# A Framework for Declarative Update Specifications in Logic Programs

**Thomas Eiter** and **Michael Fink** and **Giuliana Sabbatini** and **Hans Tompits**
Institut für Informationssysteme, Abteilung Wissensbasierte Systeme 184/3,
Technische Universität Wien, Favoritenstrasse 9-11, A-1040 Vienna, Austria
E-mail:{eiter,michael,giuliana,tompits}@kr.tuwien.ac.at

## Abstract

Recently, several approaches for updating knowledge bases represented as logic programs have been proposed. In this paper, 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 must 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, e.g., updates of logic programs [Alferes *et al.*, 2000; Eiter *et al.*, 2000; Zhang and Foo, 1998; Inoue and Sakama, 1999] or of default theories [Williams and Antoniou, 1998], 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.* [1999a] 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` $c$, stating that rule $a \leftarrow b$ should be added to $KB$ if $c$ is currently true in it. Similarly, `retract` $b$ expresses that $b$ must be eliminated from $KB$, without any further condition.

However, a certain limitation of LUPS and the above mentioned 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 update behavior of an agent. The agent receives new information in terms of a set of rules (which is called an *event*), and adjusts its $KB$ in accord to a given *update policy*, consisting of statements in a declarative language. Our main contributions are 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, generalizing LUPS by various features. Most importantly, access to incoming events is facilitated. For example, **retract**$(best\_buy(shop_1))$ $[\![\mathbf{E} : best\_buy(shop_2)]\!]$ states 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 Eiter *et al.* [2000] 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 [Subrahmanian *et al.*, 2000], providing evidence that our approach is a viable tool for developing adaptive reasoning agents.

## 2 Preliminaries

We assume the reader familiar with *extended logic programs* (ELPs) [Gelfond and Lifschitz, 1991]. For a rule $r$, we write $H(r)$ and $B(r)$ to denote the head and body of $r$, respectively. Furthermore, $not$ stands for default negation and $\neg$ for strong negation. $Lit_{\mathcal{A}}$ is the set of all literals over a set of atoms $\mathcal{A}$, and $\mathcal{L}_{\mathcal{A}}$ is the set of all rules constructible from $Lit_{\mathcal{A}}$.

An *update program*, $\boldsymbol{P}$, is a sequence $(P_1, \ldots, P_n)$ of ELPs, 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. We only assume that $Bel(\cdot)$ satisfies some elementary properties which any "reasonable" semantics satisfies. In particular, we assume that $P_n \subseteq Bel(\boldsymbol{P})$ holds, and that the following property is satisfied: given $A \leftarrow\, \in Bel(\boldsymbol{P})$ and $A \in B(r)$, then $r \in Bel(\boldsymbol{P})$ iff $H(r) \leftarrow B(r) \setminus \{A\} \in Bel(\boldsymbol{P})$.

We use here the semantics of Eiter *et al.* [2000], which coincides with the semantics of inheritance programs due to Buccafurri *et al.* [1999]. The semantics of ELPs $P$ and update sequences $\boldsymbol{P}$ with variables is defined as usual through their ground versions $\mathcal{G}(P)$ and $\mathcal{G}(\boldsymbol{P})$ over the Herbrand universe, respectively. In what follows, let $\mathcal{A}$, $P$, $\boldsymbol{P}$, etc. be ground.

An *interpretation* is a set $S \subseteq Lit_{\mathcal{A}}$ which contains no complementary pair of literals. $S$ is a (consistent) *answer set* of an ELP $P$ iff it is a minimal model of the *reduct* $P^S$, which results from $P$ by deleting all rules whose body contains some default literal $not\, L$ with $L \in S$, and by removing all default literals in the bodies of the remaining rules [Gelfond and Lifschitz, 1991]. By $\mathcal{AS}(P)$ we denote the collection of all answer sets of $P$. 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 $B(r) \cup B(r') \subseteq S$, for some $r' \in P_j \setminus Rej_j(S, \boldsymbol{P})$ with $j > i$. Then, $S$ is an *answer set* of $\boldsymbol{P} = (P_1, \ldots, P_n)$ iff $S$ is an answer set of $\bigcup_i P_i \setminus Rej(S, \boldsymbol{P})$. We denote the collection of all answer sets of $\boldsymbol{P}$ by $\mathcal{AS}(\boldsymbol{P})$. Since $n = 1$ implies $Rej(S, \boldsymbol{P}) = \emptyset$, the semantics extends the answer set semantics. [Eiter *et al.*, 2000] describes a characterization of the update semantics in terms of single ELPs.

