Contents

Declarative Update Policies for Nonmonotonic Knowledge Bases 1
Thomas Eiter, Michael Fink, Giuliana Sabbatini, and Hans Tompits
1 Introduction 1
2 Preliminaries 4
3 Framework and Language 6
4 Properties 17
5 Implementation and Update Agents 29
6 Extensions 32
7 Related Work 37
8 Conclusion 41
References 42

Declarative Update Policies for Nonmonotonic Knowledge Bases

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

Technische Universität Wien, Institut für Informationssysteme 184/3,
Favoritenstraße 9-11, A-1040 Vienna, Austria
email: [eiter,giuliana,michael,tompits]@kr.tuwien.ac.at

Abstract. Updating databases, and in particular relational databases, is a central issue that 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 chapter, we present a generic framework for declarative specification of update policies, that 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 that are dependent on the information received, the current 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 Introduction

Since the 1980s, the problem of updating logical data and knowledge bases has been studied by many researchers, and a number of different approaches have been developed (see [21,40] and [19] for accounts of early methods). In these investigations, a knowledge base is a collection of formulas or rules, and an update consists of another set of such objects. The problem then is how to incorporate the update in the given knowledge base.

For instance, given a knowledge base, K, containing some statement $best_buy(shop_1)$, and assuming that there is a constraint expressing that K should include only one atom of form $best_buy(X)$, an update of K with formula $best_buy(shop_2)$ forces the deletion of $best_buy(shop_1)$. However, in general, a completely automated approach to updating knowledge bases is rather complex, and sometimes there is not enough information to perform an update automatically. View updates are a case in point. As an illustration, suppose that a database contains

$p(a)$,
$q(a)$, and
$r(X) \leftarrow p(X), q(X)$.

Given no further information, the atom $r(a)$ can be inferred. But if $r(a)$ should be deleted from the database, the question is, should $p(a)$ be removed, or $r(a)$, or even both? Methods to handle this problem, like, for example, [25] (see also [8,21]), involve principles used in artificial intelligence which would discard removing both $r(a)$ and $p(b)$ and leave the other options. However, this may be undesired, and the most appropriate removal option may depend on the particular application and on user-specific preferences, which are not respected by an automated general-purpose update procedure.

To avoid the difficulties of a completely automated approach, *update policies* allow users to specify *how* updates should be performed. 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. A realization of ideas in this direction is *active databases* [39], in which the dynamics of a database is specified by means of *event-condition-action* (*ECA*) *rules* that are triggered by events. However, most of these methods are ad hoc, lacking a clearly defined declarative semantics, and, moreover, complex conditions can, in general, not be expressed. Update policies, on the other hand, should allow the specification of update requests in a designated policy language, providing suitable flexibility by taking into account information from the environment and the current knowledge base. We note that similar approaches have also been developed in the area of network management and security [32,13,12,9].

Besides updating classical, monotonic knowledge bases, the issue of updating *nonmonotonic knowledge bases* has received growing interest in recent years, in particular, where the knowledge base is represented in terms of (nonmonotonic) logic programs (see [1,2,14,18,28,36,42]). For nonmonotonic knowledge bases of the latter kind, 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 that is missing in LUPS, but is indispensable for the use 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 a discrete way, communicated by the environment, for example, by some cooperating agent or by some sensing device.

In this chapter, we present a generic framework for the specification of declarative update policies and a language called EPI that builds upon LUPS and provides a number of novel aspects. EPI allows 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. Beside its flexibility and expressivity, the advantages of this approach are, on the one hand, the generality, because of its use of a declarative language which is neither tailored to a specific application domain nor ad hoc for special purposes, and, on the other hand, its modularity, which allows instantiating 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 [38].

The main contributions of our work can be 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 that are to be performed on the current knowledge base. At the next layer, U is compiled into 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, determining 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 an event, is provided and can be used for triggering the execution of change actions. Recalling our best-buy example from the above, 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 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, which could be of use in practical applications, and could provide 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 that

are satisfied by the update answer set semantics of [14,18] 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. Though, unsurprisingly, reasoning is intractable in general, due to the 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 [38], 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, 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 remainder of this chapter is structured as follows. The next section contains some preliminaries and fixes notation. Our framework for update policies is then presented in Sect. 3. After that, we study in Sect. 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 multiagent platform [38]. Possible extensions of the EPI language are discussed in Sect. 6, which is followed by a comparison to LUPS and other related work, in particular to $\mathcal{PDL}$ [32]. The final Sect. 8 outlines some issues for further research. Some of the results presented here appeared in preliminary form in [15,16].

2 Preliminaries

We consider knowledge bases represented by extended logic programs under the answer set semantics. For further background and unexplained concepts, we refer to [7,31] and references therein.

An *extended logic program* (ELP) [23] is a finite set P of rules r of form

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

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 $H(r)$. The set $B(r) = \{L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n\}$ is the *body* of r. Accordingly, 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; if $B(r) \neq \emptyset$ and $H(r)$ is absent, r is called a *constraint*.

The *complement*, $\neg L$, of a literal L is A if $L = \neg A$ and $\neg A$ if $L = A$. For any set S of literals, we define $\neg S = \{\neg L \mid L \in S\}$. 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 [5], where we 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 $Bel_{\mathcal{A}}(\cdot)$, which associates with every sequence $\boldsymbol{P}$ over $\mathcal{A}$ a set $Bel_{\mathcal{A}}(\boldsymbol{P}) \subseteq \mathcal{L}_{\mathcal{A}}$ of rules. Intuitively, $Bel_{\mathcal{A}}(\boldsymbol{P})$ are the consequences of $\boldsymbol{P}$. We refer to $Bel_{\mathcal{A}}(\boldsymbol{P})$ as the *belief set* of $\boldsymbol{P}$ (if no ambiguity can arise, we usually drop the subscript "$\mathcal{A}$" in $Bel_{\mathcal{A}}(\cdot)$). Different instantiations of Bel are possible, according to various proposals for update semantics in the literature [2,14,18,42,36,33]. We only assume that $Bel(\cdot)$ 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 $\boldsymbol{P} = (P_1, \ldots, P_n)$, which holds for most knowledge-base update approaches (see [2,14,36], and [18,26]).
2. *Invariance with respect to 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 a program P or an update sequence $\boldsymbol{P}$ with variables is defined in terms of their grounded versions $\mathcal{G}(P)$ and $\mathcal{G}(\boldsymbol{P})$ over the Herbrand universe, respectively.

In the following, we use for illustration the answer set semantics for update programs from [18], which coincides with the semantics of inheritance programs in [11] 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 [23] iff it is a minimal model of the *reduct*

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

We denote by $\mathcal{AS}(P)$ the collection of all answer sets of P. Following [18], 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

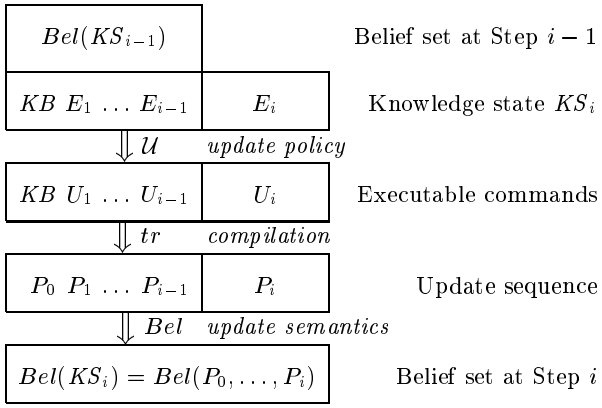

Fig. 1. From knowledge state to belief set at Step i

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 $\mathcal{AS}(\boldsymbol{P})$. Because $n = 1$ 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 [18].

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 update program (P_0, P_1) has the 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 its unique answer set with $Rej(S_2, (P_0, P_1, P_2)) = \{c \leftarrow, a \leftarrow\}$.

In the context of the update semantics of [18], $Bel_{\mathcal{A}}(\boldsymbol{P})$ is the set of all rules entailed under cautious answer set semantics, i.e. $Bel_{\mathcal{A}}(\boldsymbol{P}) = \{r \in \mathcal{L}_{\mathcal{A}} \mid S \models r \text{ for all } S \in \mathcal{AS}(\boldsymbol{P})\}$. 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 $Bel_{\mathcal{A}}(P_1, \ldots, P_n)$ instead of $Bel_{\mathcal{A}}((P_1, \ldots, P_n))$.

