Nonmonotonic Description Logic Programs: Implementation and Experiments ⋆

Thomas Eiter, Giovambattista Ianni, Roman Schindlauer, and Hans Tompits

Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{eiter, ianni, roman, tompits}@kr.tuwien.ac.at

Abstract. The coupling of description logic reasoning systems with other reasoning formalisms (possibly over the Web) is becoming an important research issue and calls for advanced methods and algorithms. Recently, several notions of *description logic programs* have been introduced, combining rule-based semantics with description logics. Among them are *nonmonotonic description logic programs* (or *dl-programs* for short) which combine nonmonotonic logic programs with description logics under a generalized version of the answer-set and the well-founded semantics, respectively, which are the predominant semantics for nonmonotonic logic programs. In this paper, we consider some technical issues regarding an efficient implementation for both semantics, which has been realized in a working prototype exploiting the two state-of-art tools DLV and RACER. A major issue in this respect is efficient interfacing between the two reasoning systems at hand, for which we devised special methods. Such methods may fruitfully be used for the implementation of systems of similar nature. Reported experimentation activities with our prototype show that the methods we have developed are effective and are a key for highly optimized nonmonotonic dl-program engines.

1 Introduction

Description logics are well-known formalisms for describing ontological knowledge, and play an important role for building the *Semantic Web* [3,4,9]. The latter is conceived as a hierarchy of different layers, of which the *Ontology Layer* is currently the highest layer of sufficient maturity, as evidenced by the W3C recommended *Web Ontology Language* (*OWL*) [21,13]. OWL has three increasingly expressive sublanguages, namely *OWL Lite*, *OWL DL*, and *OWL Full*. As shown in [12], the logical underpinnings of the former two is provided by the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, respectively.

The further steps in the development of the Semantic Web are realizing the *Rules*, *Logic*, and *Proof Layers* on top of the Ontology layer, which should offer sophisticated representation and reasoning capabilities. This requests, in particular, the need to integrate the Rules and the Ontology layer.

⋆ This work was partially supported by the Austrian Science Fund under grant P17212-N04, and by the European Commission through the IST REWERSE Network of Excellence (IST-506779) and the IST Working Group in Answer Set Programming (IST 2001-37004 WASP).

Towards this goal, several approaches for combining description logics with rule-based languages have been proposed recently [5,16,17,1,6,7,22]. Among them are *description logic programs*, or *dl-programs* for short, presented in [6,7] as a novel method to couple description logics with nonmonotonic logic programs. Roughly speaking, a dl-program consists of a knowledge base L in a description logic and a finite set P of generalized logic-program rules, called *dl-rules*. These are similar to usual rules in logic programs with negation as failure, but they may also contain *queries to* L in their bodies. Importantly, such queries also allow for specifying an input from P to L, and thus for a *bidirectional flow of information between* P *and* L. Consequently, dl-programs allow for building rules on top of ontologies, but also, to some extent, building ontologies on top of rules.

By virtue of their design, dl-programs fully support encapsulation and privacy of the description logic knowledge base, in the sense that logic programming and description logic inference are technically separated and only interfacing details need to be known. The description-logic knowledge bases in dl-programs are theories in the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$. However, the framework can be easily extended to other description logics as well.

Two basic types of semantics have been defined for dl-programs: in [6], a generalization of the answer-set semantics [10] for ordinary logic programs is given, and in [7], a generalization of the well-founded semantics [19,2]. In fact, two versions of the answer-set semantics for dl-programs are introduced in [6], namely the *weak answer-set semantics* and the *strong answer-set semantics*. Every strong answer set is also a weak answer set, but not vice versa. The two notions differ in the way they deal with *nonmonotonic dl-queries*. We recall that the answer-set semantics and the well-founded semantics are the two predominant semantics for nonmonotonic logic programs.

In this paper, we consider technical issues regarding an efficient implementation of the answer-set and the well-founded semantics for dl-programs, which has been realized in a working prototype exploiting the two state-of-the-art solvers DLV [15] and RACER [11]. A major issue in this respect is an efficient interfacing between the two reasoning systems at hand, for which we devised special methods.

The main contributions of this paper can be summarized as follows:

– We give novel methods and algorithms for computing answer sets and the well-founded semantics of dl-programs. Starting from a simple guess-and-check algorithm for weak answer sets, we devise more efficient techniques which prune the number of guesses and reduce the effort for dl-query evaluation.
– As a first improvement over the naive guess-and-check method, we discuss the case of *stratified dl-programs*, which, as follows from one of our results, can be evaluated without explicitly using or even knowing some stratification, at the ground or non-ground level, with a standard answer-set solver.
– We devise special optimization techniques in order to avoid redundant computations. To wit, we discuss a method to avoid multiple ground program generation, as well as special methods to reduce the number of calls to the description logic engine. The latter techniques involve, on the one hand, an exploitation of function calls to the description logic reasoner using non-ground queries, and, on the other hand, special caching data structures tailored for fast access to previous query calls. Also, hierarchic structures of the dependency graph can be taken into account for evaluating unstratified dl-programs.

- The well-founded semantics is computed through an iterative procedure in terms of the greatest and the least fixpoint of a monotonic operator. Techniques devised for the answer-set semantics can be fruitfully applied here as well for improving the evaluation. Moreover, the computation of answer sets can be optimized with the help of a prior computation of the well-founded semantics, by introducing suitable constraints.
- We implemented the above algorithms and optimization techniques in a working prototype, both for computing the answer-set semantics as well as the well-founded semantics, and performed experiments on a suite of benchmark problems. Our experimental results show that the optimization techniques can drastically improve the performance of dl-programs over incremental grades of optimization.

Note that most of our methods and results are at an abstract level, and thus may also be exploited for implementing similar computational-logic systems based on coupling.

2 Background

In this section, we recall syntax and semantics of description logic programs, introduced in [6,7]. In what follows, we assume a function-free first-order vocabulary, Φ, with nonempty finite sets of constant and predicate symbols, and a set $\mathcal{X}$ of variables. As usual, a *classical literal* (or *literal*), l, is an atom a or a negated atom $\neg a$.

2.1 $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$

Intuitively, description logics allow for expressing knowledge about concepts, roles, and individuals in a (possibly extended) first-order logic (with concepts and roles being unary and binary predicates $C(a)$ and $R(a,b)$, respectively) using a special syntax. Since for the purpose of this paper we mainly interface description logics through queries, we omit definitions of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ at this point and refer to Appendix A (or, alternatively, to [12,6]) for more details.

A ($\mathcal{SHIF}(\mathbf{D})$ resp. $\mathcal{SHOIN}(\mathbf{D})$) description logic *knowledge base* L is a finite set of axioms in the respective description logic. We denote logical consequence of an axiom α from L, which is defined as usual, by $L \models \alpha$.

2.2 Description logic programs

Informally, a description logic program consists of a description logic knowledge base L and a generalized normal program P which may contain queries to L. Roughly, in such a query, it is asked whether a certain description logic axiom or its negation logically follows from L or not. For details, we refer to [6,7].