**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* $Rej(S_0, P_0) = \emptyset$*;* $(P_0, P_1)$ *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 $r \in \mathcal{L}_{\mathcal{A}}$ such that $r$ is true in each $S \in \mathcal{AS}(\boldsymbol{P})$. 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*.

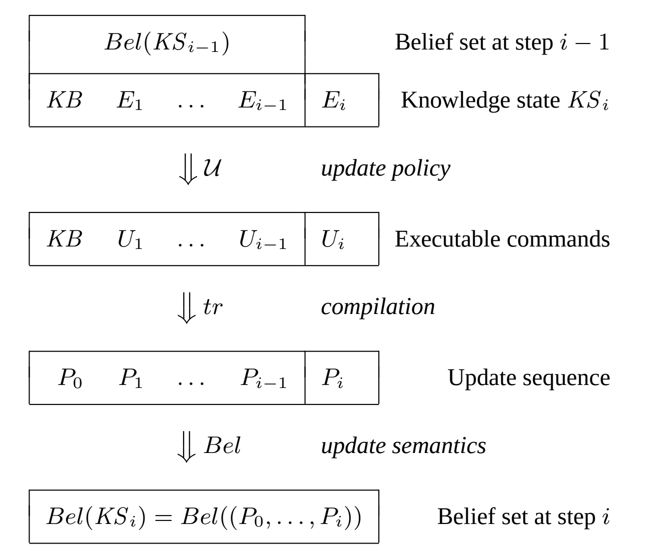


Figure 1: From knowledge state to belief set at step $i$.

### 3.1 Basic Framework

We start with the formal notions of an *event* and of the *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 (i.e., sets of communicated rules) 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.

**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 the evolution of the agent's knowledge, starting from its initial knowledge base. When a new event $E_i$ occurs, the current knowledge state $KS_{i-1}$ changes to $KS_i = \langle KB; E_1, \ldots, E_{i-1}, E_i \rangle$, which requests the agent to incorporate the event $E_i$ 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, we are 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 $E_i$, and we want to compute $Bel(KS_i)$ 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 $Bel(\boldsymbol{P})$.

| | | |
|---|---|---|
| $\langle stat \rangle$ | ::= | $\langle comm \rangle$ [**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**] $\langle comm \rangle, \langle cond1 \rangle$ ; |
| $\langle cond2 \rangle$ | ::= | $\langle kb\_conds \rangle$ \| $\mathbf{E} : \langle ev\_conds \rangle$ \| $\langle kb\_conds \rangle$ , $\mathbf{E} : \langle ev\_conds \rangle$; |
| $\langle kb\_conds \rangle$ | ::= | $\langle kb\_cond \rangle$ \| $\langle kb\_cond \rangle$ , $\langle kb\_conds \rangle$; |
| $\langle kb\_cond \rangle$ | ::= | $\langle r\_id \rangle$ \| $\langle lit\_id \rangle$ ; |
| $\langle ev\_conds \rangle$ | ::= | $\langle ev\_cond \rangle$ \| $\langle ev\_cond \rangle$ , $\langle ev\_conds \rangle$; |
| $\langle ev\_cond \rangle$ | ::= | $\langle lit\_id \rangle$ \| $\langle r\_id \rangle$ ; |

Table 1: Syntax of an update statement in EPI.