3 Framework and Language

We first describe our generic framework for event-based updating, and then the language EPI ("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, expressing, for example, the current date, weather, or sale information. More complex events in the form of logic program rules can also be thought of, for example, 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 $E_i \in \mathcal{EC}$, $1 \leq i \leq n$. Furthermore, for $0 \leq i \leq n$, the knowledge state $KS_i = \langle KB; E_1, \ldots, E_i \rangle$ 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 that the agent incorporates the event E_i 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 Fig. 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. Afterward, 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 then 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 by permitting 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 the 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 the form

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

Table 1. Syntax of an update statement in EPI

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

which expresses conditional assertion or retraction of a rule ρ_1, given by $cmd_1(\rho_1)$, depending on commands [**not**]$cmd_2(\rho_2)$, ...,[**not**]$cmd_m(\rho_m)$, and conditioned with the provisos whether c_1 belongs to the current belief set and whether c_2 is in the actual event. Each condition in "[[...]]", both of the form c_1 and of the form **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 Sect. 7.1), plus the additional command **ignore**, which allows us 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 reasserted. The option **event**[1] states that such an inertial effect is unwanted and the rule has to be asserted (respectively, retracted) *in the given step of the evolution*, and to be retracted (respectively, reasserted) afterward. The command **always**, referred to as a *persistent command*, is meant to assert a rule in the current step and, *unconditionally*, in each successive update step, until the command is itself deactivated; this, in turn, is accomplished by a **cancel** command. Similarly as above, the option **event** for **always** means that the insertion is volatile and the rule is not supposed to persist by inertia, i.e. as soon as a repeated insertion is stopped, the rule will be retracted in the next step. Note that **always_event** is different from **assert_event**: the former command initiates unconditional volatile addition of a rule in follow-up states until terminated, whereas the latter expresses conditional volatile addition of the rule which is evaluated in each step.

This is just a superficial exposition of the update commands; of course, complex interactions between them can be originated, which we ought to

[1] There may arguably be better choices to denote a nonpersistent installation of rules, but to retain previous notation, we use the **event** option here.

carefully consider in our semantic definition to avoid unintended conflicts and "side effects." For details, we refer to the precise definition below (Definition 5).

Note that, in LUPS, conditions on rules and external events cannot 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 are now in a position to 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

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

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

Examples

Let us now have a look at two examples, which will be used throughout this chapter.

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

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

and a fact $r_0 : date(0)$ is an initial time stamp. Here, r_1 expresses that a shop S, which runs 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, may 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 $\mathcal{U}$ can be defined as follows. Assume that it contains the incorporate-by-default statement (1), as well as

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

Informally, the first statement repeatedly confirms the information about a future sale, which guarantees that it is effective on the given date, and the second statement revokes this. The third one removes information about a

previously ended sale (assuming that the time stamps increase). Furthermore, $\mathcal{U}$ also includes the following statements:

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

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

Let us now consider a more complex example.

Example 3 (Mailing agent). Consider a mailing agent that has the following initial knowledge base KB, whose rules are instantiated over suitable variable domains:

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

The knowledge base KB provides a classification of message types depending on the subject (rules r_1 and r_2). By means of r_3, a default type is assigned to all messages that are neither *private* nor *business*. Automatic message removal is performed by r_4, when a message is older than thirty days and not saved yet, providing it is not an unread message (signaled by the atom $new_msg(\cdot)$). Rules r_5, r_6, and r_7 are used to check for all messages of a given type whose removal has not been asked for and to signal if at least one such message has been found (*success*) or not (*failure*).

An event E may 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 considered new);
- at most, one fact $search(t)$ for 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 $\mathcal{U}$ may 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):

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

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

$$\textbf{assert}(new_msg(M))\ [\![\mathbf{E} : from(M,N), date(M,D)]\!];$$
$$\textbf{assert}(msg(M))\ \textbf{if assert}(new_msg(M)),$$

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

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

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

$$\textbf{assert}(from(M,N))\ \textbf{if assert}(new_msg(M))\ [\![\mathbf{E} : from(M,N)]\!];$$
$$\textbf{assert}(subject(M,S))\ \textbf{if assert}(new_msg(M))\ [\![\mathbf{E} : subject(M,S)]\!];$$
$$\textbf{assert}(date(M,D))\ \textbf{if assert}(new_msg(M))\ [\![\mathbf{E} : date(M,D)]\!].$$

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

$$\textbf{assert}(save(M))\ [\![\mathbf{E} : save(M)]\!];$$
$$\textbf{assert}(remove(M))\ [\![not\ new_msg(M),\ \mathbf{E} : remove(M)]\!];$$
$$\textbf{ignore}(remove(M))\ [\![save(M)]\!];$$
$$\textbf{ignore}(remove(M))\ \textbf{if assert}(save(M));$$
$$\textbf{retract}(msg(M))\ [\![remove(M)]\!];$$
$$\textbf{retract}(msg(M))\ \textbf{if assert}(remove(M)).$$

The effect is to save or remove a message if an external request comes in, whereas the removal is possible only 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 anylonger, since 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 simple commands.

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

$$\textbf{assert_event}(search(T))\ [\![\mathbf{E} : search(T)]\!];$$
$$\textbf{retract}(found(T))\ [\![found(T)]\!];$$
$$\textbf{retract}(success)\ [\![success]\!];$$
$$\textbf{retract}(failure)\ [\![failure]\!].$$

Using the signal atom $new_msg(\cdot)$, it is possible, for example, to activate a reminder for users that they still have unread messages:

$$\begin{array}{l}\textbf{always}(new_mail!)\ \textbf{if}\ \textbf{assert}(new_msg(M));\\ \textbf{cancel}(new_mail!)\ \textbf{if}\ \textbf{retract}(new_msg(M)),\ \textbf{not}\ \textbf{assert}(new_msg(M'))\\ \qquad\qquad [\![not\ new_msg(M''), M'' \neq M]\!].\end{array}$$

Although 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, for example,

$$\begin{array}{l}\textbf{assert_event}(\ remove(M) \leftarrow\ not\ new_msg(M),\\ \qquad\qquad\qquad\qquad not\ type(M, business)\ )\ [\![\mathbf{E} : max_size]\!],\end{array}$$

which determines the cancellation of all messages that have already been read and that do not concern business issues, providing that 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 Fig. 1. First of all, the update policy is *evaluated* over the current state, i.e. the conditions in "$[\![\ldots]\!]$" are checked for each of their ground instances. Only those statements are sorted out for which the respective conditions are satisfied. Viewing them as a logic program, its answer sets are computed, resulting in a set of executable commands. In a second step, a compilation function encodes the effects of update commands over the initial knowledge base into an update sequence, such that the belief set of the resultant sequence is as intended.

Executable Commands

According to the semantics of EPI, as depicted in Fig. 1, at Step i, we first determine the set U_i of executable commands, 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

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

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

$$cmd_1(\rho_1)\ \textbf{if}\ [\textbf{not}]\ cmd_2(\rho_2), \ldots, [\textbf{not}]\ cmd_m(\rho_m). \tag{2}$$

Example 4. Consider again the shopping agent of Example 2. If the initial knowledge base KB is as described in the example, athe first event E_1 contains $date(1)$, and

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

then, $\mathcal{G}(\mathcal{U})^1$ comprises the following update statements:

$\textbf{assert}(date(1))$ **if not** $\textbf{ignore}(date(1))$
$\textbf{assert}(r_4)$ **if not** $\textbf{ignore}(r_4)$;
$\textbf{always}(r_4)$ **if** $\textbf{assert}(r_4)$;
$\textbf{retract}(date(0))$;
$\textbf{ignore}(r_4)$.

Semantically, we interpret update statements of form (2) 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 then defined as the collection of all of these rules, given $\mathcal{G}(\mathcal{U})^i$, together with the following constraints, which exclude contradictory commands:

$\leftarrow \textbf{assert}[\textbf{_event}](R), \textbf{retract}[\textbf{_event}](R)$;
$\leftarrow \textbf{always}[\textbf{_event}](R), \textbf{cancel}(R)$.

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

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

Because update statements do not contain strong negation, executable update commands are in fact *stable models* of $\mathcal{G}(\Pi_i^{\mathcal{U}})$ [22]. Furthermore, because 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 $U_i = \{\textbf{assert}(\bot_i \leftarrow)\}$, 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. Note that such a selection function can be implemented very easily if one of the possible solutions is simply chosen, for example, the first solution computed, etc. This is very similar to the selection of one truth assignment for a formula of classical logic by a SAT solver.

Compilation to Update Sequences

Next, the sets $U_1, \ldots, U_i$ of executable commands 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 long as different transformations 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 is more readable (see Sect. 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 more insight into 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 that are to be inserted, activated, and deactivated, respectively, in the resultant knowledge base.