Syntax. We first define dl-queries and dl-atoms, which are used to access the description logic knowledge base. A *dl-query*, $Q(\mathbf{t})$, is either (a) a concept inclusion axiom $C \sqsubseteq D$ or its negation $\neg(C \sqsubseteq D)$, or (b) of the form $C(t)$ or $\neg C(t)$, where C is a concept and t is a term, or (c) of the form $R(t_1, t_2)$ or $\neg R(t_1, t_2)$, where R is a role and t_1, t_2 are terms.[1]

[1] Note that $\mathcal{SHOIN}(\mathbf{D})$ does not provide terminological role negation; we use the expression $\neg(\exists R.\{b\})(a)$ in order to add and query $\neg R(a,b)$ for a specific pair of individuals.

A *dl-atom* has the form

$$DL[S_1\, op_1 p_1, \ldots, S_m\, op_m\, p_m; Q](\mathbf{t})\,, \qquad m \geq 0, \tag{1}$$

where each S_i is either a concept or a role, $op_i \in \{\uplus, \cup\!\!\!\!-, \cap\!\!\!\!-\}$, p_i is a unary resp. binary predicate symbol, and $Q(\mathbf{t})$ is a dl-query. We call $p_1, \ldots, p_m$ its *input predicate symbols*. Intuitively, $op_i = \uplus$ (resp., $op_i = \cup\!\!\!\!-$) increases S_i (resp., $\neg S_i$) by the extension of p_i, while $op_i = \cap\!\!\!\!-$ constrains S_i to p_i.

Example 2.1 The dl-atom $DL[\mathit{buying} \uplus \mathit{buy\_cand}, \mathit{buying} \uplus \mathit{contract}; \mathit{Discount}](V)$ queries for all individuals of the concept $\mathit{Discount}$ after adding the extensions of both $\mathit{buy\_cand}$ and $\mathit{contract}$ to the role buying.

A *dl-rule*, r, is an expression of the form,

$$a \leftarrow b_1, \ldots, b_k, \mathit{not}\, b_{k+1}, \ldots, \mathit{not}\, b_m\;,\; m \geq k \geq 0\,, \tag{2}$$

where a is a literal and $b_1, \ldots, b_m$ are either literals or dl-atoms. The symbol "*not*" stands for *weak negation*, also called *negation as failure* (NAF). We refer to a as the *head* of r, denoted $H(r)$, and to the part right of "$\leftarrow$" as the *body* of r. Its *positive part* is $b_1, \ldots, b_k$, and its *negative part* is $\mathit{not}\, b_{k+1}, \ldots, \mathit{not}\, b_m$. Furthermore, we define $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$.

A dl-rule is *ordinary*, if it contains no dl-atom. An ordinary program is a finite set of ordinary rules. A *description logic program*, or *dl-program*, $KB = (L, P)$, consists of a description logic knowledge base L and a finite set of dl-rules P.

Semantics. We first recapitulate the strong and weak answer-set semantics for dl-programs [6], and then the well-founded semantics for dl-programs [7]. They generalize the familiar answer-set semantics [10] and well-founded semantics [19] for ordinary programs, respectively, which are the predominant semantics for nonmonotonic logic programs.

We need some auxiliary notions. In what follows, let $KB = (L, P)$ be a dl-program.

The *Herbrand base* of P, denoted HB_P, is the set of all ground literals with a standard predicate symbol that occurs in P and constant symbols in Φ, where Φ is assumed to contain (a subset of) the constant symbols from L. An *interpretation* I relative to P is a consistent subset of HB_P. Such an I is a *model* of $l \in HB_P$ under L, denoted $I \models_L l$, iff $l \in I$, and a *model* of a ground dl-atom $a = DL[S_1\, op_1\; p_1, \ldots,\; S_m\, op_m p_m; Q](\mathbf{c})$ under L, denoted $I \models_L a$, iff $L \cup \bigcup_{i=1}^{m} A_i(I) \models Q(\mathbf{c})$, where

- $A_i(I) = \{S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$, for $op_i = \uplus$,
- $A_i(I) = \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I\}$, for $op_i = \cup\!\!\!\!-$, and
- $A_i(I) = \{\neg S_i(\mathbf{e}) \mid p_i(\mathbf{e}) \in I$ does not hold$\}$, for $op_i = \cap\!\!\!\!-$.

I is a *model* of a ground dl-rule r iff $I \models_L H(r)$ whenever both $I \models_L l$ for all $l \in B^+(r)$ and $I \not\models_L l$ for all $l \in B^-(r)$. I is a model of a dl-program $KB = (L, P)$, or I *satisfies* KB, denoted $I \models KB$, iff $I \models_L r$ for all r in the grounding, $grd(P)$, of P. We say that KB is *satisfiable* if it has some model, otherwise KB is *unsatisfiable*.

A ground dl-atom a is *monotonic* relative to $KB = (L, P)$, providing $I \models_L a$ implies $I' \models_L a$, for $I \subseteq I' \subseteq HB_P$. A dl-program $KB = (L, P)$ is *positive*, if (i) P is *not*-free and (ii) every ground dl-atom occurring in $grd(P)$ is monotonic relative to KB.

Observe that while dl-atoms containing only $\uplus$ and $\cup\!\!\!\!-$ are always monotonic, a dl-atom containing $\cap\!\!\!\!-$ may fail to be monotonic, since an increasing set of $p_i(\mathbf{e})$ in P results in a reduction of $\neg S_i(\mathbf{e})$ in L.

We are now in the position to define the answer-set semantics for dl-programs. For any dl-program $KB = (L, P)$, we denote by DL_P the set of all ground dl-atoms that occur in $ground(P)$. We assume in the following that KB has an associated set $DL_P^+ \subseteq DL_P$ of ground dl-atoms which are known to be monotonic, and we denote by $DL_P^? = DL_P - DL_P^+$ the set of all other dl-atoms. An *input literal* of $a \in DL_P$ is a ground literal with an input predicate of a and constant symbols in Φ.

Strong answer sets. The *strong dl-transform* of P relative to L and an interpretation $I \subseteq HB_P$, denoted sP_L^I, is the set of all dl-rules obtained from its grounding $grd(P)$ with respect to Φ by (i) deleting every dl-rule r such that either $I \not\models_L a$ for some $a \in B^+(r) \cap DL_P^?$, or $I \models_L l$ for some $l \in B^-(r)$, and (ii) deleting from each remaining dl-rule r all literals in $B^-(r) \cup (B^+(r) \cap DL_P^?)$.

Notice that (L, sP_L^I) is a positive dl-program, which, as shown in [6], has a least model if it is satisfiable. We call $I \subseteq HB_P$ a *strong answer set* of KB iff it is the least model of (L, sP_L^I).

Weak answer sets. Weak answer sets are like strong answer sets if monotonicity of all dl-atoms is unknown resp. ignored (i.e., technically, if $DL_P^? = DL_P$). In the respective *weak dl-transform*, wP_L^I, of P relative to L and $I \subseteq HB_P$, all dl-atoms are removed from $grd(P)$. A *weak answer set* of KB, then, is an interpretation $I \subseteq HB_P$ such that I is the least model of the ordinary positive program wP_L^I. Note that strong answer sets of KB are weak answer sets of KB, but not vice versa in general.

For any *not*-free dl-program P, both the strong reduct as well as the weak reduct coincide with the usual Gelfond-Lifschitz reduct [10], and thus the strong and weak answer sets of $KB = (L, P)$ coincide with the standard answer sets of P.

