Declarative Knowledge Updates Through Agents

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

Abstract

Cooperative, intelligent agents often share a common knowledge which, in the presence of learning agents, is likely to be subject to change. In this paper, we describe a method to construct specialized agents that incorporate new information into a given body of knowledge. These *update agents* are based on established methods from logic programming and follow a declarative update policy in order to perform their tasks. The proposed update agents not only adhere to a clear semantics, but are also able to deal with incomplete or inconsistent information in an appropriate way. We outline a concrete realization of update agents in terms of the *IMPACT* agent system and briefly discuss possible applications.

1 Introduction

Recent years witnessed a growing interest in the development of intelligent software agents for diverse application domains, like e-commerce, entertainment, knowledge management and many more. Such agents provide a wide range of services, including data mediation agents, mobile agents, personalized visualization agents, monitoring agents, mail-filtering agents and the like (see, e.g., (Decker et al., 1997; Flores-Mendez, 1999; Bargmeyer et al., 1999; Levy and Weld, 2000; Sadri and Toni, 2000)).

The environment in which intelligent agents operate is usually nondeterministic and in many cases only partially accessible. Intelligent agents must, to some extent, be able to adapt to changes in the environment and, at the same time, cooperate with other agents in order to achieve their goals. Besides communicational purposes, cooperating agents often share common knowledge. This knowledge may be static but, in the presence of learning agents, it is likely to be subject to change.

Two possible multi-agent scenarios are relevant for our purposes. In the first one, several cooperative agents share a common knowledge base, which has to be dynamically updated by a *knowledge management agent*. The latter agent receives inputs from the environment and update information from the single agents. In the second scenario, each agent in the system has to maintain its own knowledge base, depending on external updates and on internal specifications.

For our purposes, we assume some agent communication language, together with a suitable interaction protocol, and we are interested in maintaining a *dynamic knowledge state*. In particular, we propose to use declarative methods to realize agents whose task is the incorporation of new information into a given body of knowledge. Moreover, we describe how such update agents can be implemented in an existing agent architecture, namely in the multi-agent framework *IMPACT* (Subrahmanian et al., 2000).

As regards knowledge base updates, we utilize declarative techniques which have recently been developed in the area of nonmonotonic knowledge bases (Alferes et al., 2000; Eiter et al., 2000) (cf. also (Foo and Zhang, 1998; Inoue and Sakama, 1999) for related approaches). The reason for using declarative methods is that they provide a clear semantics for knowledge base updates, which allow one to reason about updates. Furthermore, they are also capable of handling incomplete and inconsistent information, modeling some form of nonmonotonic reasoning.

The basic layout of our agent architecture is as follows. The knowledge base consists of facts and rules, represented in the form of a logic program. It represents a partial description of the world in which the agent operates in, and which may be shared with other agents. The agent itself receives new information, also in the form of facts or logic programming rules. Besides an underlying update framework, which specifies under a particular semantics how new, possibly inconsistent information is to be incorporated, the agent possesses also a certain *update policy*, allowing additional flexibility for incorporating specific knowledge. For instance, the policy may specify the change or retraction of certain rules from the knowledge base, given some particular information. More precisely, given a new piece of information, the update mechanism addresses the following questions:

1. Which facts and rules should be incorporated?
2. How should facts and rules be incorporated?

In contrast to *ad-hoc* solutions for these problems, we suggest an approach which tackles these issues by resort-

ing to well-understood declarative methods based on formal principles and designed to handle incomplete and inconsistent information.

2 *IMPACT* Agents

IMPACT, the *Interactive Maryland Platform for Agents Collaborating Together* (Arisha et al., 1999), is an agent framework which allows for existing legacy code and data sources to be "agentized". Moreover, the behavior of an *IMPACT* agent, i.e., which actions it takes upon a state change, is specified declaratively by a set of rules. The possibility to employ existing implementations, together with the possibility of declarative agent specifications, makes the realization of update agents as *IMPACT* agents straightforward.

Figure 1 shows the overall architecture of an *IMPACT* agent. All *IMPACT* agents have the same architecture, and hence the same components, but the *contents* of these components can be different, leading to different behaviors and capabilities.