Definition 5. Let $(KB; U_1, \ldots, U_i)$, $i \geq 0$, be a sequence consisting of an initial knowledge base KB and sets $U_1, \ldots, U_i$ of executable commands. Then, $tr(KB; U_1, \ldots, U_i) = (P_0, P_1, \ldots, P_i)$ is inductively defined as follows:

$i = 0$: 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:

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

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

P_i is computed based on 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 defined as follows:

$$
\begin{aligned}
Add_i &= \{r \mid \mathbf{assert}[_\mathbf{event}](r) \in EC_i \vee \mathbf{always}[_\mathbf{event}](r) \in PC_i\};\\
Enable_i &= \{r \mid \mathbf{retract_event}(r) \in EC_{i-1} \wedge \mathbf{retract}(r) \notin EC_{i-1} \wedge\\
&\qquad \mathbf{retract}[_\mathbf{event}](r) \notin EC_i \wedge (\mathbf{always_event}(r) \notin PC_{i-1}\\
&\qquad \vee \mathbf{cancel}(r) \notin EC_i)\};\\
Disable_i &= \{r \mid \mathbf{retract}[_\mathbf{event}](r) \in EC_i \wedge \mathbf{always}[_\mathbf{event}](r) \notin PC_i\} \cup\\
&\quad \{r \mid \mathbf{cancel}(r) \in EC_i \wedge \mathbf{always_event}(r) \in PC_{i-1}\\
&\qquad \wedge\ \mathbf{assert}[_\mathbf{event}](r) \notin EC_i\} \cup\\
&\quad \{r \mid \mathbf{assert_event}(r) \in EC_{i-1} \wedge \mathbf{assert}(r) \notin EC_{i-1}\\
&\qquad \wedge\ \mathbf{always}[_\mathbf{event}](r) \notin PC_i\ \wedge\ \mathbf{assert}[_\mathbf{event}](r) \notin EC_i\}.
\end{aligned}
$$

Note that the sets Add_i, $Enable_i$, and $Delete_i$ are similar to the *Add* and *Delete* lists in STRIPS planning. Observe also that, for optimization purposes, Add_i could be reduced by moving the rules that already exist in KB to $Enable_i$, thus avoiding unnecessary repeated insertion of rules.

The transformation tr introduces in the body of each rule r a control atom, $on(n_r)$ (where $on(\cdot)$ is a reserved predicate), which is used to activate or deactivate the rule as requested. Alternatively, one could insert or delete rules directly in the programs of the update sequence. However, using a control atom simplifies the management effort and is supportive to *incremental transformations* (see Definition 7), where the result of the transformation for updates $U_1, \ldots, U_{n+1}$ on KB is obtained by adding a program P_{n+1} to the sequence of programs $(P_0, \ldots, P_n)$ which is the result of the transformation for the updates $U_1, \ldots, U_n$ on KB. Note that physical deletion of rules may be done as described in Sect. 4.1. Furthermore, a control atom permits us to determine *deductively within the update language* the "physical" presence of a rule in the knowledge base and to use this in update programs (if desired).

Given the set U_i of executable commands, first those commands are filtered out that 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 **ignore** commands is simply to *skip* unintended updates. For example, if a certain rule serves as a *tick of the clock* for the agent, updates to this rule can be avoided to preserve the integrity of behavior.

The set PC_i of *persistent commands* contains those update commands that affect more than one step of evolution. 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, independently of other previous noninertial insertions.

Then, while building P_i, simple syntactical checks avoid some possible conflicts that may arise either between commands in successive updates or between persistent and nonpersistent commands. Conflicts caused by successive updates are solved in favor of new update commands, whereas conflicts between persistent and nonpersistent commands are solved in favor of the former. 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:

- **retract_event**$(r) \in EC_{i-1} \wedge$ **retract**[_**event**]$(r) \in EC_i$: The option **event** for the **retract** command at Step $i-1$ requires that r be retracted *only* at that step; thus the rule should be reactivated 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 a higher priority to the commands in EC_i, and thus $on(n_r)$ is not added.
- **retract**[_**event**]$(r) \in EC_i \wedge$ **always**[_**event**]$(r) \in PC_i$: The **retract**(r) command requires adding $\neg on(n_r)$; the **always**[_**event**] command for the same rule r in PC_i, on the other hand, requires adding $on(n_r)$. Thus, a conflict would arise. This is avoided by assigning a higher priority to the command in PC_i, and hence $\neg on(n_r)$ is not added.
- **always_event**$(r) \in PC_{i-1} \wedge$ **assert**[_**event**]$(r) \in EC_i \wedge$ **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 deactivating 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 a higher priority to the last incoming command in EC_i, and thus $\neg on(n_r)$ is not added.
- **assert_event**$(r) \in EC_{i-1} \wedge$ **assert**[_**event**]$(r) \in EC_i$: The option **event** for the **assert** command at Step $i-1$ requires that r 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 a higher priority to the command in EC_i, and thus $\neg on(n_r)$ is not added.
- **assert_event**$(r) \in EC_{i-1} \wedge$ **always**[_**event**]$(r) \in PC_i$: The option **event** for the **assert** command at Step $i-1$ requires that r 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 a 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 **always**(r) is already present in PC_{i-1}, then adding $on(n_r)$ is enough, because the rule is already in the knowledge base.

Based on this compilation, we define the belief set for a knowledge state:

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 5. Reconsider Example 2, and suppose that the event $E_1 = \{r_1' : sale(s_0),\ r_2' : date(1)\}$ 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 $PC_1 = PC_0 \setminus \{\mathbf{assert_}[\mathbf{event}](date(0))\} = \emptyset$ 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 EPI framework for update policies.

4.1 Structural Properties

We first discuss some properties of $Bel(KS)$ for particular update policies, using the definition of Bel based on the update answer sets approach of [18], as explained in Sect. 2. We stress that analogous properties hold for similar instantiations of Bel, such as, for example, dynamic logic programs [2].

We start with some basic properties. Recall that we suppose that the two properties stated in Sect. 2 are satisfied by any "reasonable" belief function in our setting.

Definition 7. A compilation function, tr, is *incremental* iff for each sequence $(KB; U_1, \ldots, U_n)$, where KB is an ELP and each U_i, $1 \leq i \leq n$, is an update command, $tr(KB; U_1, \ldots, U_n) = (P_0, \ldots, P_n)$ 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 from Definition 5 is incremental.*

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

Definition 8. 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 update sequences $\boldsymbol{P}, \boldsymbol{P}'$, 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 is called *contracting under empty updates*;, if only the second condition holds, the belief function is called *contracting under rule repetition.*

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

Proof. To prove Property 1, we have to show that, given two update sequences $(P_0, \ldots, P_m)$ and $(P_{m+2}, \ldots, P_n)$, $Bel(P_0, \ldots, P_m, \emptyset, P_{m+2}, \ldots, P_n) = Bel(P_0, \ldots, P_m, P_{m+2}, \ldots, P_n)$ holds. Let us denote the first sequence by $\boldsymbol{P}^{\emptyset}$ and the second by $\boldsymbol{P}$. The rejection sets for the two sequences coincide; 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 that 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 $0 \leq i \leq m$, can it be the case that the rejection condition is satisfied by a rule $r' \in P_j$ with $j = m+1$, because $P_{m+1} = \emptyset$, 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 not affected by P_{m+1} at all, so $\bigcup_{i=m+2}^{n} Rej_i(S, \boldsymbol{P}^{\emptyset}) = \bigcup_{i=m+2}^{n} Rej_i(S, \boldsymbol{P})$ holds as well. Therefore, $Rej(S, \boldsymbol{P}^{\emptyset}) = Rej(S, \boldsymbol{P})$, 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.

For 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, because $j \geq i$. Therefore, $r_j \notin Rej(S, \boldsymbol{P})$. This means that the occurrence r_j would not be rejected as well and the semantics is not affected by deleting r_i. □

The same result holds for dynamic logic programs. 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 function tr, clearly $P_i = \emptyset$ holds as well, yielding $Bel(KS_i) = Bel(P_0, \emptyset, \ldots, \emptyset) = Bel(P_0)$, in view of the assumption that Bel is contracting under empty updates, and thus, by the definition of tr, $Bel(KS_i) = Bel(KB)$. □

The next property states that, in case of the unconditional assert policy, the update sequence consisting of the initial knowledge base and the extended logic programs comprising the events can be used to determine the belief set of a given knowledge state.

Proposition 4 (Unconditional assert policy). *For any knowledge state $KS = \langle KB; E_1, \ldots, E_n \rangle$ and given $\mathcal{U} = \{\mathbf{assert}(r)\ [\![\mathbf{E} : r]\!]\}$, $Bel(KS_i) = Bel(KB, E_1, \ldots, E_i)$, 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, $Bel(KS_i) = Bel(P_0, P_1, \ldots, P_i) = Bel(KB, E_1, \ldots, E_i)$. □

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, and consider $r \in P_i$ and $r' \in P_j$ with $i < j$, such that one of the following conditions holds, where L is ground:*

1. *$r = r'$,*
2. *$r = L \leftarrow$, $r' = \neg L \leftarrow$, and either L is not unifiable with any rule head in $P_{j+1} \cup \cdots \cup P_n$, or $L \leftarrow \ \in P_{j+1} \cup \cdots \cup P_n$,*
3. *$\neg L \in B(r)$, $r' = L \leftarrow$, 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. For Condition 1, the result follows by Proposition 2. For Condition 2, suppose for simplicity that $\boldsymbol{P}$ is ground (the nonground case is easily reduced to the ground one), and consider any update answer set S of $\boldsymbol{P}$ (respectively, 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. if $r' \notin Rej_j(S, \boldsymbol{P})$ (respectively, $r' \notin Rej_j(S, \boldsymbol{P}')$), then $r \in Rej(S, \boldsymbol{P})$ (resp., $r \in Rej(S, \boldsymbol{P}')$), 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., of $\boldsymbol{P}$). Otherwise, if r' is rejected, i.e. if $r' \in Rej_j(S, \boldsymbol{P})$ (respectively, $r' \in Rej_j(S, \boldsymbol{P}')$), then, by hypothesis, some r'' of form $L \leftarrow$ is in $P_{j'}$, where $j' > j$. Hence, $r'' \notin Rej_{j'}(S, \boldsymbol{P})$ (resp., $r'' \notin Rej_{j'}(S, \boldsymbol{P}')$), which means that $\bigcup_i P_i \setminus Rej(S, \boldsymbol{P}) = \bigcup_i P'_i \setminus Rej(S, \boldsymbol{P}')$. Therefore, S is an answer set of $\boldsymbol{P}'$ (respectively, of $\boldsymbol{P}$).