Well-founded Semantics (*WFS*). The WFS is defined in [7] for dl-programs KB without classical negation and where all dl-atoms are monotonic. The former is no real restriction but the latter a technical necessity. In practice, most dl-atoms are monotonic.

The WFS in [7] generalizes the classical WFS [19] by suitably generalizing the notion of an *unfounded set* as in [19] to the setting of dl-atoms as follows. Let, for any set S of literals, $\neg.S$ be the set of the opposite literals of S. A set $U \subseteq HB_P$ is an *unfounded set* of $KB = (L, P)$ relative to a consistent set I of ground literals, if for every $a \in U$ and every $r \in grd(P)$ with $H(r) = a$, either (i) $\neg b \in I \cup \neg.U$ for some ordinary atom $b \in B^+(r)$, or (ii) $b \in I$ for some ordinary atom $b \in B^-(r)$, or (iii) for some dl-atom $b \in B^+(r)$, $S^+ \not\models_L b$ for every consistent set S of ground literals with $I \cup \neg.U \subseteq S$, or (iv) $I^+ \models_L b$ for some dl-atom $b \in B^-(r)$.

Compared to [19], Conditions (iii) and (iv) are novel. The WFS is then defined in [7] like in [19] as the least fixpoint of a monotonic operator $W_{KB}(I)$ (this is feasible since the greatest unfounded set of I always exists); for computation purposes, an alternative characterization, discussed in Section 3.6, is more advantageous.

If P does not contain any dl-atoms, then the well-founded semantics for $KB = (L, P)$ coincides with the well-founded semantics for P in the sense of [19].

Stratified Semantics. The notion of *stratification* for dl-programs [6] is similar as for ordinary programs. Roughly speaking, stratified dl-programs are composed of hierarchic layers of positive dl-programs that are linked via default negation (for a formal definition, we refer the reader to [6]). As discussed in [7], if $KB = (L, P)$ is positive or stratified, then it has a single strong answer set, which coincides with $WFS(KB) \cap HB_P$.

Example 2.2 A computer shop obtains its hardware from several vendors. It uses a knowledge base L_1 (see Appendix B), which contains information about the product range that is provided by each vendor (property $provides$) and about possible rebate conditions (concept $Discount$, depending on property $buying$; here we assume that buying two or more parts from the same seller causes a discount). To evaluate possible combinations of purchases, the following program P_1 is specified:

(1) $vendor(s_1);\ \ vendor(s_5);\ \ vendor(s_9);$
(2) $needed(cpu);\ \ needed(harddisk);\ \ needed(case);$
(3) $contract(s_9, case);$
(4) $avoid(V) \leftarrow vendor(V), not\ rebate(V);$
(5) $rebate(V) \leftarrow vendor(V), DL[buying \uplus buy\_cand, buying \uplus contract; Discount](V);$
(6) $buy\_cand(V, P) \leftarrow vendor(V), not\ avoid(V), DL[provides](V, P), needed(P),$
$\qquad not\ exclude(P)$
(7) $exclude(P) \leftarrow buy\_cand(V_1, P), buy\_cand(V_2, P), V_1 \neq V_2;$
(8) $exclude(P) \leftarrow contract(V, P), needed(P);$
(9) $supplied(V, P) \leftarrow DL[buying \uplus buy\_cand, buying \uplus contract; buying](V, P),$
$\qquad needed(P).$

Rules (1)–(3) state the considered vendors as well as the needed parts; for some parts, a vendor may already be contracted as supplier. Rules (4)–(6) choose a possible vendor (buy_cand) for each needed part, taking into account that the selection might affect the rebate condition (by feeding the possible vendor back to L_1, where the discount is determined). Rules (7) and (8) assure that each hardware part is bought only once, considering that for some parts a supplier might already be chosen. Rule (9) eventually summarizes all purchasing results. Evaluating this program under the strong answer-set semantics yields the following answer sets (quoting only the relevant atoms):

$\{supplied(s_9, case); supplied(s_5, cpu); supplied(s_5, harddisk); rebate(s_5); \ldots\};$
$\{supplied(s_9, case); supplied(s_9, harddisk); rebate(s_9); \ldots\};$
$\{supplied(s_9, case); \ldots\}.$

For more details, discussion, and examples, see [6,7].

3 Implementing dl-Programs

In this section, we consider methods for computing dl-programs by using an answer-set solver on the one hand and a description logic (DL) engine on the other. We start with a simple method, and then present progressively methods to increase the efficiency.

3.1 Naive computation of weak answer sets

The computation of the weak answer sets of a given dl-program $KB = (L, P)$ can be encoded by ordinary logic programs under the answer-set semantics, following a generate and test approach, as follows:

1. Let P_d be the ordinary logic program having each dl-atom $a(\mathbf{t})$ occurring in P replaced by the atom $d_a(\mathbf{t})$ (we call this kind of atoms *replacement atoms*), where d_a is a fresh predicate symbol.
2. Add to P_d from Step 1 for each replacement atom $d_a(\mathbf{t})$ all rules
$$d_a(\mathbf{c}) \leftarrow \mathit{not}\, \neg d_a(\mathbf{c}) \quad \text{and} \quad \neg d_a(\mathbf{c}) \leftarrow \mathit{not}\, d_a(\mathbf{c}) \tag{3}$$
such that $a(\mathbf{c})$ is a ground instance of dl-atom $a(\mathbf{t})$. Intuitively, the rules (3) "guess" the truth values of the dl-atoms of P.[2] Denote the resulting program by P_{guess}.
3. Compute the answer sets $Ans = \{M_1, \ldots, M_n\}$ of P_{guess}.
4. For each answer set $M \in Ans$ of P_{guess}, test whether the original "guess" of the value of $d_a(\mathbf{c})$ is compliant with L. That is, for each dl-atom a of form (1), check whether $d_a(\mathbf{c}) \in M$ iff $M \models_L a$, i.e., $L \cup \bigcup_{i=1}^{m} A_i(M) \models Q(\mathbf{c})$. If this condition holds (and only if), then $M \cap HB_P$ is a weak answer set of P.

If only one answer set is desired, the algorithm may stop after the first one is found.

While simple and elegant, this method becomes quickly infeasible. If the number of ground dl-atoms grows, the number of candidate answer sets generated may become very large, and Ans may occupy a lot of space. It is more efficient to interleave Steps 3 and 4 and to test each candidate answer set M_i immediately upon its generation. Still, a lot of effort may be spent for evaluating dl-atoms. Efficient implementations try to prune the number of guesses, and to reduce the effort for dl-atom evaluation.

3.2 Stratified dl-programs

In case of a stratified dl-program $KB = (L, P)$, the guessing of the outcome of dl-atoms can be avoided entirely. In the presence of monotonic dl-atoms only, a simple method for computing the (unique) strong answer set of KB is given by a fixpoint iteration of the operator $\Lambda_{KB} : 2^{HB_P} \to 2^{HB_P}$, defined by $\Lambda_{KB}(I) = M(P_d \cup D_P(I)) \cap HB_P$, where:

- P_d is as in Step 1 of the naive computation above;
- $D_P(I)$ is the set of all facts $d_a(\mathbf{c}) \leftarrow$ such that $I \models_L a(\mathbf{c})$; and
- $M(P_d \cup D_P(I))$ is the single answer set of $P_d \cup D_P(I)$; since P_d is stratified, this answer set is guaranteed to exist and to be unique.