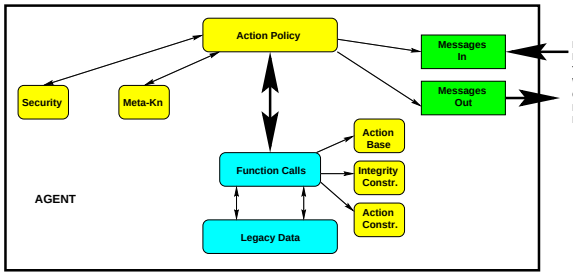

Figure 1: IMPACT Agent Architecture

Basically, the behavior of the agent is driven by an *action policy*. Each agent possesses a message box, which contains incoming and outgoing messages. On the basis of the content of the message box and of queries to legacy data (performed by means of function calls which abstract both from the structure of the underlying 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.

Agent Data Structures. Agents are built "on top" of some existing body of code. Thus, to every agent, a set of types can be assigned, which contains all data types or data structures that the agent manipulates. As usual, each data type has an associated *domain* which is the space of objects of that type. The set of data structures is manipulated by a set of functions that are callable by external programs via *code calls*. Such functions constitute the *application programmer interface* (*API*) of the package on top of which the agent is built. An agent includes a specification of all signatures of these API function calls (i.e., types of the inputs to such function calls and types of the output of such function calls).

Every code call $\mathcal{S} : \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are *terms*, i.e., either values or variables, is based on a body of software code, $\mathcal{S}$ (a so-called *software package*). Such a code call says "execute function $\mathtt{f}$ as defined in package $\mathcal{S}$ on the list of arguments". A code call can be evaluated providing it is ground, i.e., all arguments $\mathtt{t}_i$ must be values. Its output is a set of objects.

Code call atoms are expressions of the form $\mathtt{in}(\mathtt{t}, \mathtt{cc})$ or $\mathtt{notin}(\mathtt{t}, \mathtt{cc})$, where $\mathtt{t}$ is a term and $\mathtt{cc}$ is a code call. A ground term $\mathtt{t}$ succeeds (i.e., has answer *true*) if $\mathtt{t}$ is in the set of values returned by $\mathtt{cc}$, otherwise it fails (i.e., has answer *false*). If $\mathtt{t}$ is a variable, then a code call atom returns each value from the result of $\mathtt{cc}$, i.e., its answer is the set of ground substitutions for $\mathtt{t}$ such that the code call atom succeeds.

A *code call condition* is a conjunction of code call atoms and *constraint atoms*, which may involve deconstruction operations. An example of a constraint atom is $\mathtt{X} > 25$, where $\mathtt{X}$ is a variable. A code call condition checks whether the stated condition is true. In general, constraint atoms are of the form $\mathtt{t}_1 \circ \mathtt{t}_2$ where $\circ \in \{=, \neq, <, \leq, >, \geq\}$ and $\mathtt{t}_1, \mathtt{t}_2$ are terms.

Each agent has access to a message box data structure, together with some API function calls to access it.

At any given point in time, the actual set of objects in the data structures (and the message box) managed by the agent constitutes the *state* of the agent. The set of ground code calls which are true in it are identified as the state, $\mathcal{O}$, of the agent.

Actions. The agent has 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 is an action. Expressions $\alpha(\overline{t})$, where α is an action and $\overline{t}$ is a list of terms, are *action atoms*. They represent the sets of (ground) actions which result if all variables in $\overline{t}$ are instantiated by values. Only such actions may be executed by an agent. Every action has a precondition, a set of effects that describe how the agent state changes when the action is executed, and an *execution script* or *method* consisting of a body of physical code that implements the action.

Agent Programs. Each agent has a set of rules (*action rules*) called the *agent program* specifying 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. Expressions $\mathbf{O}\alpha(\overline{t})$, $\mathbf{P}\alpha(\overline{t})$, $\mathbf{F}\alpha(\overline{t})$, $\mathbf{Do}\,\alpha(\overline{t})$, and $\mathbf{W}\alpha(\overline{t})$, where $\alpha(\overline{t})$ is an action atom, are called *action status atoms*. These action status atoms are respectively read as $\alpha(\overline{t})$ is *obligatory*, $\alpha(\overline{t})$ is *permitted*, $\alpha(\overline{t})$ is *forbidden*, *do* $\alpha(\overline{t})$, and the obligation to do $\alpha(\overline{t})$ is *waived*.