Finally, in case of Condition 3, r' can not be rejected, 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. □

Theorem 1. *Let KS be a knowledge state such that $Bel(KS) = Bel(\boldsymbol{P})$, where $\boldsymbol{P} = (P_0, \ldots, P_n)$ is the compiled update sequence. Furthermore, for $j \geq 0$, let $\boldsymbol{P}^j$ be the result of applying the rule removal procedure j times on $\boldsymbol{P}$, 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*

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

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

Proof. Condition 1 follows from Lemma 1 by straightforward induction on $j \geq 0$. As for Condition 2, in view of Lemma 1, 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 $\bot_i$, for $0 \leq i \leq n$, and the latter consisting of literals of the form $(\neg)on(n_r)$. Now, the latter kind of literals can occur in the head of facts only and because $\boldsymbol{P}^*$ does not contain contradictory facts (due to Condition 2 of Lemma 1), part S_{on} is identical for each $S \in \mathcal{AS}(\boldsymbol{P}^*)$.

Let us call a set S^* *completing* for $\boldsymbol{P}^*$ iff $S^* = S \setminus S_{on}$, for an answer set S of $\boldsymbol{P}^*$. 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. □

Consequently, 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.

History Contraction

Another relevant issue is the possibility, for some special case, of *contracting the agent's update history* and of computing 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: (i) $P_0 = KB$, and (ii) the construction of each P_i, $i > 0$, 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. *For any update sequence $\boldsymbol{P} = (P_0, \ldots, P_n)$ such that each P_i, $1 \leq i \leq n$ contains (ground) facts only, $Bel_{\mathcal{A}}(\boldsymbol{P}) = Bel_{\mathcal{A}}(P_0, P_{u_n})$ holds, 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. We show the result 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 all

sequences $(Q_0, \ldots, Q_k)$ with $k < n$. Now, by the semantics of update sequences, $Bel(P_0, \ldots, P_{n-1}, P_n) = Bel(P_0, \ldots, P_{n-2}, P'_{n-1})$, where $P'_{n-1} = P_n \cup (P_{n-1} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P_n\})$. Applying the induction hypothesis to $(P_0, \ldots, P_{n-2}, P'_{n-1})$, we obtain $Bel(P_0, \ldots, P_{n-2}, P'_{n-1}) = Bel(P_0, P'_{u_{n-1}})$, with $P'_{u_{n-1}} = P'_{n-1} \cup (P_{u_{n-2}} \setminus \{L \leftarrow \mid \neg L \leftarrow \in P'_{n-1}\})$. Substituting the definition of P'_{n-1} in this equation and combining it with the general construction of P_{u_i}, we finally get $Bel(P_0, P'_{u_{n-1}}) = Bel(P_0, P_{u_n})$. Thus, $Bel(P_0, \ldots, P_n, P_n) = Bel(P_0, P_{u_n})$, as intended. □

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 $Bel_{\mathcal{A}}(P_0, P_{u_n}) = Bel_{\mathcal{A}}(KB, P_{u_n})$. □

4.2 Expressive Core

In what follows, we identify an *expressive core* of the language EPI, that is, a set of *primitives* that possesses the same expressivity as the whole language. We describe this core and show how every update policy over the original language can be transformed into an *equivalent* update policy using only these primitives. We denote the core language by $\mathsf{EPI}_{\text{core}}$, 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 $\mathsf{EPI}_{\text{core}}$ one is standard and modularized. The proposed transformation also enjoys some structural properties that allow us to derive equivalence and complexity results. The equivalence result means that the evolution depends actually only on the current belief set and on the incoming event, but 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 $KS = \langle KB; E_1, \ldots, E_n \rangle$, $Bel_{\mathcal{A}}(tr(KB, U_1, \ldots, U_n)) = Bel_{\mathcal{A}}(tr(KB, U'_1, \ldots, U'_n))$ holds, 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 only how such a policy $\mathcal{U}'$ is obtained; the actual proof of Theorem 3 can be found in [35]. The idea is to rewrite the update policy $\mathcal{U}$ in its new version $\mathcal{U}'$ such that the latter is made up of three different subprograms, yielding 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 update 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 (see the relevant modularity properties of logic programs discussed in [20,30]).

The semantics of knowledge states is defined in the usual way; applying tr to the sets $U_1', \ldots, U_n'$ of executable commands 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 two new atoms as follows for each command occurring in $\mathcal{U}$:

- for each command **assert**[_**event**](r), atoms $ASR_0(r)$ and $ASR(r)$ (respectively, $ASR_EV_0(r)$ and $ASR_EV(r)$ for the option **event**);
- for commands of the form **retract**[_**event**](r), $RET_0(r)$ and $RET(r)$ [respectively, $RET_EV_0(r)$ and $RET_EV(r)$ for the option **event**];
- for **always**[_**event**](r) commands, $ALW_0(r)$ and $ALW(r)$ [respectively, $ALW_EV_0(r)$ and $ALW_EV(r)$];
- for **cancel**(r), $CAN_0(r)$ and $CAN(r)$; and
- for **ignore**(r), atoms $IGN_0(r)$ and $IGN(r)$.

Intuitively, as will become clear by analyzing the content of Module **A**, the new atoms with subscript "0" (let us denote them by $\alpha_0(\cdot)$) serve to simulate the computation of the set U_i by $\mathcal{U}$. Given a rule r, $\alpha_0(r)$ is only asserted in $\mathcal{U}'$ in case the corresponding command belongs to U_i. Moreover, these special atoms that are new and do not appeare 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 $\mathcal{U}'$ comprises 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 $cmd(r)$ is replaced by **assert**$(atom_cmd_0(r))$, where $atom_cmd(r)$ is the new atom corresponding to $cmd(r)$, as introduced above.

(A.2) Additional statements are added to preserve consistency, playing the role of the integrity constraints. For each pair of commands α_1, α_2, 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 added, where *dummy* is a reserved atom:

$$\mathbf{assert}(dummy) \textbf{ if } \alpha_1, \alpha_2;$$
$$\mathbf{retract}(dummy) \textbf{ if } \alpha_1, \alpha_2.$$

(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 $\beta \in \{ASR, ASR_EV, RET, RET_EV, CAN\}$, the following statements are added to $\mathcal{U}'$:

$$\mathbf{retract}(\beta(r)) \textbf{ if not } \mathbf{assert}(\beta_0(r)) \ [\![\beta(r)]\!];$$
$$\mathbf{assert}(\beta(r)) \textbf{ if } \mathbf{assert}(\beta_0(r)), \textbf{ not } \mathbf{assert}(IGN_0(r)).$$

(B.2) For each $\beta \in \{ALW, ALW_EV\}$, the following statement is added to $\mathcal{U}'$:

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

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

$$\begin{aligned}
\mathbf{assert}(ALW(r)) \textbf{ if } & \mathbf{assert}(ALW_0(r)), \\
& \textbf{not } \mathbf{retract}(ALW(r)), \\
& \textbf{not } \mathbf{assert}(IGN_0(r));
\end{aligned}$$

$$\begin{aligned}
\mathbf{assert}(ALW_EV(r)) \textbf{ if } & \mathbf{assert}(ALW_EV_0(r)), \\
& \textbf{not } \mathbf{retract}(ALW_EV(r)), \\
& \textbf{not } \mathbf{assert}(ALW_0(r)), \\
& \textbf{not } \mathbf{assert}(IGN_0(r))[\![not\ ALW(r)]\!];
\end{aligned}$$

$$\mathbf{retract}(ALW_EV(r)) \textbf{ if } \mathbf{assert}(ALW_0(r)), \textbf{not } \mathbf{assert}(IGN_0(r));$$
$$\mathbf{retract}(ALW_EV(r)) \textbf{ if } \mathbf{assert}(CAN_0(r)), \textbf{not } \mathbf{assert}(IGN_0(r));$$
$$\mathbf{retract}(ALW(r)) \textbf{ if } \mathbf{assert}(CAN_0(r)), \textbf{not } \mathbf{assert}(IGN_0(r)).$$

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

(C.1) Set Add_i:

$$\mathbf{assert}(r) \textbf{ if } \mathbf{assert}(ASR(r));$$
$$\mathbf{assert}(r) \textbf{ if } \mathbf{assert}(ASR_EV(r));$$
$$\mathbf{assert}(r) \textbf{ if } \mathbf{assert}(ALW(r));$$
$$\mathbf{assert}(r) \textbf{ if } \mathbf{assert}(ALW_EV(r)).$$

(C.2) Set $Enable_i$:

$$\begin{aligned}
\mathbf{assert}(r) \textbf{ if } & \textbf{not } \mathbf{assert}(RET(r)), \textbf{ not } \mathbf{assert}(RET_EV(r)) \\
& [\![RET_EV(r), not\ RET(r), not\ ALW_EV(r)]\!];
\end{aligned}$$

$$\begin{aligned}
\mathbf{assert}(r) \textbf{ if } & \textbf{not } \mathbf{assert}(RET(r)), \textbf{ not } \mathbf{assert}(RET_EV(r)), \\
& \textbf{not } \mathbf{assert}(CAN(r))[\![RET_EV(r), not\ RET(r)]\!].
\end{aligned}$$

(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 that 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 is factual, and also the stratifiability (defined in Sect. 4.3 below) of the original policy is preserved. Due to complexity properties, these are 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. In particular, we analyze the following reasoning task:

KS-INF: Given a ground update policy $\mathcal{U}$, a ground rule r, and a ground knowledge state $KS = \langle KB; E_1, \ldots, E_n\rangle$, decide whether $r \in Bel(KS)$, i.e. decide whether r holds in the knowledge base after events $E_1, \ldots, E_n$.