For the sequence of powers $I_{KB}^0 = \emptyset$, $I_{KB}^{i+1} = \Lambda_{KB}^{i+1}(\emptyset) = \Lambda_{KB}(I_{KB}^i)$, $i \geq 0$, we then have:

Lemma 3.1. *For each stratified KB, the sequence I_{KB}^i, $i \geq 0$, converges, and its limit I_{KB}^∞ coincides with the strong answer set of KB.*

[2] Note that, when using the system DLV, rules (3) can equivalently be replaced by the disjunctive facts $d_a(\mathbf{c}) \vee \neg d_a(\mathbf{c}) \leftarrow$.

Notice that Λ_{KB} is neither monotonic nor anti-monotonic, and that the sequence I^i_{KB}, $i \geq 0$, is not a chain. The proof of convergence is along a stratification.

In view of this lemma, we can evaluate a stratified dl-program very easily *without explicitly using or even knowing some stratification, at the ground or non-ground level*, with a standard answer-set solver (which is used to compute $M(P_d \cup D_P(I))$ and multiple calls to a DL reasoner (for deciding $I \models_L a(\mathbf{c})$ when it is needed to add facts $d_a(\mathbf{c})$ to $D_P(I)$), in a simple loop.

In fact, the above method is applicable beyond stratified dl-programs. Let us call a program P *dl-stratified*, if in the usual dependency graph G of $grd(P_d \cup DLI(P))$, where $DLI(P)$ consists of all rules $d_a(\mathbf{X}) \leftarrow p_i(\mathbf{Y})$, $i \in \{1, \ldots, m\}$, for each dl-atom a of form (1) occurring in P, no replacement atom $d_a(\mathbf{c})$ is reachable from a cycle having negative arcs.

The class of dl-stratified dl-programs is still rich in the sense that it features non-determinism for problem solving, where ontologies can be accessed in portions of the program that computes information in a stratified layer (possibly through positive recursion, e.g., by taking transitive closure).

Let us call an answer-set solver *deterministic*, if it returns for any input program P on each call always the same result; i.e., if multiple answer sets exist, a "canonical" answer set $can(P)$ will be output. Then the following holds:

Proposition 3.1. *Given a deterministic answer-set solver, for each dl-stratified KB, the sequence I^i_{KB}, $i \geq 0$, (where $can(P_d(I))$ replaces $M(P_d(I))$ in Λ_{KB}) converges, and its limit I^∞_{KB} is a strong answer set of KB.*

Since in dl-stratified programs terminological knowledge is involved only in a stratified portion of the program, it can be dealt with a quick preprocessing (cf. Section 3.5) by easy means. We thus can solve a very relevant class of unstratified dl-programs through the above technique.

Intuitively, assuming the given program has a stratification $\lambda = \{KB_0, \ldots, KB_n\}$, a disadvantage of this simple method is the effort spent for evaluating dl-atoms in higher levels KB_i of the stratification in the early stages of the fixpoint iteration, where the input from lower levels has not converged yet. This effort can be saved by proceeding along λ and computing $\Lambda_{KB_0}, \ldots, \Lambda_{KB_n}$, for the associated strata $KB_0, \ldots KB_n$, at the cost of pre-computing λ. This may pay off in general, given that the entailment to dl-atoms is costly.

Furthermore, it turns out that both the answer-set solver and the DL engine are invoked repeatedly, so that it is very important to avoid redundant computations. Thus, two more additional optimization techniques are fruitful, namely ground program reusing, and dl-atom caching and intelligent evaluation, discussed next.

3.3 Avoiding multiple ground program generation

The above method relies on the evaluation of a collection of ordinary logic programs $P_d \cup D(I^j_{KB})$ starting from $I^0_{KB} = \emptyset$. The programs of this sequence are very similar: indeed, for any I^j_{KB} and $I^{j'}_{KB}$, the programs $P_d \cup D(I^j_{KB})$ and $P_d \cup D(I^{j'}_{KB})$ differ only in the set of facts $d_a(\mathbf{c})$ such that $I^j_{KB} \models_L a(\mathbf{c})$ is different from $I^{j'}_{KB} \models_L a(\mathbf{c})$, and so, their ground versions are very similar.

Indeed, it is possible to compute and store $grd(P_d)$ only once, and then, for any interpretation I, compute $M(grd(P_d) \cup D(I))$ whenever necessary.

This method can be enhanced by considering that answer-set programming systems, like DLV, allow to obtain significantly smaller versions of ground programs, where only meaningful rules are kept; in particular, such grounding systems compute only those ground rules which can be grounded not with respect to the whole Herbrand base HB_P but with respect to a notion of "active" domain of the rules (for more technical details, see [8]). Let $ogrd(P)$ denote the optimized ground version of a program P. For space reasons, we cannot describe this operator in detail, but we observe that, in general, for a given I, $ogrd(P_d \cup D(I)) \neq ogrd(P_d) \cup ogrd(D(I))$, whereas for the usual grounding of $P_d \cup D(I)$ with respect to Φ it holds that $grd(P_d \cup D(I)) = grd(P_d) \cup grd(D(I))$.

This latter property prevents, in principle, to have any benefit in computing and storing $ogrd(P_d)$ instead of $grd(P_d)$. Nonetheless, we can prove that there exists an optimized version $ogrd^*(P_d)$ of $grd(P_d)$, such that it holds that $M(ogrd^*(P_d) \cup D(I)) = M(grd(P_d) \cup D(I))$, for each I, and $ogrd^*(P_d) \subseteq grd(P_d)$. Details of this optimization technique are somehow intricate, so we give only an intuition on how it is carried out for not-free programs.

Given a rule $r \in P_d$, we consider a replacement atom $d_a(\mathbf{t})$ *safe*, if each variable $X \in \mathbf{t}$ appears at least once in some non-replacement atom in the body of r. The program $ogrd^*(P_d)$ is obtained as follows:

1. Build a program P'_d from P_d by removing from every rule $r \in P_d$ each replacement atom $d_a(\mathbf{t})$. In case this atom is not safe, we add in the body of r a predicate $dom(X)$ for each variable $X \in \mathbf{t}$ witnessing unsafety. Furthermore, we add to P'_d a rule r' with head $d_a(\mathbf{t})$ and body consisting of the ordinary body atoms of r, plus an atom $dom(X)$ for each variable $X \in \mathbf{t}$.
2. Add to P'_d a fact $dom(a) \leftarrow$ for each $a \in HB_P$.
3. Let $\bar{D}$ be the set $\{d_a(\mathbf{c}) \leftarrow \ | d_a(\mathbf{c}) \in M(P'_d)\}$.[3]
4. Define $ogrd^*(P_d) = ogrd(P_d \cup \bar{D}) - \bar{D}$.

Intuitively, in Step 1, we create an envelope for the least model of $P_d \cup D(I)$ on the original predicates, which then allows to limit the set of ground dl-atoms $a(\mathbf{c})$, potentially relevant for evaluating P, to those such that $d_a(\mathbf{c})$ is true in the least model of P'_d. For programs with not, we can proceed similarly discarding in Step 1 all not literals.

