantly, access to incoming events is facilitated. 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 like this may involve further conditions on the current belief set, and other commands to be executed (which is not possible in LUPS). The language thus enables the flexible handling of events, such as simply recording changes in the environment, skipping uninteresting updates, or applying default actions.

(3) We analyze some properties of the framework, using the update answer set semantics of [6] as a representative of similar approaches. In particular, useful properties concerning KB maintenance are explored, and the complexity of the framework is determined. Moreover, we describe a possible realization of the framework in the agent system IMPACT [16], providing evidence that our approach can serve as a viable tool supporting the development of adaptive reasoning agents.

2 Preliminaries

An *extended logic program* (ELP, or simply *program*) is a finite set P of rules r of the form

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

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 called *head* of r and is denoted by $H(r)$. We allow the case where L_0 may be absent, in which case r is said to be 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 $B(r) = \emptyset$. 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, $\neg S = \{\neg L \mid L \in S\}$. In particular, $Lit_{\mathcal{A}} = \mathcal{A} \cup \neg\mathcal{A}$ is the set of all literals over $\mathcal{A}$. $\mathcal{L}_{\mathcal{A}}$ denotes 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 $n \geq 1$.

We adopt an abstract view of the semantics of ELPs and update programs, given as a mapping $Bel(\cdot)$, 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 $Bel(\cdot)$ are possible, according to various proposals for update semantics in the literature (e.g., [1, 18, 10, 6, 13]). We only assume that $Bel(\cdot)$ has some elementary properties which these or any other "reasonable" semantics satisfies. In particular, $P_n \subseteq Bel(\boldsymbol{P})$ must hold, and the following property must be satisfied: 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 here for illustration the answer set semantics for update programs from [6], which coincides with the semantics of inheritance programs in [4]. 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 [9] iff it is a minimal model of the *reduct* P^S, defined by

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

$\mathcal{AS}(P)$ denotes the collection of all answer sets of P.

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

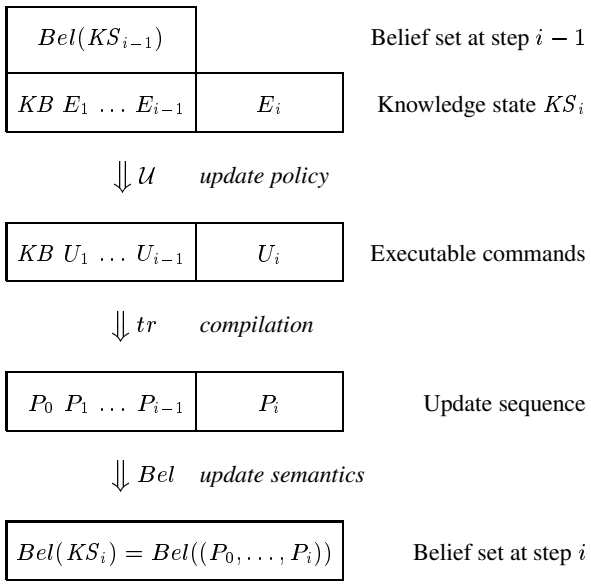

Fig. 1. From knowledge state to belief set at step i.

$j > i$. 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 answer set semantics. A syntactic transformation of $\boldsymbol{P}$ into an ELP realizing this semantics is given in [6].

Example 2.1. 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 $Rej(S_0, P_0) = \emptyset$; (P_0, P_1) has answer set $S_1 = \{\neg a, c, b\}$ 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 $L \leftarrow \in Bel_{\mathcal{A}}(\boldsymbol{P})$.

3 Update Policies

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

3.1 Basic Framework

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

Definition 3.1. *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. In what follows, we assume that an event class $\mathcal{EC}$ has been fixed.