If A is an action status atom, then A and $\neg A$ are called *action status literals*. An *agent program* $\mathcal{P}$ is a finite set of rules of the form

$$A \quad \leftarrow \quad \chi \,\&\, L_1 \,\&\, \cdots \,\&\, L_n,$$

where A is an action status atom, χ is a code call condition, and $L_1, \ldots, L_n$ are action status literals.

Each agent program has a *formal semantics* which is defined in terms of semantical structures called *status sets*, i.e., sets of ground action status atoms. More specifically, the semantics of an agent is defined with respect to *feasible status sets*, which satisfy various conditions. For instance, feasible status sets are required to be closed under the rules of the agent program and comply to certain deontic axioms. Additionally, stronger semantical notions than feasible status sets have been introduced for *IMPACT* agents, namely *rational status sets* and *reasonable status sets*.

There are also further components of *IMPACT* agents which are not relevant for our purposes here. A detailed description of the semantics can be found in (Subrahmanian et al., 2000; Eiter et al., 1999).

3 Knowledge Bases

Update agents operate on knowledge bases, which we assume to be represented by *extended logic programs* (*ELPs* for short) (Gelfond and Lifschitz, 1991), i.e., finite sets of facts and rules. Facts are represented by *literals*, i.e., atomic formulae A or negations $\neg A$ of atomic formulae. We only deal here with propositional formulas; facts and rules involving variables may be represented by their respective ground instances. A rule, r, is an expression of the form

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

where each L_i, $0 \leq i \leq n$, is a literal. We call L_0 the *head* of r (symbolically $H(r)$) and the set

$$B(r) = \{L_1, \ldots, L_m, \mathit{not}\, L_{m+1}, \ldots, \mathit{not}\, L_n\}$$

the *body* of r. The set $\{L_1, \ldots, L_m\}$ will also be denoted by $B^+(r)$, and $\{L_{m+1}, \ldots, L_n\}$ will be denoted by $B^-(r)$.

Intuitively, rule r means that we can conclude L_0 if (i) $L_1, \ldots, L_m$ are known and (ii) $L_{m+1}, \ldots, L_n$ are *not* known. Furthermore, note the difference between *strong negation* $\neg A$ of an atom A, expressing the fact that A is false, and *weak negation* $\mathit{not}\, A$ of the atom A, which is true *if we cannot assert that* A *is true*, i.e., if either A is false or we do not know whether A is true or false.

A set of literals is *consistent* iff it does not contain a complementary pair A, $\neg A$ of literals. Consistent sets of literals are also referred to as *interpretations*. A literal L is *true* in an interpretation I (symbolically $I \models L$) iff $L \in I$, and *false* otherwise.

Given a rule r, the body $B(r)$ of r is true in I iff (i) each $L \in B^+(r)$ is true in I and (ii) each $L \in B^-(p)$ is false in I. In other words, $B(r)$ is true in I iff $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. We write $I \models B(r)$ to express that $B(r)$ is true in I. Rule r is true in I iff $H(r)$ is true in I whenever $B(r)$ is true in I. The fact that r is true in I will be denoted by $I \models r$. Likewise, for a program P, $I \models P$ means that $I \models r$ for all $r \in P$. In this case, I is said to be a *model* of P.

Since rules may include weak negation, they are more expressive than ordinary Horn clauses. This is the reason why we cannot always assign a unique set of consequences to a knowledge base. Several semantics for extended logic programs exist; in the following, we describe the well-known *answer set semantics* due to Gelfond and Lifschitz (1991).

Let r be a rule. Then, r^+ denotes the rule obtained from r by deleting all weakly negated literals in the body of r, i.e., $r^+ = H(r) \leftarrow B^+(r)$. Furthermore, we say that rule r is *defeated* by a set of literals S if some literal in $B^-(r)$ is true in S, i.e., if $B^-(r) \cap S \neq \emptyset$.