3.4 Efficient dl-atom evaluation and caching

Since the calls to the DL reasoner are a bottleneck in the coupling of an ASP solver with a DL engine, special methods need to be devised in order to save on the number of calls to the DL engine. To this end, we use complementary techniques.

DL-function calls. One of the features of DL reasoners which may be fruitfully exploited for speed up are non-ground queries. RACER provides the possibility to retrieve in a function call all instances of a concept C (resp., of a role R) that are provable in

[3] In order to prevent DLV from optimized unfolding cancelling out significant rules, some other specialized rules are added.

the DL knowledge base. Given that the cost for accessing the DL reasoner is high, in the case when several different ground instances $a(\mathbf{c}_1), a(\mathbf{c}_2), \ldots, a(\mathbf{c}_k)$ of the dl-atom $a(\mathbf{t})$ have be evaluated, it is a reasonable strategy to retrieve at once, using the apposite function call feature from the DL reasoner, all instances of the concept C (resp., a role R) in $a(\mathbf{t}) = DL[S_1 op_1 p_1, \ldots; C](\mathbf{t})$. This allows to avoid issuing k separate calls for the single ground atoms $a(\mathbf{c}_1)$, ..., $a(\mathbf{c}_k)$.

If the retrieval set has presumably many more than k elements, we can filter it with respect to $\mathbf{c}_1, \ldots, \mathbf{c}_k$, by pushing these instances to a DL engine as follows. For the query concept C, we add in L axioms to the effect that $C'' = C \sqcap C'$, where C' and C'' are fresh concept names, and axioms $C'(\mathbf{c}_1)$, ..., $C'(\mathbf{c}_k)$; then we ask for all instances of C''. For roles, a similar yet more involved approximation method is introduced, given that $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$ do not offer role intersection.

With the above techniques, the number of calls to the DL reasoner can be greatly reduced. Another very useful technique to achieve this goal is caching.

DL-Caching. Whatever semantics is considered, a number of calls will be made to the DL engine. Therefore, it is is very important to avoid an unnecessary flow of data between the two engines, and to save time when a redundant DL query has to be made. In order to achieve these objectives, it is important to introduce some special caching data structures tailored for fast access to previous query calls. Such a caching system needs to deal with the case of Boolean as well as non-Boolean DL-calls.

For any dl-atom $DL[\lambda; Q](\mathbf{t})$, where λ is a list $S_1 op_1 p_1, \ldots, S_n op_n p_n$, and interpretation I, let us denote by I^λ the projection of I on $p_1, \ldots, p_n$.

Boolean DL-calls. In this case, an external call must be issued in order to verify whether a given ground dl-atom b fulfills $I \models_L b$, where I is the current interpretation and L is the DL knowledge base hosted by the DL engine. In this setting, the caching system exploits properties of monotonic dl-atoms $a = DL[\lambda; Q](\mathbf{c})$.

Given two interpretations I_1 and I_2 such that $I_1 \subseteq I_2$, monotonicity of a implies that (i) if $I_1 \models_L a$ then $I_2 \models_L a$, and (ii) if $I_2 \not\models_L a$ then $I_1 \not\models_L a$. This property allows to set up a caching machinery where only the outcome for ground dl-atoms with minimal/maximal input is stored.

Roughly speaking, for each monotonic ground dl-atom a we store a set $cache(a)$ of pairs $\langle I^\lambda, o\rangle$, where $o \in \{true, undefined\}$. If $\langle I^\lambda, true\rangle \in cache(a)$, then we can conclude that $J \models_L a$ for each J such that $I^\lambda \subseteq J^\lambda$. Dually, if $\langle I^\lambda, undefined\rangle \in cache(a)$, we can conclude that $J \not\models_L a$ for each J such that $I^\lambda \supseteq J^\lambda$.

We sketch the maintenance strategy for $cache(a)$ in the following. The rationale is to cache minimal (resp., maximal) input sets I^λ for which a is evaluated to $true$ (resp., $undefined$) in past external calls.

Suppose a ground dl-atom $a = DL[\lambda; Q](\mathbf{c})$, an interpretation I, and a cache set $cache(a)$ are given. With a small abuse of notation, let $I(a)$ be a function whose value is $true$ iff $I \models_L a$ and $undefined$ otherwise. In order to check whether $I \models_L a$, $cache(a)$ is consulted and updated as follows:

1. Check whether $cache(a)$ contains some $\langle J, o\rangle$ such that $J \subseteq I^\lambda$ if $o = true$, or $J \supseteq I^\lambda$ if $o = undefined$. If such J exists, conclude that $I(a) = o$.
2. If no such J exists, then decide $I \models_L a$ through the external DL engine. If $I \models_L a$, then add $\langle I^\lambda, true\rangle$ to $cache(a)$, and remove from it each pair $\langle J, true\rangle$ such that

$I^\lambda \subset J$. Otherwise (i.e., if $I \not\models_L a$) add $\langle I^\lambda, \mathit{undefined}\rangle$ to $\mathit{cache}(a)$ and remove from it each pair $\langle J, \mathit{undefined}\rangle$ such that $I^\lambda \supset J$.

Some other implementational issues are worth mentioning. First of all, since the subsumption test between sets of atoms is a critical task, some optimization is made in order to improve cache look-up. For instance, an element count is stored for each atom set, in order to prove early that $I \nsubseteq J$ whenever $|I| > |J|$. More intelligent strategies could be envisaged in this respect. Furthermore, a standard *least recently used* (LRU) algorithm has been introduced in order to keep a fixed cache size.

Non-Boolean DL-calls. In most cases, a single non-ground query for retrieving all instances of a concept or role might be employed. Caching of such queries is also possible, but cache look-up cannot take advantage of monotonicity as in the Boolean case. For each non-ground dl-atom $a = DL[\lambda; Q](\mathbf{c})$, a set $\mathit{cache}(a)$ of pairs $\langle I^\lambda, a{\downarrow}(I^\lambda)\rangle$ is maintained, where $a{\downarrow}(I)$ is the set of all ground instances a' of a such that $I \models_L a'$. Whenever for some interpretation I, $a{\downarrow}(I)$ is needed, then $\mathit{cache}(a)$ is looked up for some pair $\langle J, a{\downarrow}(J)\rangle$ such that $I^\lambda = J$.

3.5 Unstratified dl-programs

When looking at the corresponding dependency graph, it often occurs in practice that answer-set programs are structured in three separate and hierarchic layers:

- a first, stratified layer at the bottom which performs some preprocessing on the input data;
- a second, strongly connected and unstratified layer, usually aimed at encoding some nondeterministic choice, and, eventually,
- a third "checking" layer on top, where values computed through the other layers are filtered with respect to some constraint criteria.

Following this common setting, we conceived an evaluation strategy where each component is evaluated sequentially and results are fed from one layer to another. This way, the bottom layer is computed exploiting techniques from Subsection 3.2. General techniques are strictly limited to situations in which this cannot be avoided, as in non-stratified layers.

