An Update Front-End for Extended Logic Programs

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

Institut für Informationssysteme, Abt. Wissensbasierte Systeme 184/3,
Technische Universität Wien, Favoritenstrasse 9-11, A-1040 Vienna, Austria
{eiter,michael,giuliana,tompits}@kr.tuwien.ac.at

Abstract. In recent years, several approaches for dealing with updates of logic programs have been proposed. In this paper, we describe the system `upd`, an implementation of the update formalism due to Eiter *et al.* This method is based on a compilation technique to standard answer set semantics, in which update sequences are translated into single logic programs, and which allows the use of existing logic programming systems as underlying reasoning engine. In the present case, `upd` is conceived as a front-end to the state-of-the-art solver `DLV`. Besides the basic update semantics of Eiter *et al.*, the implementation handles also refinements of the semantics involving certain minimality-of-change criteria.

1 Background

The problem of updating nonmonotonic knowledge bases has gained increasing interest in recent years. In particular, several update approaches have been proposed in which knowledge bases are represented as logic programs [1–3, 6, 7]. In this paper, we present the system `upd`, which is an implementation of the method for updating logic programs due to Eiter *et al.* [2, 3].This approach is based on the answer set semantics for extended logic programs, and, like related update formalisms, it incorporates new information into the current knowledge base according to a *causal rejection principle*. This principle enforces that, in case of conflicts between rules, more recent rules have precedence over older rules. The general approach can be described as follows.

Given a sequence $(P_1, \ldots, P_n)$ of extended logic programs, each P_i is assumed to update the information expressed by the initial sequence $(P_1, \ldots, P_{i-1})$. The sequence $(P_1, \ldots, P_n)$ is then translated into a single logic program P', respecting the successive update information, such that the answer sets of P' represent the "update answer sets" of $(P_1, \ldots, P_n)$. The translation is realized by introducing new atoms which control the applicability of rules with respect to the given update information. Informally, if two rules, $r \in P_i$ and $r' \in P_j$, assert conflicting information, where $i < j$, then the more recent rule, r', is applied, whilst r is "rejected". From a technical point of view, this rejection principle is expressed in terms of the so-called *rejection set*, $rej(S, \boldsymbol{P})$, which consists of all rules of the given update sequence $\boldsymbol{P} = (P_1, \ldots, P_n)$ which are rejected on the basis of an update answer set S.

A property which this basic update semantics intuitively does not respect is *minimality of change*. In general, however, it is desirable to incorporate a new set of rules into an existing program with as little change as possible. This is realized by the notions of *minimal* and *strictly minimal update answer sets*, as introduced in [2, 3]. Intuitively, an update answer set S is minimal iff there is no update answer set S' of

Algorithm Compute_Minimal_Models(***P***)

Input: A sequence of ELPs $\boldsymbol{P} = (P_1,\ldots,P_n)$.
Output: All minimal answer sets of ***P***.

var $S : AnswerSet$;
var $MinModels : Set\_Of\_AnswerSets$;
$S := Next\_Answer\_Set(\boldsymbol{P}_{\lhd})$;
while $S \neq$ nil **do**
    var $Counter : Set\_Of\_AnswerSets$;
    $Counter := Compute\_Answer\_Sets(\boldsymbol{P}_S^{min})$;
    if $(Counter = \emptyset)$ **then**
        $MinModels := MinModels \cup \{S\}$;
    fi
    $S := Next\_Answer\_Set(\boldsymbol{P}_{\lhd})$;
od
return $MinModels$;

Fig. 1. Algorithm to calculate minimal update answer sets.

the update sequence $\boldsymbol{P} = (P_1, \ldots, P_n)$ yielding a smaller rejection set, i.e., such that $rej(S', \boldsymbol{P}) \subset rej(S, \boldsymbol{P})$ holds. Strict minimality is a somewhat stronger notion taking also the rules rejected at specific levels into account. More specifically, let $rej_i(S, \boldsymbol{P})$ be the rejected rules contained in P_i $(1 \leq i \leq n)$, then the update answer set S is strictly minimal iff there is no update answer set S' of the update sequence ***P*** such that $rej_i(S', \boldsymbol{P}) \subset rej_i(S, \boldsymbol{P})$ and $rej_j(S', \boldsymbol{P}) = rej_j(S, \boldsymbol{P})$ for $i + 1 \leq j \leq n$.

Generally speaking, the implementation `upd` handles the following reasoning tasks: (i) checking the existence of an update answer set for a given update sequence, (ii) brave reasoning, and (iii) skeptical reasoning. Each of these tasks is realized for the basic update semantics, as well as for minimal and strictly minimal update answer sets. Furthermore, the tasks are defined for function-free (datalog) programs, utilizing the advanced grounding mechanism of `DLV`.

2 System Specifics

2.1 General Information

Since the above update approach is based on a compilation technique to standard answer set semantics, it is possible to build an implementation using an existing logic programming system as underlying reasoning engine. In the present case, `upd` is realized as a front-end to the logic programming tool `DLV` [4, 5], which is a state-of-the-art solver for disjunctive logic programs under the answer set semantics. Of course, `smodels` [8], a state-of-the-art system for normal logic programs, could also be employed as underlying reasoning engine.

Given a sequence of update programs as input, `upd` first translates this sequence into a single extended logic program, $\boldsymbol{P}_{\lhd}$, and then invokes `DLV` to calculate the answer

sets of $\boldsymbol{P}_{\lhd}$. In order to obtain update answer sets of the given input sequence, the special-purpose atoms introduced by the translation are filtered from the answer sets of $\boldsymbol{P}_{\lhd}$.