The *reduct*, P^S, of a program P *relative to* a set S of literals is defined by

$$P^S = \{r^+ \mid r \in P \text{ and } r \text{ is not defeated by } S\}.$$

In other words, P^S is obtained from P by (i) deleting any $r \in P$ which is defeated by S and (ii) deleting each weakly negated literal occurring in the bodies of the remaining rules.

An interpretation I is an *answer set* of a program P iff it is a minimal model of P^I (the reduct P^I is often called the *Gelfond-Lifschitz reduction*). Observe that any answer set of P is *a fortiori* a model of P.

Example 1 *Consider the knowledge base KB consisting of the following rules:*

$$KB \quad = \quad \big\{r_1 : \mathit{sleep} \leftarrow \mathit{not}\, \mathit{tv\_on},\ r_2 : \mathit{night} \leftarrow,\ r_3 : \mathit{tv\_on} \leftarrow,\ r_4 : \mathit{watch\_tv} \leftarrow \mathit{tv\_on}\big\}.$$

Given the interpretation $S = \{night, tv\_on, watch\_tv\}$, the reduct KB^S consists of the rules r_2, r_3, and r_4. It is easily verified that S is a minimal model of KB^S. Hence, S is an answer set of KB.

A program may possess one or several answer sets, or none at all. Therefore, we associate with every knowledge base KB a specific set of beliefs, $Bel(KB)$. Taking a cautious view, a rule r (or a fact L) is in $Bel(KB)$ iff every answer set of KB is a model for it.

If changes occur in the environment, or if new information is acquired which has to be incorporated, then the knowledge base must be updated. This is represented in terms of sequences $(P_1, \ldots, P_n)$ of logic programs, where each P_i $(1 < i \leq n)$ is assumed to update the information given by $P_1, \ldots, P_{i-1}$.

Several update formalisms for logic programs have been suggested in the literature (see, e.g., (Foo and Zhang,

1998; Inoue and Sakama, 1999; Alferes et al., 2000; Eiter et al., 2000)). For all these formalisms, the update policy is fixed and implicitly encoded in the semantics of the update process itself. In order to resolve conflicting information, the update semantics assigns more recent information precedence over old one. However, the information learnt cannot be "filtered" or "pre-processed", and is dealt with in the same uniform way.

Example 2 *Assume the knowledge base KB of Example 1 is updated with the following information:*

$$U_1 \quad = \quad \{r_5 : \neg tv\_on \leftarrow power\_failure, \\ r_6 : power\_failure \leftarrow \}.$$

Simply adding the new rules to KB would result in an inconsistent knowledge base, since it would be able to derive the facts 'tv_on' and '$power\_failure$', and, due to rule r_5, '$\neg tv\_on$'. Update formalisms avoid such situations by employing more subtle update mechanisms. For example, applying the update semantics proposed by Eiter et al. (2000), rule r_3 would be rejected in favour of the more recent rule r_5. In this case,

$$S_1 = \{power\_failure, \neg tv\_on, night, sleep\}$$

would be an answer set for KB updated by U_1.

The language LUPS, due to Alferes et al. (1999), allows for more flexibility of the update process. It permits to dynamically specify the contents of a sequence of updates by compiling a corresponding sequence of meta-programs into a concrete update sequence. Within the meta-programs, one can express *conditional assertion* or *retraction of rules*, based on the semantics of the current knowledge state. This is achieved by means of the following commands:

- **assert** r: add rule r to the current knowledge state;
- **assert event** r: add rule r only *once* to the current knowledge state, i.e., r is not supposed to persist by inertia;
- **retract** r: remove rule r from the current knowledge state;
- **retract event** r: remove rule r *once* from the current knowledge state, i.e., r persists in ensuing updates;
- **always** r: apply **assert** r in every update;
- **always event** r: apply **assert event** r in every update;
- **cancel** r: cancel rule r in a previous **always** command.

These commands permit the specification of changes to a knowledge base KB in terms of specifying which rules should hold or not hold in the resulting knowledge state (and whether they should hold or not hold by inertia). For instance,

$$\textbf{assert } sleep \leftarrow power\_failure \textbf{ when } night$$

asserts that the rule $sleep \leftarrow power\_failure$ should be added to KB if $night$ is currently true in it.