$\langle stat \rangle$	::=	$\langle comm \rangle$ [**if** $\langle cond1 \rangle$] [$[\![$ $\langle cond2 \rangle$ $]\!]$];
$\langle c_name \rangle$	::=	**assert**[_**event**] | **retract**[_**event**] | **always**[_**event**] | **cancel** | **ignore** ;
$\langle r_id \rangle$	::=	$\langle rule \rangle$ | $\langle r_var \rangle$;
$\langle lit_id \rangle$	::=	$\langle literal \rangle$ | $\langle lit_var \rangle$;
$\langle comm \rangle$	::=	$\langle c_name \rangle(\langle r_id \rangle)$;
$\langle cond1 \rangle$	::=	[**not**] $\langle comm \rangle$ | [**not**] $\langle comm \rangle$, $\langle cond1 \rangle$;
$\langle cond2 \rangle$	::=	$\langle kb_conds \rangle$ | **E** : $\langle ev_conds \rangle$ | $\langle kb_conds \rangle$, **E** : $\langle ev_conds \rangle$;
$\langle kb_conds \rangle$	::=	$\langle kb_cond \rangle$ | $\langle kb_cond \rangle$, $\langle kb_conds \rangle$;
$\langle kb_cond \rangle$	::=	$\langle r_id \rangle$ | $\langle lit_id \rangle$;
$\langle ev_conds \rangle$	::=	$\langle ev_cond \rangle$ | $\langle ev_cond \rangle$, $\langle ev_conds \rangle$;
$\langle ev_cond \rangle$	::=	$\langle lit_id \rangle$ | $\langle r_id \rangle$;

Table 1. Syntax of an update statement in EPI.

Definition 3.2. *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 procedure for adapting the belief set $Bel(KS_{i-1})$ on arrival of E_i is illustrated in Figure 1. Informally, at step i of the knowledge evolution, given the 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 in terms of the update policy $\mathcal{U}$. First, a set U_i of *executable commands* is determined from $\mathcal{U}$. 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 $\boldsymbol{P} = (P_0, P_1, \ldots, P_i)$. Then, $Bel(KS_i)$ is given by $Bel(\boldsymbol{P})$.

3.2 Language EPI: Syntax

The language EPI generalizes the update specification language LUPS [2], by allowing update statements to depend on other update 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 external event).

The syntax of EPI is given in Table 1. In what follows, we use cmd to denote update commands and ρ may stand for rules or rule variables. In general, an EPI statement may have the 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, expressed by $cmd_1(\rho_1)$, depending on other commands $[\textbf{not}]cmd_2(\rho_2)$, ...,$[\textbf{not}]cmd_m(\rho_m)$, and conditioned with the proviso whether c_1 belongs to the current belief set and whether c_2 is in the actual event. The basic EPI commands are the same as in LUPS (for their meaning, see also [2]), plus the additional command **ignore**, which allows to skip unintended updates from the environment, which otherwise would be incorporated into the knowledge base. Each condition in $[\![\cdot]\!]$, both of the form c_1 and $\mathbf{E} : c_2$, can be substituted by a list of such conditions. 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.

Definition 3.3. *An* update policy $\mathcal{U}$ *is a finite set of* EPI *statements.*

For instance, the EPI statement

$$\mathbf{assert}(R)\ \mathbf{if\ not\ ignore}(R)[\![E : R]\!] \tag{2}$$

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. Similarly, the command **retract** forces a rule to be deactivated. The option **event** states that an assertion or retraction has only temporary value and is not supposed to persist by inertia in subsequent steps. The exact meaning of the different update commands will be made clear in the next section.

Example 3.1. Consider a simple agent selecting Web shops in search for some specific merchandise. Suppose its knowledge base, KB, contains the rules

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

and a fact $r_0 : date(0)$ is 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, and r_2, 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 sets of facts or ground rules of the 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 (2), as well as:

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

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.

3.3 Language EPI: Semantics

According to the overall structure of the semantics of EPI, as depicted in Figure 1, at step i, we first determine the executable command 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

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

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

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

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:

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

Definition 3.4. *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 $\Pi_i^{\mathcal{U}}$.*

Since update statements do not contain strong negation, executable update commands are in fact *stable models* of $\mathcal{G}(\Pi_i^{\mathcal{U}})$ [8]. 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 $Bel(\cdot)$, if needed, restricting the outcomes of the update to the original language.

Next we compile the executable commands $U_1, \ldots, U_i$ into an update sequence $(P_0, \ldots, P_i)$, serving as input for the belief function $Bel(\cdot)$. This is realized by means of a transformation $tr(\cdot)$, which is a generic and adapted version of a similar mapping introduced by Alferes *et al.* [2]. 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.