The complexity of this task is of particular interest if updates are delayed and incorporated *demand driven*, rather than instantaneously on each update occurrence, to facilitate faster query answering.

Clearly, the complexity of the above problem crucially depends on the complexity of computing the set Bel, which is determined by the underlying update semantics. We consider here the update answer sets of [14,18]; similar results hold for dynamic logic programs [2]. At a generic level, access to Bel might be modeled by an oracle that is available to the reasoning process; part of the results below (in particular, upper bounds) can be easily reformulated for such relativized computations. We remark that for other approaches, for example, abductive updates [36], the results change, as the underlying semantics lies at the next level of the polynomial hierarchy. Other complexity results on reasoning in a general update framework appeared in [17].

Besides r, KS, and $\mathcal{U}$, the selection function $Sel(\cdot)$, whose purpose is to select one in case of multiple sets of executable commands (see Definition 4

in Sect. 3.3), contributes to the construction of $Bel(KS)$. Therefore, 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 (for example, lexicographic preference has this property).

However, it might happen that the precise $Sel(\cdot)$ employed is unknown to the user or that it is difficult to assess (like, for example, which satisfying assignment a SAT solver returns if multiple such assignments exist, in case the solver makes input-dependent choices). To account for this, we cautiously consider all possible realizations for $Sel(\cdot)$ that emerge from the different answer sets of the programs $\Pi_i^{\mathcal{U}}$. We refer to this case as "unknown $Sel(\cdot)$."

Unsurprisingly, because inference from logic programs under the answer set semantics is well-known to be intractable, KS-INF for update answer sets is intractable as well. However, the answer set semantics is tractable for the important class of stratified programs that have at most one answer set which, moreover, is computable in polynomial time. Toward a class of tractable update policies, this motivates the following definition.

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, and its computational cost is zero. Furthermore, the answer set is computable in polynomial time.

The benign properties of a stratified initial knowledge base KB are easily lost if updates by arbitrary rules take place, which might destroy stratification. Updates need to be restricted to prevent this. A class of update policies that preserve stratification are factual update policies, which we have introduced above. Thus, we pay special attention to such policies in our analysis.

In total, we consider the following three orthogonal syntactic criteria: KB stratified, $\mathcal{U}$ stratified, and $\mathcal{U}$ factual.

The following result about reasoning from an update program under the update answer set semantics is easily established (see 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*, to decide 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.

Theorem 4. *Let Bel be the update answer set semantics. Then, the problem* KS-INF *is* (*i*) *in* P^{NP}*, assuming that* $Sel(\cdot)$ (*which may be part of the input*) *is polynomial-time computable with an* NP *oracle; and* (*ii*) *polynomial, if KB is stratified and* $\mathcal{U}$ *is factual and stratified.*

Proof. (i) Each U_i and P_i, as in Fig. 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. Because 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}.

(ii) If KB is stratified and $\mathcal{U}$ is factual and stratified, $U_i = Sel(\Pi_i^{\mathcal{U}})$ is computable in polynomial time, P_0 in $Bel(P_0, \ldots, P_{i-1})$ is stratified, and each P_j, $1 \leq j \leq i-1$, contains only facts. Thus, by Proposition 5, the NP oracle in (i) can be replaced by a polynomial-time procedure, and each $\Pi_i^{\mathcal{U}}$ is computable in polynomial time. This proves the result. □

Note that Part (ii) of the previous result identifies a tractable case of the problem. As the following results entail, this is the only one within the range of subcases that we consider.

Theorem 5 (General KB with stratified $\mathcal{U}$). *The problem* KS-INF *for stratified $\mathcal{U}$ is* P^{NP}*-complete. Moreover,* P^{NP}*-hardness holds even if $\mathcal{U}$ is also factual.*

Proof. By Theorem 4(i), it remains to show hardness. We encode the problem of finding the last bit (denoted α_n^*) of the lexicographically maximal model of a formula ϕ in conjunctive normal form (CNF), which is well-known to be P^{NP}-hard [34], into KS-INF. Let ϕ be defined by clauses c_j, $1 \leq j \leq m$, 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) $unsat \leftarrow \tau(c_j)$, for all $j \in \{1, \ldots, m\}$, where $\tau(c_j)$ is obtained from $\neg c_j$ by replacing each $\neg x_i$ by $not\ \hat{x}_i$, for $i \in \{1, \ldots, n\}$;
(β) $x_i \leftarrow not\ \hat{x}_i$ and $\hat{x}_i \leftarrow not\ x_i$, for all $i \in \{1, \ldots, n\}$; and
(γ) the fact $x_1 \leftarrow$.

Note that in this way, $KB \models unsat$ iff 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 $\mathcal{U}$:

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

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

Let us now 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 iff $unsat \notin Bel(KS_{i-1})$. In particular, $x_n \in Bel(KS)$ iff 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 $\mathcal{U}$. □

Theorem 6 (Stratified KB with stratified $\mathcal{U}$). *The problem* KS-INF *for stratified KB and stratified $\mathcal{U}$ is* P^{NP}*-complete.*

Proof. By Theorem 4(i), it remains to show hardness. The proof is by a reduction similar to that in the proof of Theorem 5. However, KB is stratified; it is obtained from KB there by removing clauses $\hat{x}_i \leftarrow \mathit{not}\, x_i$. The first event in the sequence serves to add exactly these clauses to the knowledge base (we denote it by $E_0 = \{\mathit{add_clauses}\}$). The update policy $\mathcal{U}$ contains for this purpose the additional statements $\mathbf{assert}(\hat{x}_i \leftarrow \mathit{not}\, x_i)$ $[\![\mathbf{E} : \mathit{add_clauses}]\!]$, for each $i \in \{1, \ldots, n\}$. Thus, after the first step of evolution, $KS_1 = \langle KB; E_0 \rangle$ amounts to the same program as KB in the proof of Theorem 5, and P^{NP}-hardness follows for a stratified initial knowledge base with a stratified update policy. □

Theorem 7 (Stratified KB with factual $\mathcal{U}$). *The problem* KS-INF *for stratified KB and factual $\mathcal{U}$ is* P^{NP}*-complete (assuming that Sel is computable in polynomial time with an* NP *oracle).*

Proof. By Theorem 4(i), it remains to show hardness. We give a reduction from the problem in the proof of Theorem 5, finding the last bit (denoted α_n^*) of the lexicographically maximal model of a formula ϕ in CNF. To this end, we construct an update policy $\mathcal{U}$ such that the answer sets of $\Pi_i^{\mathcal{U}}$ correspond one-to-one to the satisfying assignments of ϕ and use the selection function $\mathit{Sel}(\cdot)$ to single out the answer set U_i corresponding to $x_1^*, \ldots, x_n^*$ (which clearly is computable in polynomial time with an NP oracle).

We map each clause $c_j = l_{j,1} \vee \cdots \vee l_{j,k_j}$ in ϕ to the update statement

$$\mathbf{assert}(\mathit{unsat})\ \mathbf{if}\ \mathbf{assert}(\rho(l_{i,1})), \ldots, \mathbf{assert}(\rho(l_{i,k_j})),$$

where $\rho(l) = \hat{x}_i$ if $l = x_i$, and $\rho(l) = x_i$ if $l = \neg x_i$, and establish

$$\begin{array}{l}\mathbf{assert}(x_j)\ \mathbf{if\ not}\ \mathbf{assert}(\hat{x}_j);\\ \mathbf{assert}(\hat{x}_j)\ \mathbf{if\ not}\ \mathbf{assert}(x_j);\\ \mathbf{assert}(f)\ \mathbf{if}\ \mathbf{assert}(\mathit{unsat}),\ \mathbf{not}\ \mathbf{assert}(f).\end{array}$$

Then, for $KS = \langle KB; E_1 \rangle$, where $KB = \emptyset$ and $E_1 = \emptyset$, the answer sets of $\Pi_1^{\mathcal{U}}$ correspond one-to-one to the satisfying assignments of ϕ. For Sel as described, then, $x_n \in \mathit{Bel}(KS)$ iff α_n^* is *true*. □

Thus, Theorems 4–7 imply that only if all restrictions are jointly imposed (KB is stratified, $\mathcal{U}$ is stratified and factual), KS-INF is tractable, but if only one of them is dropped, we face its full complexity. However, it is not much higher than NP, and admits polynomial transformations to special logic programs featuring weak constraints (see [29]).