### 3.2 Language EPI: Syntax

The language EPI generalizes the update specification language LUPS [Alferes *et al.*, 1999a], 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 provision to support external events). These features make it suitable for implementing rational reactive agents, capable, e.g., of filtering incoming information.

The syntax of EPI is given in Table 1. In what follows, we use $cmd$ to denote update commands and $\rho$ to refer to rules or rule variables. In general, an EPI statement may have the form

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

which states conditional assertion or retraction of a rule $\rho_1$, expressed by $cmd_1(\rho_1)$, depending on other commands $[\mathbf{not}]cmd_2(\rho_2)$, ..., $[\mathbf{not}]cmd_m(\rho_m)$, 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 those in LUPS (for their meaning, cf. [Alferes *et al.*, 1999a]), 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** *An* update policy $\mathcal{U}$ *is a finite set of* EPI *statements.*

For instance, the EPI statement

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

means that all rules in the event have to be incorporated into the new knowledge base, except if it is explicitly 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 precise meaning of the different update commands will be made clear in the next section.

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

$$r_1: \quad query(S) \leftarrow sale(S), up(S), not\ \neg query(S);$$
$$r_2: \quad try\_query \leftarrow query(S);$$
$$r_3: \quad notify \leftarrow not\ try\_query;$$

*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, 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 set 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* $\mathcal{U}$ *may be defined as follows. Assume it contains the incorporate-by-default statement* (1)*, as well as:*

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

*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:*

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

*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

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

The update statements in $\mathcal{G}(\mathcal{U})^i$ are thus of the form $cmd_1(\rho_1)$ **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:

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

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

Since update statements do not contain strong negation, executable update commands are actually *stable models* of $\mathcal{G}(\Pi_i^{\mathcal{U}})$ [Gelfond and Lifschitz, 1988]. Furthermore, since programs may in general have more than one answer set, or no answer set at all, and the agent must commit itself to a single set of update commands, 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 reserved atom. 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.* [1999a]. 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 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 $\emptyset$.*

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

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

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

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

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

**Definition 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 Definition 4. Then, the* belief set *of $KS$ is given by $Bel(KS) = Bel(tr(KB; U_1, \ldots, U_n))$.*

**Example 3** *Reconsider Example 2 and suppose the event $E_1 = \{sale(s_0),\, date(1)\}$ occurs at $KS = \langle KB\rangle$. Then,*

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

*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 function $tr(\cdot)$, 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 Eiter *et al.* [2000], 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 [Alferes *et al.*, 2000].

First, we note some basic properties:

- If $\mathcal{U} = \emptyset$ (called *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 $\mathcal{U} = \{\mathbf{assert}(R)[\![\mathbf{E} : R]\!]\}$ (called *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., $P_i = \emptyset$. We thus have $Bel(KS_i) = Bel(KS_{i-1})$.
- 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 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 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 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 thus: (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 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 2** *Let $\mathcal{U}$ be a factual assert policy and $\boldsymbol{P} = (P_1, \ldots, P_n)$ 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.*

Simple examples show that Theorem 2 does not hold in general. The investigation of classes of policies for which similar results hold are a subject for further research.

**Computational Complexity**

Finally, we briefly address the complexity of reasoning about a knowledge state $KS$. An update policy $\mathcal{U}$ is called *stratified* iff, 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 $\mathbf{not}$ 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, for update answer sets, deciding $r' \in Bel((Q_0, \ldots, Q_m))$ is in coNP; it is polynomial, if $Q_0$ is stratified and each $Q_i$, $1 \leq i \leq m$, contains only facts.