Definition 3.5. *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 3.4:*

$i = 0$: *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 $\emptyset$.*

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

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

$$\begin{aligned} PC_i = {} & PC_{i-1} \cup \{\mathbf{always}(r) \mid \mathbf{always}(r) \in EC_i\} \\ & \cup \{\mathbf{always_event}(r) \mid \mathbf{always_event}(r) \in EC_i \wedge \mathbf{always}(r) \notin EC_i \cup PC_{i-1}\} \\ & \setminus (\{\mathbf{always_event}(r) \mid \mathbf{always}(r) \in EC_i\} \cup \{\mathbf{always}[_\mathbf{event}](r) \mid \mathbf{cancel}(r) \in EC_i\}); \end{aligned}$$

$$\begin{aligned} P_i = {} & \{on(n_r) \leftarrow\ , H(r) \leftarrow B(r), on(n_r) \mid \\ & \quad \mathbf{assert}[_\mathbf{event}](r) \in EC_i \vee \mathbf{always}[_\mathbf{event}](r) \in PC_i\} \\ & \cup \{on(n_r) \leftarrow \mid \mathbf{retract_event}(r) \in EC_{i-1} \wedge \mathbf{retract}[_\mathbf{event}](r) \notin EC_i\} \\ & \cup \{\neg on(n_r) \leftarrow \mid (\mathbf{retract}[_\mathbf{event}](r) \in EC_i \wedge \mathbf{always}[_\mathbf{event}](r) \notin PC_i) \\ & \quad \vee (\mathbf{cancel}(r) \in EC_i \wedge \mathbf{always_event}(r) \in PC_{i-1} \wedge \mathbf{assert}[_\mathbf{event}](r) \notin EC_i) \\ & \quad \vee (\mathbf{assert_event}(r) \in EC_{i-1} \wedge \mathbf{always}[_\mathbf{event}](r) \notin PC_i \\ & \quad\quad \wedge \mathbf{assert}[_\mathbf{event}](r) \notin EC_i)\}. \end{aligned}$$

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

Definition 3.6. *Let KS and $\mathcal{U}$ be as in Definition 3.5, and let $U_1, \ldots, U_n$ be the corresponding executable commands obtained from Definition 3.4. Then, the* belief set *of KS is given by $Bel(KS) = Bel(tr(KB; U_1, \ldots, U_n))$.*

Example 3.2. Reconsider Example 3.1 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

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

which is compiled, via $tr(\cdot)$, to $PC_1 = PC_0 \setminus \{\mathbf{assert_[event]}(date(0))\} = \emptyset$ and $P_1 = \{sale(s_0) \leftarrow on(r_1');\ on(r_1') \leftarrow ;\ date(1) \leftarrow on(r_2');\ on(r_2') \leftarrow ;\ \neg on(r_0) \leftarrow \}$. As easily seen, the belief set $Bel(\langle KB, E_1\rangle)$ = $Bel((P_0, P_1))$ contains $sale(s_0)$ and $query(s_0)$.

4 Properties

In this section, we discuss some properties of $Bel(KS)$ for particular update policies, using the definition of $Bel(\cdot)$ based on the update answer sets approach of [6], as explained in Section 2. We stress that the properties given below are also satisfied by similar instantiations of $Bel(\cdot)$, like e.g., dynamic logic programming [1].

First, we note some basic properties:

- If $\mathcal{U} = \emptyset$ (*empty policy*), then KB will never be updated; the belief set is independent of $E_1, \ldots, E_n$, and thus *static*. Hence, $Bel(KS_i) = Bel(KB)$, for each $i = 1, \ldots, n$.
- If $\mathcal{U} = \{\mathbf{assert}(R)[\![\mathbf{E} : R]\!]\}$ (*unconditional assert policy*), then all rules contained in the received events are directly incorporated into the update sequence. Thus, $Bel(KS_i) = Bel((KB, E_1, \ldots, E_i))$, for each $i = 1, \ldots, n$.
- If U_i is empty, then the knowledge is not updated, i.e., $P_i = \emptyset$. Thus, $Bel(KS_i) = Bel(KS_{i-1})$.
- Similarly, if $U_i = \{\mathbf{assert}(\bot_i) \leftarrow \}$, then $Bel(KS_i) = Bel(KS_{i-1})$.