Let us now consider the case of an unknown selection function. Here we have the following result.

Theorem 8 (General $\mathcal{U}$ and unknown $Sel(\cdot)$). *The problem* KS-INF *under unknown $Sel(\cdot)$ (i.e. deciding whether $r \in Bel(KS)$ holds cautiously for all selection functions $Sel(\cdot)$ is computable in polynomial time with an* NP *oracle) is Π_2^P-complete. Moreover, hardness holds, even if KB is stratified or $\mathcal{U}$ is factual.*

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

Any such sequence describes the results $U_i = Sel(\Pi_i^{\mathcal{U}})$ 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 NP oracle, thus, deciding whether $r \notin Bel(KS)$ for some selection function Sel is in $\mathrm{NP}^{\mathrm{NP}} = \Sigma_2^P$. This proves the membership result.

For the hardness part, let us consider a quantified Boolean formula of form $\forall X_1 \exists X_2 \varphi$, where φ is a CNF on atoms $X = X_1 \cup X_2$. We define KB containing the sets of rules (α) and (β) as defined in the proof of Theorem 5 over X, plus the fact $sat \leftarrow$. We further define $\mathcal{U}$ consisting of the following sets of statements:

> **assert**$(x_{1,j})$ **if not assert**$(\hat{x}_{1,j})$ $[\![\mathbf{E} : guess]\!]$, for all $x_{1,j} \in X_1$;
> **assert**$(\hat{x}_{1,j})$ **if not assert**$(x_{1,j})$ $[\![\mathbf{E} : guess]\!]$, for all $x_{1,j} \in X_1$; and
> **retract**(sat) $[\![unsat, \mathbf{E} : check]\!]$.

Let us consider $KS = \langle KB; E_1, E_2 \rangle$, with $E_1 = \{guess\}$ and $E_2 = \{check\}$. The question then is 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 step of evolution $KS_1 = \langle KB, E_1 \rangle$, 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 iff $\varphi[\sigma(X_1)]$ is unsatisfiable. In this case, sat is removed; otherwise, it remains. Thus, $sat \in Bel(KS)$ under any implementation of Sel iff for every truth assignment $\sigma(X_1)$, formula $\varphi[\sigma(X_1)]$ is satisfiable; in other words, iff $\forall X_1 \exists X_2 \varphi$ is true. This proves the Π_2^P-hardness for a general KB. As in the previous case, we can start from a stratified subset KB and add the remaining clauses by means of a preliminary event $E_0 = \{add_clauses\}$, thus proving that the Π_2^P-hardness also holds for a stratified initial KB. □

Thus, the problem is significantly harder ("double NP-hard") than the previous problems. In particular, by current complexity beliefs, it cannot be solved in polynomial time with a SAT solver; however, it can be polynomially transformed to the disjunctive answer set solver DLV [29].

Imposing both applicable restrictions jointly (KB stratified, $\mathcal{U}$ factual) lowers the complexity to coNP, however, because then the NP oracle in the membership proof of the previous theorem can be replaced by a polynomial-time procedure (see the proof of Theorem 4(ii)).

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 multiagent scenarios are interesting for our purpose. In the first, several cooperative agents share a common knowledge base, that has to be dynamically updated by a *knowledge-management agent*, which receives inputs from the environment and update information from the individual agents. In the second, 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 multiagent system platform [38].

5.1 IMPACT

IMPACT [38], the *Interactive Maryland Platform for Agents Collaborating Together*, is an agent framework that allows existing code and data sources to be "agentized." The behavior of an IMPACT agent, i.e. which actions it executes upon a state change, is specified declaratively by a set of rules. The possibility of employing 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 their *content* 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 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. 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 invokable by external programs via *code calls*.

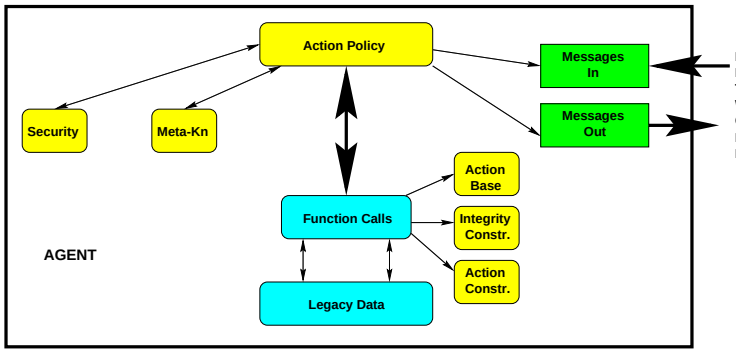

Fig. 2. IMPACT Agent Architecture

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

The agent can execute a set of *actions*, for example, 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 terms of so-called *status sets*. In particular, the semantics of an agent is defined by means of *feasible status sets*. A feasible status set is a set of ground action status atoms, satisfying various conditions. For example, it is closed under the rules of the agent program and complies with certain deontic axioms. Additionally, semantical notions stronger 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 [38].

5.2 Update Agents

The idea is to integrate, on the one hand, the features of the declarative IMPACT action language, and, on the other hand, the declarative EPI formalism for updating knowledge bases in a suitable way, leading to a rule-based language for specifying update behavior for 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 frame-

work without extensions (see Sect. 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 built over the following agent data structures and actions:

1. Suppose 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 $bel()$ that returns the current belief set $Bel(KS)$. Moreover, assume that an event is a finite set of rules and that $\mathcal{SP}$ (or another package) has a function $event()$ that lists all rules in the current event. Then, code call atoms, i.e. atoms of the form $in(l, bel())$, $not_in(l, bel())$, or $in(l, event())$, where l is either a specific rule r or a variable R, can be used to express conditions on the belief set and the event.[2]
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 $ignore(R)$, for a rule identifier R, are required.
3. An update policy $\mathcal{U}$ is represented as a set of IMPACT action rules

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

where $cmd_i(\overline{t}_i)$, $1 \leq i \leq n$, are action atoms, as introduced above, and $CC(\texttt{cond})$ is a list of code call atoms expressing conditions on the belief set and the event in `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 $CC(\texttt{cond})$. For example, 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, unless it is explicitly specified that they are to be ignored.

Example 6. Consider the shopping agent from Example 2. Its update policy may be defined as an IMPACT agent program as follows. Assume that it contains the incorporate-by-default rule from above, as well as,

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

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

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, $\mathcal{U}$ also includes the following rules:

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

Intuitively, an update agent follows its update policy by calculating a new (reasonable) status set on each event that it receives and by executing exactly those (ground) actions $cmd(\overline{t})$ such that the action status atom $\mathbf{Do}\, cmd(\overline{t})$ 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 having the agent perform the corresponding actions, the knowledge base would be updated accordingly.

Obviously, not all actions are compatible with each other, for example, the actions $assert(r)$ and $retract(r)$ should not be executed simultaneously. This can be handled naturally in IMPACT by formulating *action constraints* that 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 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. We note that such straightforward extensions do not require new semantic definitions; they just fit in the usual ones, providing that the evaluation of the condition part in each statement is suitably adapted.

6.1 Rule Patterns

The general form of an update statement in the extended language is as in the basic framework, but rules can be specified by *patterns*, including the use of wild cards and variables to stand for parts of rules: conditions of form $\rho := \mathbf{H}(\rho') \leftarrow A, \mathbf{B}^+(\rho'')$ (so-called *extended conditions*) are allowed, where ρ, ρ', and ρ'' are rule variables, A is a variable for a literal, and $\mathbf{H}(\rho')$ and $\mathbf{B}^+(\rho'')$ refer to the head and positive body of rules ρ' and ρ'', respectively (see also Table 2). This possibility can be very useful when expressing, for example,

retraction of a whole set of rules satisfying some condition. In the context of information agents, assuming, for example, that a certain information source s_1 is and will no longer be available (signaled by the event $out(s_1) \leftarrow$), then the following update statements can serve to retract all rules of the knowledge base that 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 for the last two rules 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 $\mathbf{in}$ and $\mathbf{in_a}$ are reserved symbols) can be evaluated over the current knowledge base (in the first part of the "$[\![\ldots]\!]$" condition). Intuitively, these atoms record whether 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 $on(n_r)$, determining whether 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 define $Bel(KS) \models \mathbf{in_a}(r)$ for some rule r iff $Bel(KS) \models on(n_r)$.