3.6 Implementing the well-founded semantics

An implementation of WFS for KB by fixpoint iteration of the defining monotonic operator $W_{KB}(I)$ as in [7] is not attractive, since a polynomial-time algorithm for computing the greatest unfounded set of KB with respect to I, due to Condition (iii) of an unfounded set, is not evident (even if deciding $I \models_L l$ is polynomial).

As shown in [7], the WFS for KB, denoted $\mathit{WFS}(KB)$, is alternatively given by

$$\mathit{WFS}(KB) = \mathit{lfp}(\gamma_{KB}^2) \cup \{\neg a \mid a \in HB_P - \mathit{gfp}(\gamma_{KB}^2)\},$$

where the operator $\gamma_{KB}(I)$ assigns each interpretation $I \subseteq HB_P$ the least model M_{KB^I} of the strong reduct $KB^I = (L, sP_L^I)$. Since γ_{KB} is anti-monotonic, γ_{KB}^2 is monotonic and thus has a least and greatest fixpoint, $\mathit{lfp}(\gamma_{KB}^2)$ and $\mathit{gfp}(\gamma_{KB}^2)$, respectively.

This way, $WFS(KB)$ is computable through a fixpoint iteration which computes and outputs the greatest and the least fixpoint of the γ^2_{KB} operator, starting from $\emptyset$ resp. HB_P (which may be represented by its complement). Since KB^I is a positive dl-program, machinery developed in Section 3.2 for computing M_{KB^I} is very helpful in this respect. Caching also proves to be very fruitful.

3.7 Enhancing answer-set generation with well-founded semantics

Another interesting result from [7] allows to speed up the computation of the answer sets of a given $KB = (P, L)$ by means of a pre-evaluation of $WFS(KB)$:

Theorem 3.1 *Every strong answer set of a dl-program KB* $= (L, P)$ *includes lfp*(γ^2_{KB}) *and no atom* $a \in HB_P - gfp(\gamma^2_{KB})$.

For computing answer sets, we can exploit the possibility to introduce *constraints* to a DLV program [15]. Constraints allow to filter out models which do not fulfill prescribed requirements. An intermediate ordinary program P' obtained from P can be then enriched with the constraint $\leftarrow\ not\, a$ for any atom a such that $a \in WFS(KB)$, and with a constraint $\leftarrow a$ for any atom a such that $\neg a \in WFS(KB)$. Notice that such constraints may also be added only for a subset of $WFS(KB)$ (e.g., the one obtained after some steps in the least resp. greatest fixpoint iteration of γ^2_{KB}). This technique proves to be useful for helping the answer-set programming solver to converge to solutions faster.

4 System Prototype

The architecture of our system prototype is depicted in Figure 1. The system comprises six modules: the two external engines DLV and RACER, the latter embedded into a caching module, a WFS module, an answer-set semantics module, as well as a preprocessing and a postprocessing module. Each internal module is coded in the PHP scripting language; the overhead is insignificant, provided that most of the computing power is devoted to the execution of the two external reasoners.

Our prototypical implementation is capable of evaluating a dl-program in three different modes: (1) under answer-set semantics, (2) under WFS, and (3) under answer-set semantics with preliminary computation of the WFS.

In Mode (1), the answer-set semantics is computed through a preprocessing step, aimed at computing all those dl-atoms which do not depend from the program P itself. Then, an ordinary program P_d is generated whose models $M_1, \ldots, M_n$ are checked and filtered through several consistency checks performed by querying the RACER engine in an interleaved fashion. The stratified bottom portion of P_d is evaluated iteratively as in Subsection 3.2. Eventually, the system outputs a list of answer sets $M_{k_1}, \ldots, M_{k_m}$.

In Mode (2), we compute the well-founded semantics of a program P by generating a corresponding ordinary program P_d which is grounded using the grounding module of the DLV system. This instantiation $grd(P_d)$ is fed back to the well-founded semantics module, where an iterative algorithm, calling the RACER engine several times, is carried out in order to compute the well-founded semantics of P.

In Mode (3), the answer-set semantics is computed by taking advantage of the WFS which is combined with P_d in order to get a better constrained program, as described in Section 3.7.

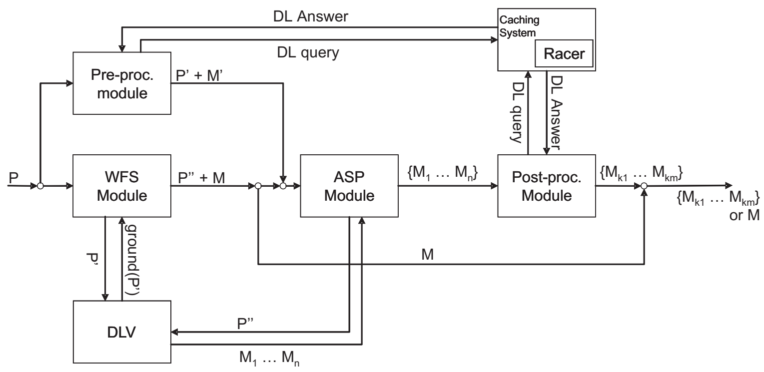

Fig. 1. System architecture of the dl-program evaluation prototype

5 Experiments

As mentioned in the previous section, we decided to exploit the scripting language PHP. Clearly, the speed and grade of optimization of PHP applications cannot be compared to ones natively compiled in a high-level programming language; however, during the development of our prototype, we realized that the major bottlenecks are the external reasoning applications. Thus, our benchmarks already show significant results with respect to different methods of integrating the external reasoners.

RACER's restriction of not allowing reasoning with nominals in concept definitions as well as its slow performance on large knowledge bases seriously limited the ability of performing realistic assertional knowledge reasoning tests with existing ontologies (e.g., the OWL wine ontology from [20]). For this reason, we decided to carry out the benchmarks with abstract, but well-scalable graph examples in addition to the already presented computer shop application.

The benchmarks were carried out on an AMD Athlon 1.2GHz CPU with 256MB RAM. We used the official DLV version of May 23th, 2004, and RACER version 1.7.23.

Positive Programs. In order to assess our evaluation strategy for positive dl-programs, we considered the computation of the transitive closure of a graph. We evaluated five graphs (taken from [18]) of different size with two different dl-programs, $KB_{LP} = (L_2, P_2)$ and $KB_{ONT} = (L_3, P_3)$, where:

$L_2 = \{arc(1,2);\ arc(1,4);\ \ldots\}$;
$P_2 = \{tc(X,Y) \leftarrow DL[arc \uplus tc; arc](X,Y); tc(X,Y) \leftarrow DL[arc](X,Z), tc(Z,Y)\}$;
$L_3 = L_2 \cup \mathrm{Trans}(arc)$;
$P_3 = \{tc(X,Y) \leftarrow DL[arc](X,Y)\}$.

Here, $\mathrm{Trans}(arc)$ denotes the DL transitivity axiom. Figure 2 shows the results against a logarithmic time scale. We display total evaluation times for KB_{ONT} and KB_{LP} as well as the respective time needed for querying the DL engine. The logarithmic scale shows very clearly that although KB_{ONT} scales as good as KB_{LP}, it is always two orders of magnitude slower than KB_{LP}. In both cases, a significant percentage of the overall execution time is spent by RACER calls.