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 transformation $tr(\cdot)$ 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 4.1. *Let $\boldsymbol{P} = (P_0, \ldots, P_n)$ be an update sequence. For every $r \in P_i, r' \in P_j$ with $i < j$, the following holds: if (i) $r = r'$, or (ii) $r = L \leftarrow$ and $r' = \neg L \leftarrow$, or (iii) $r' = L \leftarrow$ such that no rule $r'' \in P_k$ with $H(r'') = \neg L$ exists, where $k \in \{j+1, \ldots, n\}$, and $\neg L \in B(r)$, then $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, \ldots, P_{i-1}, P_i \setminus \{r\}, P_{i+1}, \ldots, P_n)$.*

The following property holds:

Theorem 4.1. *Let KS be a knowledge state and $Bel(KS) = Bel(\boldsymbol{P})$, where $\boldsymbol{P} = (P_0, \ldots, P_n)$. Furthermore, let $\boldsymbol{P}^*$ result from $\boldsymbol{P}$ after repeatedly removing rules as in Lemma 4.1, and let $\boldsymbol{P}^- = (P_0^-, \ldots, P_n^-)$, where*

$$P_i^- = \{H(r) \leftarrow B(r) \setminus \{on(n_r)\} \mid r \in \boldsymbol{P}_i^*,\ on(n_r) \leftarrow \in \boldsymbol{P}^*\} \setminus \{on(n_s) \leftarrow \mid on(n_s) \leftarrow \in \boldsymbol{P}\}.$$

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

Thus, we can purge the knowledge base and remove duplicates of rules, as well as all deactivated (retracted) rules.

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 $tr(\cdot)$ 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 4.2. *Let $\boldsymbol{P} = (P_0, \ldots, P_n)$ be an update sequence such that P_i contains only 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 $P_{u_{i+1}} = P_{i+1} \cup (P_{u_i} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P_{i+1}\})$.*

We can thus assert the following proposition for history contraction:

Theorem 4.2. *Let $\mathcal{U}$ be a factual assert policy and $\boldsymbol{P} = (P_0, \ldots, P_n)$ 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 4.2.*

Computational Complexity Finally, we briefly address the complexity of reasoning about a knowledge state KS. An update policy $\mathcal{U}$ is called *stratified*, if for all EPI statements $cmd(\rho)$ **if** $C_1[\![C_2]\!] \in \mathcal{U}$, the associated rules $cmd_1(\rho) \leftarrow C_1'$ form a stratified logic program, where C_1' results from C_1 by replacing the EPI declaration **not** by 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. If $Sel(\cdot)$ is unknown, we consider all possible return values (i.e., all answer sets of $\Pi_i^{\mathcal{U}}$) and thus, in a cautious reasoning mode, all possible $Bel(KS) = Bel((P_0, \ldots, P_n))$ from Figure 1. Clearly, deciding $r' \in Bel((Q_0, \ldots, Q_m))$ is for update answer sets in coNP; it is polynomial, if Q_0 is stratified and each Q_i, $1 \leq i \leq m$, contains only facts.

Theorem 4.3. *Let $Bel(\cdot)$ be the update answer set semantics, and $Sel(\cdot)$ polynomial-time computable with an* NP *oracle. Then, given a ground rule r and a ground knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$, the problem of deciding whether $r \in Bel(KS)$ is Π_2^{P}-hard. If the update policy $\mathcal{U}$ is stratified, then the problem is P^{NP}-hard. Moreover, if the update policy is factual and stratified, and the initial knowledge base KB is stratified as well, then the problem is polynomial.*

Similar results hold, e.g., for dynamic logic programming.