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

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

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 that rules are allowed in events, which may specify the classification of new types of messages, i.e. rules with head $type(M, t_{\text{new}})$, 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:

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

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 the 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, in any case, affects only the evaluation part resulting in the executable update commands. In particular, the notion

Table 2. Extended EPI syntax

$\langle stat \rangle$	::=	$\langle c_name \rangle(\langle r_info \rangle)$ [**if** $\langle if_cond \rangle$] [[[$\langle wh_cond \rangle$]]]
$\langle c_name \rangle$	::=	**assert_[event]** | **retract_[event]** | **always[_event]** | **cancel** | **ignore**
$\langle r_info \rangle$	::=	$\langle rule \rangle$ | $\langle r_id \rangle$ | $\langle r_name \rangle$:=$\langle r_str \rangle$
$\langle r_id \rangle$	::=	$\langle r_name \rangle$ | $\langle r_var \rangle$
$\langle if_cond \rangle$	::=	$\langle cond1 \rangle$ | $\langle cond1 \rangle$, $\langle if_cond \rangle$
$\langle wh_cond \rangle$	::=	$\langle kb_conds \rangle$ | **E** : $\langle ev_conds \rangle$ | $\langle kb_conds \rangle$, **E** : $\langle ev_conds \rangle$
$\langle cond1 \rangle$	::=	[**not**] $\langle c_name \rangle(\langle r_info \rangle)$
$\langle lit_id \rangle$	::=	$\langle literal \rangle$ | $\langle lit_var \rangle$
$\langle r_str \rangle$	::=	$\langle head \rangle \leftarrow \langle body \rangle$
$\langle body \rangle$	::=	$\langle b_cond \rangle$,$\langle body \rangle$ | ϵ
$\langle head \rangle$	::=	$\langle lit_id \rangle$ | # | **H**($\langle r_id \rangle$) | ϵ
$\langle b_cond \rangle$	::=	# | $[not]\langle lit_id \rangle$ | **B**($\langle r_id \rangle$) | $\mathbf{B}^+(\langle r_id \rangle)$ | $\mathbf{B}^-(\langle r_id \rangle)$ | **H**($\langle r_id \rangle$)
$\langle kb_conds \rangle$	::=	$\langle kb_cond \rangle$ | $\langle kb_cond \rangle$, $\langle kb_conds \rangle$
$\langle kb_cond \rangle$	::=	$\langle r_info \rangle$ | $[not]$ **in**$_{[\mathbf{a}]}(\langle r_info \rangle)$ | $[not]\langle lit_id \rangle$
$\langle ev_conds \rangle$	::=	$\langle ev_cond \rangle$ | $\langle ev_cond \rangle$, $\langle ev_conds \rangle$
$\langle ev_cond \rangle$	::=	$\langle lit_id \rangle$ | $\langle r_info \rangle$

of $\mathcal{G}(\mathcal{U})^i$, the set of reduced update statements at Step i, has to be redefined for the new syntax. It is now possible for conditions c_i or r_j ($1 \leq i \leq l$ and $1 \leq j \leq m$) 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 end, let us consider a relation $\subset_{\mathcal{U}}$ between rule variables such that $R_1 \subset_{\mathcal{U}} R_2$ iff 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

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

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

1. For each statement s and 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 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 $L_1, \ldots, L_k = \#$, the rule head, respectively, body need 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 s with the rule name, $n_{r'}$, and then delete r_i from s.

3. For each statement s and 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 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 $L_1, \ldots, L_k = \#$, the rule head, respectively, body need not 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 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 the form

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

   where each $\sigma_{h,1}, \ldots, \sigma_{h,n}$ is either a rule, a rule name, or an extended condition of the form $\rho := L \leftarrow L_1, \ldots, L_k$.
5. For each statement s_h, $1 \leq h \leq p$, and 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 rule head, respectively, body need not be considered), then delete s_h. Otherwise, if such a statement $s_{h'}$ exists, substitute for each occurrence of the variable ρ in s_h the name of the rule in $\sigma_{h',0}$.

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

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

where each σ, σ_i $(1 \leq i \leq n)$ is a rule or a rule name, which represents $\mathcal{G}(\mathcal{U})^i$ 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 (see [6]). This can be realized in essentially two ways, one requiring an extension of the language and one including adhoc statements in the update policy, with special atoms (reserved predicates) that serve to record the past. The two methods are briefly described in the following. Similar mechanisms can be thought of 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 the form "**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 ith step of the evolution. In the first form (more generic), conditions like **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 **Past** : (j, r), for some j, $0 \leq j < i$, and some rule r, which is satisfied iff $Bel(KS_j) \models r$. These two alternatives are suitable for different modeling tasks: the first for checking whether a rule has been in the belief set at any state and the second for checking a particular step in the knowledge evolution.

Adhoc 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 depends on what kind of past conditions are of interest in the problem in question:

1. **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 the form of conditions checking, at Step i, the content of the belief set of Step $i-1$).
2. **retract**$(past(n_r) \leftarrow)\ [\![past(n_r)]\!]$ and **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, i.e. executing Step i, the content of the belief set at Step $i-1$ (as usual) and at Step $i-2$ can be queried.
3. **assert**$(past(1, n_r) \leftarrow)\ [\![on(n_r)]\!]$, together with the retraction command **retract**$(past(1, n_r) \leftarrow)\ [\![past(1, n_r), not\ on(n_r)]\!]$ and, for $1 \leq j \leq k-1$ (k predefined), the commands:

   assert$(past(j+1, n_r) \leftarrow)\ [\![past(j, n_r)]\!]$;
   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 whether $Bel(KS_{i-1-l}) \models on(n_r)$ holds).

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

assert$(system_date_error)\ [\![$**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 the form

$$\texttt{cmd}\ \mathit{rule}\ \texttt{when}\ \mathit{cond}, \tag{3}$$

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

From this point of view, the following differences between the LUPS and the EPI semantics have to be considered and managed by the metainterpreter (for a description of the LUPS syntax and semantics, see [3]):

1. In LUPS, `always` and `always event` commands have a slightly different semantics than the corresponding commands in EPI. In LUPS, the condition *cond* of a command `always` [`event`] *rule* `when` *cond* is evaluated at *every* state transition until the command is canceled, causing an assertion of *rule* only in the follow-up states where *cond* holds. In contrast, in EPI, *rule* is asserted *unconditionally* once *cond* is fulfilled until the command is canceled. Thus, in LUPS, these commands are like macros similar to a cancelable **assert_[event]** command in EPI.
2. The commands `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`] command corresponds thus in EPI to a **retract[_event]** plus a **cancel** for the same rule.
3. Inconsistencies may occur in a sequence of LUPS programs due to the combined effects of commands in successive updates that 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 3 can be illustrated as follows: 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, using a negative control atom for it ($\neg on(n_R)$ in our notation), as long as rule R is not supposed to persist by inertia. But if LU_i contains a new command `assert` R, this should be stronger than a deactivation simulating noninertial 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 $\mathcal{U}_{\mathrm{LUPS}}$.

Let us suppose that LUPS statements of the form (3) are slightly rewritten to rules `cmd` $(rule) \leftarrow cond$. The update policy $\mathcal{U}_{\mathrm{LUPS}}$ 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 $\alpha \in \{\texttt{assert[_event]}, \texttt{retract[_event]}\}$. They 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. Statements implementing LUPS commands `always[event]` and `cancel`:

   $$\begin{array}{l}\mathbf{assert}(ALW_\alpha(n_{rule}))[\![\mathbf{E} : \alpha(rule) \leftarrow cond]\!];\\ \beta(rule)\ \mathbf{if\ assert}(ALW_\alpha(n_{rule}))\ [\![cond]\!];\\ \mathbf{assert}(ALW_\alpha(n_{rule}))\ \mathbf{if\ not\ retract}(ALW_\alpha(n_{rule}))\ [\![ALW_\alpha(n_{rule})]\!];\\ \mathbf{retract}(ALW_\alpha(n_{rule}))[\![cond, \mathbf{E} : \texttt{cancel}(rule) \leftarrow cond\ ]\!],\end{array}$$

   where $ALW_\alpha(\cdot)$ is a reserved predicate, $\alpha \in \{\texttt{always}, \texttt{always event}\}$, $\beta = \mathbf{assert}$ if $\alpha = \texttt{always}$, and $\beta = \mathbf{assert_event}$ if $\alpha = \texttt{always event}$. These statements implement the LUPS semantics for `always [event]` commands by means of EPI **assert_[event]** commands (second statement) and a special atom which is continuously added (first and third statement) until the command is canceled (fourth statement).
3. Additional statements to express the effect of `retract[event]` commands over persistent commands:

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

   where $\alpha \in \{\texttt{always}, \texttt{always event}\}$. These statements simply express the additional effect of LUPS `retract` commands, providing that the corresponding condition is satisfied.
4. Statements raising a conflict by successive updates, where $AE(\cdot)$ and $RE(\cdot)$ are reserved predicates:

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

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

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

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

   $$\begin{array}{l}\mathbf{assert_event}(RE(n_{rule}))[\![cond, \mathbf{E} : \texttt{retract event}(rule) \leftarrow cond]\!];\\ \mathbf{assert}(rule)[\![cond, RE(n_{rule}), \mathbf{E} : \alpha(rule) \leftarrow cond]\!],\end{array}$$

where α = `retract` [`event`]. As long 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 that 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.