The reason of feeding the extension of tc back to the DL knowledge base in L_2 is not obvious here at first sight. However, we wanted to simulate a situation where the

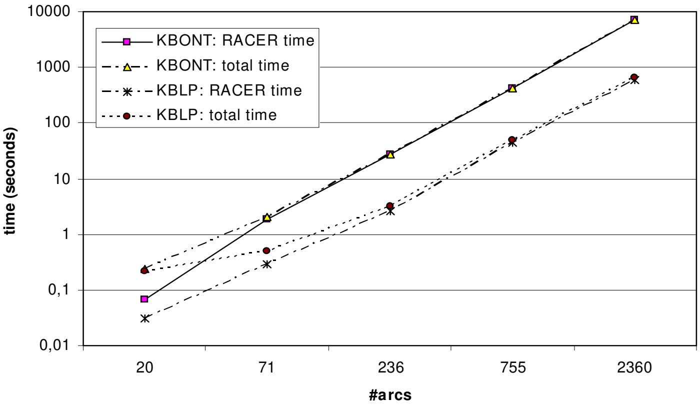

Fig. 2. Graph experiment benchmark results

terminological information enlarges the extension of the relation. We illustrate this with the following dl-program $KB = (L_4, P_4)$:

$L_4 = \{\exists R.\{c\} \sqsubseteq \exists R^-.\{d\};\ R(a,b);\ R(b,c)\};$
$P_4 = \{r(X,Y) \leftarrow DL[R \uplus r; R](X,Y);\ r(X,Y) \leftarrow r(X,Z), r(Z,Y)\}.$

The task of this program is to compute the transitive closure of R. In contrast to the graph example, here it is not possible to query the entire relation and compute the closure solely by rules, since the given subsumption axiom creates new tuples from existing ones, which makes it necessary to feed the inferred facts back to the DL reasoner. Unfortunately, we were not able to conduct experiments with such a scenario because RACER is not able to handle individuals in concept expressions.

Unstratified dl-programs. Unstratified dl-programs have been assessed exploiting Example 2.2. The data set at hand is constituted of about 20 individuals.

The computation of this example involves evaluating the least model of the stratified part, then the answer-set validation of the entire program. Figure 3 shows the result for three different evaluation scenarios.

In the first setting, we switched off the DL engine caching module: the number of DL calls is in this case very high, and stems from the fact that (almost) each query is preceded by calls that clone and extend the knowledge base at hand with facts coming from the logic program. Since RACER does not provide other ad-hoc features, this technique proves to be effective in order to quickly augment and restore a given knowledge base.

In the second case, we switched caching on, and this saved a lot of calls to RACER. The remaining computation time, apart from DLV and RACER external calls, is consumed mainly by a loop that examines answer sets for validity with respect to the DL knowledge base and also by initializing RACER. These two experiments involved the validation of 1280 answer sets, generated by the guessing mechanism for unstratified dl-atoms.

	total time	DLV time	RACER time	#cache hits	#DL calls
cache off	23.83	0.82	13.65	0	11535
cache on	9.65	0.81	0.26	3786	179
cache on, WFS first	6.57	1.02	4.50	152	137
cache on, $\mathit{lfp}(\gamma^2_{KB})$ first	5.82	0.61	0.11	2283	131

Fig. 3. Shop example results (time expressed in seconds).

In the third setting, we did a pre-computation of the WFS of the program before the answer-set generation. The pre-evaluation of this model limits the number of possible answer sets to 24, which narrows the execution time mainly to DL-calls.

An interesting variation of this method is to calculate only the positive facts of $WFS(KB)$, i.e., $\mathit{lfp}(\gamma^2_{KB})$. When this method is applied on the current example, some time-consuming calls to RACER that are involved in computing $\mathit{gfp}(\gamma^2_{KB})$ are avoided. However, the overall time is only slightly less, since this subset of $WFS(KB)$ reduces the number of answer sets to be checked only to 768.

In this last experiment with the unstratified dl-program, we considered to compute only a subset of the well-founded semantics prior to the answer-set generation. Although this resulted in a reduced overall execution time, this might not apply to other programs. As we pointed out, the advantage of having less calls to the DL reasoner by omitting $\mathit{gfp}(\gamma^2_{KB})$ is compensated by an increase of the answer sets that have to be checked for compliance with L. This tradeoff very much depends on the size of the assertional facts in the DL knowledge base as well as on the number of answer sets, i.e., on the specific design of the program and its stratification.

6 Conclusion

We have presented methods and algorithms for implementing nonmonotonic description logic programs. The issue of efficient interfacing between a description logic reasoner and an answer-set solver has been solved by means of several methods which can be fruitfully exploited for the implementation of systems of similar nature.

We assumed in most cases to deal with monotonic dl-atoms only. It is worth pointing out that this confinement benefits of useful nonmonotonic features in this setting as well. Our semantics provides a safe coupling between rule-based languages and description logics, since decidability is preserved. Furthermore, extending our semantics for dealing with many of the special features of answer-set programming systems (e.g., weak and integrity constraints, or aggregates) is quite straightforward.

Experimental results proved that description logics systems would benefit from this kind of coupling, since relieving a reasoner like RACER from some reasoning tasks, which such kind of systems are not aimed at, proved to be effective, also from a performance perspective. It turned out also that our system heavily relies on both reasoning systems, and would benefit from any performance improvement on both sides.

Our experimental prototype implementation, using DLV [15] and RACER [11], is available at

`http://www.kr.tuwien.ac.at/staff/roman/semweblp/.`

A $\mathcal{SHIF}$(D) and $\mathcal{SHOIN}$(D) Syntax

We briefly recall the elements of the description logics $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, starting with the latter. We assume a set $\mathbf{D}$ of *elementary datatypes*. Every $d \in \mathbf{D}$ has a set of *data values*, called the *domain* of d, denoted $\mathrm{dom}(d)$. We use $\mathrm{dom}(\mathbf{D})$ to denote $\bigcup_{d\in\mathbf{D}} \mathrm{dom}(d)$. A *datatype* is either an element of $\mathbf{D}$ or a subset of $\mathrm{dom}(\mathbf{D})$ (called *datatype oneOf*). Let $\mathbf{A}$, $\mathbf{R}_A$, $\mathbf{R}_D$, and $\mathbf{I}$ be nonempty finite and pairwise disjoint sets of *atomic concepts*, *abstract roles*, *datatype roles*, and *individuals*, respectively. We use $\mathbf{R}_A^-$ to denote the set of all inverses R^- of abstract roles $R \in \mathbf{R}_A$.

A *role* is an element of $\mathbf{R}_A \cup \mathbf{R}_A^- \cup \mathbf{R}_D$. *Concepts* are inductively defined as follows. Every $C \in \mathbf{A}$ is a concept, and if $o_1, o_2, \ldots \in \mathbf{I}$, then $\{o_1, o_2, \ldots\}$ is a concept (called *oneOf*). If C and D are concepts and $R \in \mathbf{R}_A \cup \mathbf{R}_A^-$, then $(C \sqcap D)$, $(C \sqcup D)$, and $\neg C$ are concepts (called *conjunction*, *disjunction*, and *negation*, respectively), as well as $\exists R.C$, $\forall R.C$, $\geq nR$, and $\leq nR$ (called *exists*, *value*, *atleast*, and *atmost restriction*, respectively) for an integer $n \geq 0$. If $d \in \mathbf{D}$ and $U \in \mathbf{R}_D$, then $\exists U.d$, $\forall U.d$, $\geq nU$, and $\leq nU$ are concepts (called *datatype exists*, *value*, *atleast*, and *atmost restriction*, respectively) for an integer $n \geq 0$. We write $\top$ and $\bot$ to abbreviate $C \sqcup \neg C$ and $C \sqcap \neg C$, respectively, and we eliminate parentheses as usual.