For dealing with minimal and strictly minimal update answer sets, upd employs a two-phase evaluation approach. The overall algorithm for calculating minimal update answer sets is depicted in Figure 1. Roughly speaking, the algorithm proceeds as follows: First, the answer sets of the update program $\boldsymbol{P}_{\lhd}$ are calculated. As soon as an answer set S is produced (denoted by $Next\_Answer\_Set(\boldsymbol{P}_{\lhd})$), it is tested for being minimal by calculating the answer sets of a particular test program, $\boldsymbol{P}_S^{min}$, consisting of the rules of $\boldsymbol{P}_{\lhd}$ together with a set of additional rules. S is minimal iff the test program $\boldsymbol{P}_S^{min}$ has no answer set. The algorithm for strictly minimal update answer sets is analogous, the only difference is that the test program $\boldsymbol{P}_S^{min}$ is replaced by a suitable test program $\boldsymbol{P}_S^{strict}$.

2.2 Applying the System

The general syntax of upd coincides with the syntax of DLV. Update sequences are represented by grouping rules using the braces "{" and "}", as illustrated by the following example:

```
{sleep :- night, not tv_on.
 watch_tv :- tv_on.
 night.
 tv_on.}
{-tv_on :- power_failure.
 power_failure.}
```

This input represents an update sequence (P_1, P_2), where the first group of rules constitutes the initial knowledge base, P_1, and the second group corresponds to the update information P_2.

Intuitively, the above example expresses the following situation: The initial program specifies that someone sleeps at night unless the TV is on, in which case the person is watching TV. This knowledge is updated by the information that the TV is not on providing there is a power failure, and there is actually a power failure.

The program upd processes inputs either in the form of files or as immediate input via a command shell. Supposing the above sequence of programs has been saved in a file named tv.lps, the computation of the corresponding update answer sets can be engaged by the command "upd", producing the following output:

```
> upd -p=~/bin/dlv  tv.lps
upd [build BEN/Nov 15 2000   gcc 2.95.2 19991024 (release)]

dlv [build BEN/Jun 11 2001   gcc 2.95.2 19991024 (release)]

{night, power_failure, sleep, -tv_on}
```

Observe that upd requires the explicit specification which particular prover should be invoked during the computation process. This choice is determined by the option -p, which allows for selecting alternate evaluation tools besides DLV.

It is also possible to feed upd with multiple inputs. For instance, suppose we have another file, say tv_cont.lps, containing the following sequence of programs:

```
{-power_failure.}
{switched_off :- not tv_on, not power_failure.
 tv_on :- not switched_off, not power_failure.
 -tv_on :- switched_off.}
```

If these programs are assumed to update the information given by file `tv.lps`, the update answer sets of the overall sequence (comprised of four programs) can be computed as follows:

```
> upd -silent -p=~/bin/dlv -o=-silent tv.lps tv_cont.lps
{-tv_on, night, switched_off, -power_failure, sleep}
{tv_on, watch_tv, night, -power_failure}
```

Here, options `-silent` and `-o=-silent` have been invoked to suppress any additional `upd` and `DLV` messages, where `-o` allows to pass options to the employed evaluation program (`DLV` in the present case).

Further options of `upd` are `-min` and `-strict`, which specify whether minimal or strictly minimal update answer sets should be computed, respectively. For instance, if we are interested in computing the minimal update answer sets of the sequence given by the files `tv.lps` and `tv_cont.lps`, we may call `upd` as follows:

```
> upd -min -silent -p=~/bin/dlv -o=-silent tv.lps tv_cont.lps
{tv_on, watch_tv, night, -power_failure}
```

Finally, `upd` can be downloaded from the Web at

`http://www.kr.tuwien.ac.at/staff/giuliana/project.html`.

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

References

1. Alferes, J.J., Leite, J.A., Pereira, L.M., Przymusinska, H., & Przymusinski, T. C.: Dynamic Updates of Non-Monotonic Knowledge Bases. *Journal of Logic Programming*, 45(1-2):43-70, 2000.
2. Eiter, T., Fink, M., Sabbatini, G., & Tompits, H.: Considerations on Updates of Logic Programs. In *Proc. JELIA 2000.* Lecture Notes in AI (LNAI), vol. 1919. Springer.
3. Eiter, T., Fink, M., Sabbatini, G., & Tompits, H.: On Updates of Logic Programs: Semantics and Properties. Technical Report INFSYS 1843-00-08, TU Wien, 2000.
4. Eiter, T., Leone, N., Mateis, C., Pfeifer, G., & Scarecello, F.: A Deductive System for Nonmonotonic Reasoning. *Pages 363–374 of: Proc. LPNMR'97*, 1997. Lecture Notes in AI (LNAI), vol. 1265.
5. Eiter, T., Faber, W., Leone, N., & Pfeifer, G.: Declarative Problem-Solving Using the dlv System. *Pages 79–103 of: Logic-Based Artificial Intelligence*, 2000. Kluwer Academic Publishers.
6. N. Foo, N., & Zhang, Y.: Updating Logic Programs. *Pages 403–407 of: Proc. ECAI'98*,1998. John Wiley and Sons.
7. Inoue, K., & Sakama, C.: Updating Extended Logic Programs through Abduction. *Pages 147–161 of: Proc. LPNMR'99*,1999 Lecture Notes in AI (LNAI), vol. 1730. Springer.
8. Niemelä, I., & Simons, P.: Efficient Implementation of the Well-founded and Stable Model Semantics. *Pages 289–303 of: Proc. Joint International Conference and Symposium on Logic Programming*, 1996.