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

In [27], 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 noninertial commands is modified, to obtain, as the authors argue, somewhat more intuitive behavior in certain cases (for example, for empty updates) and to ensure that the language is *stable* with respect to noninertial commands. For this purpose, special atoms are introduced that signal at which step a noninertial command holds by computing the interaction between inertial and noninertial commands. We do not address this issue here. We only note that EPI has, in this respect, 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 us 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, by introducing the necessary changes in tr. In any case, it does not further affect the semantics and properties and, since 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 network management, the domain-independent policy description language $\mathcal{PDL}$ has been introduced in [32] and further elaborated and studied in [12,13,37]. $\mathcal{PDL}$ 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 the form 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, respectively, that the event e_2 is generated, provided that condition c holds. Such policies are given a declarative

semantics which associates a history of sets of actions to be taken with a history of sets of primitive events (called *epochs*). Roughly speaking, epochs in $\mathcal{PDL}$ play the role of events in EPI, and actions correspond to update commands.

Despite similarities, $\mathcal{PDL}$ and EPI actually differ in several respects. First of all, $\mathcal{PDL}$ is much more action-oriented than EPI and accepts general user-defined actions. In contrast, EPI employs a predefined set of actions for modifying the knowledge base. The generality of $\mathcal{PDL}$ requires conflict management between actions; for this purpose, an additional control layer for conflict detection and removal, called *policy monitor* [13], has been introduced. The EPI framework *implicitly* performs, as pointed out earlier, a (restricted) form of conflict avoidance by means of 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 can refer to the execution and nonexecution of other actions as a precondition 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 nonreflective reactive behavior. In summary, the expressiveness of the two languages is different; however, neither is strictly more expressive than the other.

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

7.3 Access-Control Policies

In [38,10], the problem of how to derive a main access-control policy by combining partial policies in different ways is addressed. The single policies could, for example, be generated by different authorities regulating access to (areas of) the Intranet of an institution. An algebraic approach is proposed to resolve conflicts and some inconsistencies, which allows the use of several algebra-like operators combining the single policies in different ways; a central authority which decides how the policies must be composed is obviously presupposed.

Though the chosen approach is modeled on a particular application domain and is not directly suitable for extension to areas such as knowledge-base update, it is in general an interesting question whether 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 simplest reasoning tasks.

7.4 Metarules in Active Databases

In [24], the use of metarules is discussed for solving potential conflicts between active rules. The idea is to introduce a set of declarative metarules that together control the interaction and the execution order of active rules.

Metarules 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; and (iv) *scheduling criteria*, i.e. control information about the relative order in which rules are to be executed.

The framework also comprises some static reasoning tasks, namely, determining whether a rule will ever execute, whether two rules can be executed together, or 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, for example, by positive and negative requirements. The treatment of preference and scheduling information would require careful extension because explicit priority handling is not available in EPI.

8 Conclusion

In this chapter, 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. To define concrete update policies, we introduced a declarative specification language, EPI, which extends previous work by several additional features which, in turn, are important for modeling dynamic agents. Among these features, the language allows flexible handling of external events, making it possible for update actions to depend on the content of the incoming information. The relevance of the proposed language for realizing adaptive reasoning agents is further strengthened by the fact that EPI allows direct implementation within the agent platform IMPACT [38].

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 [41]. Other properties may involve temporal relationships such as if *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 [17], where the complexity of its associated reasoning problem has also been analyzed.

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

Acknowledgements. The authors would like to thank the anonymous referees for their constructive comments that helped to improve this chapter. This work was partially supported by the Austrian Science Fund (FWF) under grants P13871-INF and Z29-INF.

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 Kaufmann, San Francisco, 1998.
2. J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic Updates of Non-Monotonic Knowledge Bases. *Journal of 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, Berlin Heidelberg, 1999.
4. J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS – A Language for Updating Logic Programs. *Artificial Intelligence*, 138(1–2):87–116, 2002.
5. K. R. Apt, H. A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, (ed.), *Foundations of Deductive Databases and Logic Programming*, pp. 89–148. Morgan Kaufmann, San Francisco, 1988.
6. C. Baral, J. Llopis, J. Lobo, and G. Mendez. Temporal Logic and Reasoning about Actions. In *Common Sense 96*, 1996.
7. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, in press.
8. L. Bertossi and J. Chomicki. Consistent Query Answers in Inconsistent Databases. Chap. 2 of this volume. Springer-Verlag, Berlin Heidelberg, 2003.
9. 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, ACM Press, New York, 2000.
10. P. Bonatti, S. De Capitani di Vimercati, and P. Samarati. An Algebra for Composing Access Control Policies. *ACM Transactions on Information and System Security*, 5(1):1–35, 2002.

11. F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. *Theory and Practice of Logic Programming*, 2(3): 293–321, 2002.
12. 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, Berlin Heidelberg, 2001.
13. 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, San Francisco, 2000.
14. 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, Berlin Heidelberg, 2000.
15. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. A Framework for Declarative Update Specifications in Logic Programs. In *Proc. 17th Int. Joint Conf. on Artificial Intelligence* (*IJCAI'01*), pp. 649–654. Morgan Kaufmann, San Francisco, 2001.
16. 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*), pp. 79–84. The Society for the Study of AI and the Simulation of Behaviour (AISB), York, 2001.
17. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Reasoning about Evolving Nonmonotonic Knowledge Bases. In *Proc. LPAR 2001*, LNCS 2250, pp. 407–421. Springer, Berlin Heidelberg, 2001.
18. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On Properties of Update Sequences Based on Causal Rejection. *Theory and Practice of Logic Programming*, 2(6):711–767, 2002.
19. T. Eiter and G. Gottlob. On the Complexity of Propositional Knowledge Base Revision, Updates, and Counterfactuals. *Artificial Intelligence*, 57(2–3):227–270, 1992.
20. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, 1997.
21. 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, New York, 1983.
22. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. ICSLP'88*, pp. 1070–1080. MIT Press, Cambridge, MA, 1988.
23. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3–4):365–386, 1991.
24. H. Jagadish, A. Mendelzon, and I. Mumick. Managing Rule Conflicts in Active Databases. *Journal of Computer and System Sciences*, 58(1):13–28, 1999.
25. A.C. Kakas and P. Mancarella. Database Updates Through Abduction. In *Proc. 16th VLDB Conference*, pp. 650–661. Morgan Kaufmann, San Francisco, 1990.
26. H. Katsuno and A. Mendelzon. On the Difference Between Updating a Knowledge Base and Revising it. In *Proc. KR'91*, pp. 387–394. Morgan Kaufmann, San Francisco, 1991.
27. J. Leite. A Modified Semantics for LUPS. In *Proc. 10th Portuguese Conf. on Artificial Intelligence* (*EPIA 2001*), LNCS 2258, pp. 261–275. Springer, Berlin Heidelberg, 2001.
28. J. Leite and L. Pereira. Generalizing Updates: From Models to Programs. In *Logic Programming and Knowledge Representation, Third Int. Workshop* (*LPKR'97*), LNCS/LNAI 1471, pp. 224–246. Springer, Berlin Heidelberg, 1998.

29. N. Leone et al. The DLV System for Knowledge Representation and Reasoning. Technical Report INFSYS RR-1843-02-14, Inst. f. Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, 2002.
30. V. Lifschitz and H. Turner. Splitting a Logic Program. In *Proc. ICLP'94*, pp. 23–38. MIT Press, Cambridge, MA, 1994.
31. V. Lifschitz. Foundations of Logic Programming. In G. Brewka (ed.), *Principles of Knowledge Representation*, pp. 69–127. CSLI Publications, Stanford, 1996.
32. J. Lobo, R. Bhatia, and S. Naqvi. A Policy Description Language. In *Proc. AAAI/IAAI'99*, pp. 291–298. MIT Press, Cambridge, MA, 1999.
33. V. Marek and M. Truszczyński. Revision Programming. *Theoretical Computer Science*, 190(2):241–277, 1998.
34. C. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.
35. G. Sabbatini. *Declarative Logic Programming for Representing Evolving Nonmonotonic Knowledge Bases*. Ph.D. Thesis, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, 2002.
36. C. Sakama and K. Inoue. Updating Extended Logic Programs Through Abduction. In *Proc. LPNMR'99*, LNCS/LNAI 1730, pp. 147–161. Springer, Berlin Heidelberg, 1999.
37. T. Son and J. Lobo. Reasoning about Policies using Logic Programs. In *Proc. AAAI 2001 Spring Symp. on Answer Set Programming: Towards Efficient and Scalable Knowledge Representation and Reasoning*. Technical Report SS-01-01, pp. 210–216. AAAI Press, Menlo Park, CA, 2001.
38. V.S. Subrahmanian, J. Dix, T. Eiter, P. Bonatti, S. Kraus, F. Ozcan, and R. Ross. *Heterogeneous Agent Systems*. MIT Press, Cambridge, MA, 2000.
39. J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, San Francisco, 1996.
40. M. Winslett. *Updating Logical Databases*. Cambridge University Press, Cambridge, 1990.
41. M. Wooldridge. The Computational Complexity of Agent Design Problems. In *Proc. 4th Int'l Conf. on Multi-Agent Systems* (*ICMAS 2000*). IEEE Press, Los Alamitos, CA, 2000.
42. Y. Zhang and N. Foo. Updating Logic Programs. In *Proc. ECAI'98*, pp. 403–407. Wiley, New York, 1998.