**Theorem 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 ground $KS = \langle KB; E_1, \ldots, E_n \rangle$, the complexity of deciding whether $r \in Bel(KS)$ is as follows (entries denote completeness results; the case of unknown $Sel(\cdot)$ is given at the right of "/"):*

| $KB \setminus \mathcal{U}$ | *fact. assert & strat.* | *stratified* | *general* |
|---|---|---|---|
| *stratified* | P | $\mathrm{P}^{\mathrm{NP}}$ | $\mathrm{P}^{\mathrm{NP}}/\Pi_2^P$ |
| *general* | $\mathrm{P}^{\mathrm{NP}}$ | $\mathrm{P}^{\mathrm{NP}}$ | $\mathrm{P}^{\mathrm{NP}}/\Pi_2^P$ |

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 ($1 \leq i \leq n$), 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, $P_i$ is easily computed in polynomial time. Since $P_i$ contains less than $|\mathcal{U}|$ 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 decidability result. In all other cases, $\mathrm{P}^{\mathrm{NP}}$-hard problems such as computing the lexicographically maximal model of a CNF formula are easily reduced to the problem.

If $Sel(\cdot)$ is unknown, each possible result of $Sel(\Pi_i^{\mathcal{U}})$ can be nondeterministically guessed and verified in polynomial time. This leads to $\mathrm{coNP}^{\mathrm{NP}} = \Pi_2^P$ complexity.

## 5 Implementational Issues

An elegant and straightforward realization of update policies is possible through IMPACT agent programs. IMPACT [Subrahmanian *et al.*, 2000] 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 ourselves to a particular update semantics $Bel(\cdot)$.

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`, `DLV`, or `DeRes`. 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()` returning the current belief set $Bel(KS)$. Moreover, we assume that $\mathcal{SP}$ 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 by 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 `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 through *status sets*. A *reasonable status set* $S$ is equivalent to a stable model of a logic program, and prescribes the agent to perform all actions $\alpha$ where $\mathtt{Do}(\alpha)$ is in $S$. Thus, $S$ represents the executable commands $U_i$ of Figure 1 in accord with $\mathcal{U}$, and the respective action execution affects the computation of $P_i$ via $tr(\cdot)$. For more details, cf. [Eiter *et al.*, 2001].

## 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* (*ECA*) *rules* triggered by events. However, ADBs have in general no declarative semantics, and only one rule at a time fires, possibly causing successive events. In [Baral and Lobo, 1996], 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., [Lifschitz and Turner, 1999]). 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; events would have to be emulated. Furthermore, a state is, essentially, a set of literals rather than a belief set as we define it. Investigating the relationships of our framework to these languages in detail—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 $\mathcal{PDL}$ [Lobo *et al.*, 1999]. It extends traditional action languages with the notion of *event sequences*, and serves for specifying actions as reactive behavior in response to events. A $\mathcal{PDL}$ policy is a collection of ECA rules, interpreted as a function associating with an event sequence a set of actions. $\mathcal{PDL}$ seems thus to be more expressive than EPI; possible embeddings of EPI into $\mathcal{PDL}$ remain to be explored.

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 $on(n_r)$), allowing 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 thus be conveniently expressed.

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

In concluding, our generic framework, which extends other 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.

### Acknowledgements

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

## References

[Alferes *et al.*, 1999a] 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.

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

[Baral and Lobo, 1996] C. Baral and J. Lobo. Formal characterization of active databases. In *Proc. LID'96*, LNCS 1154, pp. 175-195. Springer, 1996.

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

[Eiter *et al.*, 2000] Th. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on updates of logic programs. In *Proc. JELIA'00*, LNAI 1919, pp. 2-20. Springer, 2000.

[Eiter *et al.*, 2001] Th. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Declarative knowledge updates through agents. In *Proc. AISB'01 Symp. on Adaptive Agents and Multi-Agent Systems, York, UK*, pp. 79-84. AISB, 2001.

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

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

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

[Lifschitz and Turner, 1999] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In *Proc. LPNMR'99*, LNAI 1730, pp. 92-106. Springer, 1999.

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

[Marek and Truszczyński, 1998] W. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science*, 190:241-277, 1998.

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

[Williams and Antoniou, 1998] M.-A. Williams and G. Antoniou. A strategy for revising default theory extensions. In *Proc. KR'98*, pp. 24-35. Morgan Kaufmann, 1998.

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