4 Update Agents

Although LUPS enhances the flexibility of logic program updates, a limitation is that it has no notion of incoming information from the environment, i.e., LUPS possesses no notion of *event*. Therefore, unforeseen updates cannot be modeled. Also, it is not possible for the assertion or retraction of a rule to be conditioned on the execution of other LUPS commands.

In what follows, we introduce update agents which overcome these limitations. These agents are able to handle LUPS commands and, additionally, execute them depending on other commands and events that occur. The basic idea is to combine and integrate the features of the declarative action language of *IMPACT* and the declarative LUPS formalism for updating knowledge bases in a suitable way, leading to a rule-based language for specifying update behavior to a knowledge base. A formal semantics for update policies, given in this language, is inherited from the semantics for agent programs in *IMPACT*, combined with the semantics of the LUPS language as in (Alferes et al., 1999).

More specifically, an update policy consists of an *IMPACT* agent program, which is build over the following agent data structures and actions.

Software Package. We developed a software package $\mathcal{SP}$ for updating and querying a knowledge base KB, which implements the methods described above. The API consists of two code calls:

- `bel()` accesses KB and returns the current belief set $Bel(KS)$, and
- `event()` lists all rules in the current event, which is a finite set of rules.[1]

Code call atoms, i.e., atoms of the form $in(t, bel())$, $not\_in(t, bel())$, and $in(t, event())$, where t is either a specific rule r or a variable R, can be used to express conditions on the belief set and the event. In case t is a variable, it is quantified over all r such that $in(r, bel())$, $not\_in(t, bel())$, and $in(t, event())$) is true, respectively.

Actions. Certain *IMPACT* actions, which can be realized in any suitable programming language (like, e.g., Java or C), model the operations of the LUPS language by means of their effects, which are logically specified in terms of *add lists* and *delete lists*. In particular, the actions

[1] Note that a fact L can be equivalently represented as a rule $L \leftarrow$.

- $always(R)$, $always\_event(R)$,
- $assert(R)$, $assert\_event(R)$,
- $retract(R)$, $retract\_event(R)$, and
- $cancel(R)$,

where R is a (rule) parameter, are required. Further auxiliary actions, such as $ignore(R)$, whose semantical effect is a "no operation," may be defined, facilitating more convenient formulations of the agent policy.

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

Update Policy. An update policy, $\mathcal{U}$, can be represented as a set of *IMPACT* action rules of the form

$$\begin{aligned}\mathbf{Do}\, cmd_1(\bar{t}_1) \leftarrow \quad & [\neg]\mathbf{Do}\, cmd_2(\bar{t}_2), \ldots, \\ & [\neg]\mathbf{Do}\, cmd_n(\bar{t}_n), \texttt{CC\_cond},\end{aligned}$$

where each $cmd_i(\bar{t}_i)$ $(1 \leq i \leq n)$ is an action atom and `CC_cond` is a list of code call atoms expressing conditions on the belief set and the event 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_cond`. E.g., the action rule

$$\mathbf{Do}\, assert(R) \leftarrow \neg\mathbf{Do}\, ignore(R), in(R, event())$$

encodes a simple *incorporate-by-default* policy, in the sense that event rules are incorporated in the knowledge base except if it is explicitly specified that the rule has to be ignored.

Example 3 *Consider a simple agent selecting Web shops s_i, $1 \leq i \leq n$, in search for some specific merchandise. Suppose its knowledge base, KB, contains the rules*

$$\begin{array}{lcl} r_i & : & query(s_i) \leftarrow sale(s_i), up(s_i), \\ & & \qquad\qquad not\ \neg query(s_i); \\ r_j & : & try\_query \leftarrow query(s_i); \\ r_{2n+1} & : & notify \quad \leftarrow not\ try\_query \end{array}$$

$(1 \leq i \leq n, n+1 \leq j \leq 2n)$ and a fact $r_0 : date(0)$ as an initial time stamp. Here, $r_1, \ldots, r_n$ express that shop s_i, which has a sale and whose Web site is up, is queried by default, and $r_{n+1}, \ldots, r_{2n+1}$ serve to detect that no site is queried, which causes '$notify$' to be true.