The results can be intuitively explained as follows. 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 previous results, P_i can be computed in polynomial time. Since P_i contains at most $|\mathcal{U}|$ rules, step i is feasible in polynomial time with an NP oracle. Thus, $\boldsymbol{P} = (P_0, \ldots, P_n)$ is polynomially computable with an NP oracle, and $r \in Bel(\boldsymbol{P})$ is decided with another oracle call. Updating a stratified P_0 such that only sets of facts $P_1, P_2, \ldots$ may be added preserves polynomial decidability of $r' \in Bel((P_0, \ldots, P_{i-1}))$; this explains the polynomial result. Otherwise, P^{NP}-hard problems such as computing the lexicographically maximal model of a CNF formula [15] are easily reduced to the problem.

For general update policies, multiple answer sets are possible, and each possible result of $Sel(\Pi_i^{\mathcal{U}})$ can be nondeterministically guessed and verified in polynomial time. This leads to $\mathrm{coNP}^{\mathrm{NP}} = \Pi_2^{\mathrm{P}}$ complexity.

5 Implementational Issues

An elegant and straightforward realization of update policies is possible through IMPACT agent programs. IMPACT [16] is a platform for developing software agents, which allows to build agents on top of legacy code, i.e., existing software packages, that operates on arbitrary data structures. Thus, in accordance with our approach, we can design a *generic implementation* of our framework, without committing to a particular update semantics $Bel(\cdot)$.

In fact, since every update policy $\mathcal{U}$ is semantically reduced to a logic program, the corresponding executable commands can be computed using well-known logic programming engines like `smodels` [14], `DLV` [7] or `DeRes` [5]. Hence, we may assume that 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 $\mathcal{SP}$ (or another package) has a function `event()`, which lists all rules of a current event. Then, an update policy $\mathcal{U}$ can be represented in IMPACT as follows.

(1) Conditions on the belief set and the event can be modeled through IMPACT *code call atoms*, i.e., atoms $\mathtt{in}(\mathtt{t}, \mathtt{bel}())$, $\mathtt{not_in}(\mathtt{t}, \mathtt{bel}())$, and $\mathtt{in}(\mathtt{t}, \mathtt{event}())$, where `t` is a constant r or a variable R. In IMPACT, $\mathtt{in}(r, \mathtt{f}())$ is true if constant r is in the result returned by $\mathtt{f}()$; a variable R is bound to all r such that $\mathtt{in}(r, \mathtt{f}())$ is true; '`not_in`' is negation.

(2) Update commands can be easily represented as IMPACT *actions*. An action is implemented by a body of code in any programming language (e.g., C); its effects are specified in terms of add and delete lists (sets of code call atoms). Thus, actions like $\mathbf{assert}(R)$, $\mathbf{retract}(R)$, etc., where R is a parameter, are introduced.

(3) EPI statements are represented as IMPACT action rules

$$\mathtt{Do}(cmd_1(\rho_1)) \leftarrow [\neg]\mathtt{Do}(cmd_2(\rho')), \ldots, [\neg]\mathtt{Do}(cmd_m(\rho')), code_call_atoms(\mathtt{cond}),$$

where $code_call_atoms(\mathtt{cond})$ is the list of the code call atoms for the conditions on the belief set and the event in `cond` as described above.

The semantics of IMPACT agent programs is defined in terms of *status sets*. The notion of a *reasonable status set* S is equivalent to a stable model of a logic program, and prescribes the agent to perform all actions α where $\mathtt{Do}(\alpha)$ is in S. Thus, S represents the executable commands U_i in Figure 1 according to $\mathcal{U}$, and the respective action execution affects the computation of P_i via $tr(\cdot)$.

6 Related Work and Conclusion

Our approach is similar in spirit to the work in active databases (ADBs), where the dynamics of a database is specified through *event-condition-action rules* triggered by events. In contrast to our context, ADBs have in general no declarative semantics, and only one rule at a time fires, maybe causing successive events. In [3], a declarative characterization of ADBs is given, in terms of a reduction to logic programs, by using situation calculus notation.

Our language for update policies is also related to *action languages*, which can be compiled to logic programs as well (cf., e.g., [11]). A change to the knowledge base may be considered as an action, where the execution of actions may depend on other actions and conditions. However, action languages are tailored for planning and reasoning about actions, rather than reactive behavior specification. Furthermore, a state is, essentially, a set of literals rather than a belief set as we define it. Investigating the relationship of our framework to these languages in detail, and in particular concerning embeddings, is an interesting issue for further research.