An *axiom* is of one of the following forms: (1) $C \sqsubseteq D$, where C and D are concepts (*concept inclusion*); (2) $R \sqsubseteq S$, where either $R, S \in \mathbf{R}_A$ or $R, S \in \mathbf{R}_D$ (*role inclusion*); (3) $\mathrm{Trans}(R)$, where $R \in \mathbf{R}_A$ (*transitivity*); (4) $C(a)$, where C is a concept and $a \in \mathbf{I}$ (*concept membership*); (5) $R(a, b)$ (resp., $U(a, v)$), where $R \in \mathbf{R}_A$ (resp., $U \in \mathbf{R}_D$) and $a, b \in \mathbf{I}$ (resp., $a \in \mathbf{I}$ and $v \in \mathrm{dom}(\mathbf{D})$) (*role membership*); and (6) $a = b$ (resp., $a \neq b$), where $a, b \in \mathbf{I}$ (*equality* (resp., *inequality*)).

A *knowledge base* L is a finite set of axioms. (For decidability, number restrictions in L are restricted to simple $R \in \mathbf{R}_A$ [14]).

$\mathcal{SHIF}(\mathbf{D})$ is the restriction of $\mathcal{SHOIN}(\mathbf{D})$ which excludes the *oneOf* constructor and limits the *atleast* and *atmost* constructors to 0 and 1.

For the semantics of $\mathcal{SHIF}(\mathbf{D})$ and $\mathcal{SHOIN}(\mathbf{D})$, we refer to [12] or [6].

B Example Ontology L_1

$\geq 1\ \mathit{buying} \sqsubseteq \mathit{Shop};\ \top \sqsubseteq \forall \mathit{buying}.\mathit{Part};\ \geq 2\ \mathit{buying} \sqsubseteq \mathit{Discount};$
$\mathit{Part}(\mathit{graphiccard});\ \mathit{Part}(\mathit{memory});\ \mathit{Part}(\mathit{fan});$
$\mathit{Part}(\mathit{harddisk});\ \mathit{Part}(\mathit{cdrom});\ \mathit{Part}(\mathit{dvdrom});$
$\mathit{Part}(\mathit{soundcard});\ \mathit{Part}(\mathit{cpu});\ \mathit{Part}(\mathit{wlan});\ \mathit{Part}(\mathit{case});$
$\mathit{provides}(s_1, \mathit{case});\ \mathit{provides}(s_1, \mathit{cpu});$
$\mathit{provides}(s_2, \mathit{dvdrom});$
$\mathit{provides}(s_3, \mathit{cpu});\ \mathit{provides}(s_3, \mathit{fan});\ \mathit{provides}(s_3, \mathit{wlan});$
$\mathit{provides}(s_4, \mathit{case});\ \mathit{provides}(s_4, \mathit{cdrom});\ \mathit{provides}(s_4, \mathit{harddisk});$
$\mathit{provides}(s_5, \mathit{cpu});\ \mathit{provides}(s_5, \mathit{harddisk});$
$\mathit{provides}(s_6, \mathit{graphiccard});\ \mathit{provides}(s_6, \mathit{soundcard});\ \mathit{provides}(s_6, \mathit{harddisk});$
$\mathit{provides}(s_7, \mathit{graphiccard});\ \mathit{provides}(s_7, \mathit{memory});$
$\mathit{provides}(s_8, \mathit{wlan});$
$\mathit{provides}(s_9, \mathit{case});\ \mathit{provides}(s_9, \mathit{harddisk}).$

References

1. G. Antoniou. Nonmonotonic Rule Systems on Top of Ontology Layers. In *Proc. ISWC 2002*, volume 2342 of *LNCS*, pages 394–398, 2002.
2. C. Baral and V. S. Subrahmanian. Dualities Between Alternative Semantics for Logic Programming and Nonmonotonic Reasoning. *J. Automated Reasoning*, 10(3):399–420, 1993.
3. T. Berners-Lee. *Weaving the Web*. Harper, San Francisco, CA, 1999.
4. T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):34–43, 2001.
5. F. M. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. $\mathcal{AL}$-log: Integrating Datalog and Description Logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.
6. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. In *Proc. KR 2004*, pages 141–151, 2004. Extended Report RR-1843-03-13, Institut für Informationssysteme, TU Wien, 2003.
7. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Well-founded Semantics for Description Logic Programs in the Semantic Web. In *Proc. RuleML 2004*, number 3323 in LNCS, pages 81–97. Springer, 2004.
8. W. Faber, N. Leone, C. Mateis, and G. Pfeifer. Using Database Optimization Techniques for Nonmonotonic Reasoning. In *Proc. DDLP-99*, pages 135–139. Prolog Association of Japan, September 1999.
9. D. Fensel, W. Wahlster, H. Lieberman, and J. Hendler, editors. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*. MIT Press, 2002.
10. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Deductive Databases. *New Generation Computing*, 17:365–387, 1991.
11. V. Haarslev and R. Möller. RACER System Description. In *Proc. IJCAR 2001*, volume 2083 of *LNCS*, pages 701–705, 2001.
12. I. Horrocks and P. F. Patel-Schneider. Reducing OWL Entailment to Description Logic Satisfiability. In *Proc. ISWC 2003*, volume 2870 of *LNCS*, pages 17–29, 2003.
13. I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From $\mathcal{SHIQ}$ and RDF to OWL: The Making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.
14. I. Horrocks, U. Sattler, and S. Tobies. Practical Reasoning for Expressive Description Logics. In *Proc. LPAR 1999*, volume 1705 of *LNCS*, pages 161–180, 1999.
15. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 2004. To appear.
16. A. Y. Levy and M.-C. Rousset. Combining Horn Rules and Description Logics in CARIN. *Artif. Intell.*, 104(1-2):165–209, 1998.
17. R. Rosati. Towards Expressive KR Systems Integrating Datalog and Description Logics: Preliminary Report. In *Proc. DL-99*, pages 160–164, 1999.
18. M. Trick. Graph Coloring Instances, 1994. `http://mat.gsia.cmu.edu/COLOR/instances.html`.
19. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The Well-Founded Semantics for General Logic Programs. *Journal of the ACM*, 38(3):620–650, 1991.
20. W3C. OWL Web Ontology Language Guide, 2003. W3C Proposed Recommendation 15 December 2003. `http://www.w3.org/TR/2003/PR-owl-guide-20031215/`.
21. W3C. OWL Web Ontology Language Overview, 2004. W3C Recommendation 10 February 2004. `www.w3.org/TR/2004/REC-owl-features-20040210/`.
22. K. Wang, D. Billington, J. Blee, and G. Antoniou. Combining Description Logic and Defeasible Logic for the Semantic Web. In *Proc. RuleML 2004*, number 3323 in LNCS, pages 170–181. Springer, 2004.