Assume that an event, E, might be any consistent set of facts or ground rules of the form $sale(s_i) \leftarrow date(t)$, stating that shop s_i has a sale on date t, such that E contains at most one time stamp $date(\cdot)$. An update policy $\mathcal{U}$ might be defined as an IMPACT agent program as follows. Assume it contains the above incorporate-by-default rule, as well as:

$$\begin{aligned}&\mathbf{Do}\, always(sale(S) \leftarrow date(T)) \leftarrow \\ &\quad \mathbf{Do}\, assert(sale(S) \leftarrow date(T)), \\ &\quad in(S, shops()), in(T, dates());\end{aligned}$$

$$\begin{aligned}&\mathbf{Do}\, cancel(sale(S) \leftarrow date(T)) \leftarrow \\ &\quad in(date(T), isBel()), T \neq T', \\ &\quad in(date(T'), event()), in(S, shops()), \\ &\quad in(T, dates()), in(T', dates());\end{aligned}$$

$$\begin{aligned}&\mathbf{Do}\, retract(sale(S) \leftarrow date(T)) \leftarrow \\ &\quad in(date(T), isBel()), T \neq T', \\ &\quad in(date(T'), event()), in(S, shops()), \\ &\quad in(T, dates()), in(T', dates());\end{aligned}$$

Informally, the first rule repeatedly confirms the information about a future sale, which guarantees that it is effective on the given date, while the second rule revokes this. The third one removes information about a previously ended sale (assuming the time stamps increase). Note that we assume the existence of two code calls, given by '$shops()$' and '$dates()$', which we use for grounding, i.e., they return all possible assignments for shops and time stamps. Furthermore, $\mathcal{U}$ includes also the following rules:

$$\begin{aligned}&\mathbf{Do}\, retract(date(T)) \leftarrow in(date(T), isBel()), \\ &\quad T \neq T', in(date(T'), event()), \\ &\quad in(T, dates()), in(T', dates());\end{aligned}$$

$$\mathbf{Do}\, ignore(sale(s_1)) \leftarrow in(sale(s_1), event());$$

$$\begin{aligned}&\mathbf{Do}\, ignore(sale(s_1) \leftarrow date(T)) \leftarrow \\ &\quad in(sale(s_1) \leftarrow date(T), event()), in(T, dates());\end{aligned}$$

The first rule keeps the time stamp '$date(t)$' in KB unique, and removes the old value. The other statements simply express that sales information about shop s_1 is ignored.

Intuitively, an update agent follows its update policy by calculating a new (reasonable) status set on each event it receives, and by executing exactly those (ground) actions $cmd(\bar{t})$ such that the action status atom $\mathbf{Do}\, cmd(\bar{t})$ is in the status set.

For instance, suppose the update agent specified in Example 3 is triggered by the event $\{sale(s_2), date(1)\}$. Then, its reasonable status set includes the action status atoms $assert(sale(s_2))$, $assert(date(1))$, as well as $retract(date(0))$. By performing the corresponding actions, the knowledge base is updated by the agent accordingly.

Using Action Modalities. A direct benefit of using *IMPACT* programs is the availability of deontic modalities. By allowing arbitrary modalities in the action rules of an update policy, its expressiveness can be further enhanced, i.e., one admits rules of the form

$$\begin{aligned}\mathbf{Op}_1\, cmd_1(\bar{t}_1) \leftarrow \quad & [\neg]\mathbf{Op}_2\, cmd_2(\bar{t}_2), \ldots, \\ & [\neg]\mathbf{Op}_n\, cmd_n(\bar{t}_n), \texttt{CC\_cond},\end{aligned}$$

where each $\mathbf{Op}_i \in \{\mathbf{O}, \mathbf{P}, \mathbf{F}, \mathbf{Do}, \mathbf{W}\}$ is a modality. This makes it possible to state under which conditions the

assertion or retraction of a rule is possible, allowed, forbidden, etc. For example,

$$\mathbf{Do}\, cmd_1(\bar{t}_1) \leftarrow \neg \mathbf{F} cmd_1(\bar{t}_1), \texttt{CC\_cond}$$

expresses that $cmd_1(\bar{t}_1)$ is executed unless it is explicitly forbidden.