A development in the area of action languages, with purposes similar to those of EPI, is the policy description language $\mathcal{PDL}$, introduced by Lobo *et al.* [12]. $\mathcal{PDL}$ is a declarative language which extends traditional action languages with the notion of sequences of events, and offers the possibility of specifying actions as reactive behavior in response to such external events. A $\mathcal{PDL}$ policy is a collection of *event-condition-action rules*, interpreted as a function associating a sequence of events with a set of actions. The relation between the semantics of EPI and that of $\mathcal{PDL}$ is worth being further investigated.

The EPI language could be extended with several features:

(1) Special atoms $\mathbf{in}(r)$ telling whether r is actually part of KB (i.e., activated by $on(n_r)$), which allow to access the "extensional" part of KB.

(2) Rule terms involving literal constants and variables, e.g., "$H \leftarrow up(s_1), B$", where H, B are variables and $up(s_1)$ is a fixed atom, providing access to the structure of rules. Combined with (1), commands such as "remove all rules involving $up(s_1)$" can then be conveniently expressed.

(3) More expressive conditions on the knowledge base are conceivable, requesting for more complex reasoning tasks, and possibly taking temporal evolution into account. E.g., "$\mathbf{prev}(a)$" expressing that a was true at the previous stage.

Our generic framework, which extends approaches to logic program updates, represents a convenient platform for declarative update specifications and could also be fruitfully used in several applications. Exploring these issues is part of our ongoing research.

Acknowledgments. This work was partially supported by the Austrian Science Fund (FWF) under grants P13871-INF and N Z29-INF.

References

1. J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic Updates of Non-Monotonic Knowledge Bases. *J. of Logic Programming*, 45(1–3):43–70, 2000.
2. J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS - A Language for Updating Logic Programs. In *Proc. LPNMR'99*, volume 1730 of *LNAI*, pages 162–176. Springer, 1999.
3. C. Baral and J. Lobo. Formal Characterization of Active Databases. In D. Pedreschi and C. Zaniolo, editors, *Workshop of Logic on Databases (LID '96)*, volume 1154 of *Lecture Notes in Computer Science*, pages 175–195, San Miniato, Italy, 1996.
4. F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. In *Proc. ICLP'99*, pages 79–93. MIT Press, 1999.
5. P. Cholewinski, W. Marek, and M. Truszczyński. Default Reasoning System DeReS. In *Proc. KR-96*, pages 518–528, 1996.
6. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on Updates of Logic Programs. In *Proc. JELIA 2000*, volume 1919 of *LNAI*, pages 2–20. Springer, 2000.
7. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A Deductive System for Non-monotonic Reasoning. In *Proc. LPNMR-97*, pages 363–374, 1997.
8. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. ICSLP'88*, pages 1070–1080. MIT Press, 1988.
9. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3–4):365–386, 1991.
10. K. Inoue and C. Sakama. Updating Extended Logic Programs through Abduction. In *Proc. LPNMR'99*, volume 1730 of *LNAI*, pages 147–161. Springer, 1999.
11. V. Lifschitz and H. Turner. Representing Transition Systems by Logic Programs. In *Proc. LPNMR'99*, volume 1730 of *LNAI*, pages 92–106. Springer, 1999.
12. J. Lobo, R. Bhatia, and S. Naqvi. A Policy Description Language. In *Proc. AAAI/IAAI'99*, pages 291–298. AAAI Press / MIT Press, 1999.
13. W. Marek and M. Truszczyński. Revision Programming. *Theoretical Computer Science*, 190:241–277, 1998.
14. I. Niemelä and P. Simons. Smodels: An Implementation of the Stable Model and Well-Founded Semantics for Normal Logic Programs. In *Proc. LPNMR-97*, pages 420–429, 1997.
15. C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
16. V. Subrahmanian, J. Dix, T. Eiter, P. Bonatti, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, 2000.
17. M.-A. Williams and G. Antoniou. A Strategy for Revising Default Theory Extensions. In *Proc. KR'98*, pages 24–35. Morgan Kaufmann, 1998.
18. Y. Zhang and N. Foo. Updating Logic Programs. In *Proc. ECAI'98*, pages 403–407. Wiley, 1998.