In the above mentioned framework where only **Do** modalities are allowed, the modality **F** can be used to simulate the $ignore$ action. More specifically, every occurrence of $\mathbf{Do}\, ignore(R)$ is replaced by $\mathbf{F} assert(R)$, and rules $\mathbf{F} cmd(R) \leftarrow \mathbf{F} assert(R)$ are added which force $\mathbf{F} cmd(R)$ for any $cmd \in \{always, always\_event, cancel, assert\_event, retract, retract\_event\}$.

5 Conclusion

We introduced update agents for the purpose of updating knowledge bases, represented as logic programs. The agents follow a declarative policy, and they can easily be implemented within the *IMPACT* agent framework, which implicitly assigns a formal semantics to the update policies of the agents.

In a companion paper (Eiter et al., 2001), the meta-language EPI is defined, specifying update behavior over nonmonotonic knowledge bases. EPI is a generalization of LUPS, and its formal semantics is based on declarative logic programming. Furthermore, properties of the EPI language are investigated. EPI can be mapped to *IMPACT* update agents, as described in the present paper, which provide under reasonable status set semantics an implementation of this meta-language.

An implementation of update agents as outlined in the previous section is part of ongoing work. It utilizes an implementation of the update semantics due to Eiter et al. (2000), which has been integrated within the *IMPACT* environment.

The generic nature of the underlying update mechanism, as well as the independently specified update policy, allows a straightforward change of either module. Moreover, the well-defined semantics, designed to deal with incomplete and inconsistent information, admits the analysis of formal properties of the framework. These features allow update agents to be a valuable part of (an infrastructure for) learning agents. They can not only serve as knowledge management agents for cooperative agents sharing a common knowledge base, but they can also be part of adaptive agents which learn from their interaction with the environment and which update their knowledge accordingly. With such a scenario in mind, the integration of learning components with update agents is an interesting subject for further research.

Acknowledgments

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

References

J. Alferes, J. Leite, L. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic Updates of Non-Monotonic Knowledge Bases. *J. of Logic Programming*, 45(1–3): 43–70, 2000.

J. Alferes, L. Pereira, H. Przymusinska, and T. Przymusinski. LUPS—A Language for Updating Logic Programs. In *Proc. LPNMR'99*, volume 1730 of *LNAI*, pages 162–176. Springer, 1999.

K. Arisha, T. Eiter, S. Kraus, F. Ozcan, R. Ross, and V.S. Subrahmanian. IMPACT: A Platform for Collaborating Agents. *IEEE Intelligent Systems*, 14(2):64–72, 1999.

B. Bargmeyer, J. Fowler, M. Nodine, and B. Perry. Agent-Based Semantic Interoperability in InfoSleuth. *SIGMOD Record*, 28(1):60–67, 1999.

K. Decker, K. Sycara, and M. Williamson. Middle-Agents for the Internet. In *Proc. IJCAI'97*, volume 1, pages 578–583. Morgan Kaufmann, 1997.

T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Considerations on Updates of Logic Programs. In *Proc. JELIA 2000*, volume 1919 of *LNAI*. Springer, 2000.

T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. Specifying Update Policies for Nonmonotonic Knowledge Bases. In *Proc. DGNMR 2001*, 2001.

T. Eiter, G. Pick, and V.S. Subrahmanian. Heterogeneous Active Agents, I: Semantics. *Artificial Intelligence*, 108 (1–2):179–255, 1999.

R. Flores-Mendez. Towards a Standardization of Multi-Agent System Frameworks. *ACM Crossroads*, 5(4), 1999.

N. Foo and Y. Zhang. Updating Logic Programs. In *Proc. ECAI'98*, pages 403–407. Wiley, 1998.

M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9(3–4):365–386, 1991.

K. Inoue and C. Sakama. Updating Extended Logic Programs through Abduction. In *Proc. LPNMR'99*, volume 1730 of *LNAI*, pages 147–161. Springer, 1999.

A. Levy and D. Weld. Intelligent Internet Systems. *Artificial Intelligence*, 118(1–2):1–14, 2000.

F. Sadri and F. Toni. Computational Logic and Multi-Agent Systems: a Roadmap. *Computational Logic, Special Issue on the Future Technological Roadmap of Compulog-Net*, 2000.

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