Merging and Aligning Ontologies in dl-Programs

Kewen Wang[1], Grigoris Antoniou[2], Rodney Topor[1], and Abdul Sattar[1]

[1] Griffith University, Australia
{k.wang,r.topor,a.sattar}@griffith.edu.au
[2] University of Crete, Greece
ga@csd.uoc.gr

Abstract. The language of dl-programs is a latest effort in developing an expressive representation for Web-based ontologies. It allows to build answer set programming (ASP) on top of description logic and thus some attractive features of ASP can be employed in the design of the Semantic Web architecture. In this paper we first generalize dl-programs by allowing multiple knowledge bases and then accordingly, define the answer set semantics for the dl-programs. A novel technique called forgetting is developed in the setting of dl-programs and applied to ontology merging and aligning.

1 Introduction

A key part of the Semantic Web architecture is designing a set of languages so that web-based ontologies can be represented and reasoned easily and correctly. The Web Ontology Language (OWL) is the latest standard recommended by the World Wide Web Consortium (W3C). The design and standardization of OWL is largely influenced by description logic [2]. As observed by many researchers, for example, [1, 7, 10, 11, 16], OWL is still too limited in representing, reasoning about and merging ontologies on the Web. In particular, the following three issues are still far from solved:

- *How to represent commonsense knowledge in ontologies.*
- *How to represent and reason with multiple ontologies.*
- *How to effectively reuse and share ontologies in the Semantic Web.*

Many researchers believe that the next step in the development of the Semantic Web is to realize the logic layer. This layer will be built on top of the ontology layer and provide sophisticated representation and reasoning abilities. Given that most current reasoning systems are based on rules, it is a key task to combine rules with ontologies. The RuleML initiative (http://www.ruleml.org) is considered to be a first attempt in this direction. Theoretically, the problem of integrating the ontology layer with the logic layer is reduced to combine rule-based systems with description logics. Recently, a number of attempts at combining description logic with logic programs have been made, for example, [5, 11]. A more recent work in [7] aimed to build nonmonotonic logic programs on the top of description logic (or OWL) by combining answer set programming (ASP) and description logic. In particular, the notion of dl-atoms allows to query and virtually align the dl-knowledge base.

ASP is a paradigm of logic programming under the answer sets [9] and it is becoming one of the major tools for knowledge representation due to its simplicity, expressive power, connection to major nonmonotonic logics and efficient implementations, such as DLV [6] and Smodels [14] However, some aspects of dl-programs introduced in [7] should be extended and improved:

- dl-programs can only query or align a fixed dl-knowledge base.
- there is no construct provided in dl-programs so that concepts from different ontologies can be used. This issue can be reduced to the problem of how to merge or align ontologies.
- The operator $\ominus$ provides a means to constrain some objects from a concept but there is no construct for discarding a concept. For example, suppose we want to define a concept of $Top100Singers$. We may wish to use an ontology "MUSICIAN" on the Web and thus have to import the ontology. However, we do not want to import some irrelevant concepts like "Violinist".

In this paper we first generalize the language of dl-programs and its semantics. Then the notion of forgetting [17] is introduced into dl-programs and thus show how to use this technique to merge and align different ontologis in dl-programs.

2 Preliminaries

The language for representing ontologies in this paper is a combination of a simple description logic and extended logic programs. In this section we briefly recall some background knowledge of logic programs, description logic, and their relation to Web markup languages.

2.1 Description Logic and Web Markup Languages

Although the Web is a great success, it is basically a collection of human-readable pages that cannot be automatically processed by computer programs.The Semantic Web is to provide tools for explicit markup of Web content and to help create a repository of computer-readable information. RDF is a language that can represent explicit metadata and separate content of Web pages from their structure. However, as noted by the W3C Web Ontology Working Group (http://www.w3.org/2001/sw/WebOnt/) , RDF/RDFS is too limited to describe some application domains which require the representation of ontologies on the Web and thus, a more expressive ontology modeling language was needed. This led to a number of ontology languages for the Web including the well-known DAML+OIL [3] and OWL [4]. In general, if a language is more expressive, then it is less efficient. To suit different applications, the OWL language provides three species for users to get a better balance between expressive power and reasoning efficiency: OWL Full, OWL DL and OWL Lite.

The cores of these Semantic Web languages are description logics, and in fact, the designs of OWL and its predecessor DAML+OIL were strongly influenced by description logics, including their formal semantics and language constructors. In these Semantic Web languages, an ontology is represented as a knowledge base in a description logic.

Description logics are a family of concept-based knowledge representation languages [2]. They are fragments of first order logic and are designed to be expressively powerful and have an efficient reasoning mechanism.

A dl-knowledge base L has two components: a TBox and an ABox.

The TBox specifies the vocabulary of an application domain, which is actually a collection of concepts (sets of individuals) and roles (binary relations between individuals). So the TBox can be used to assign names to complex descriptions. For example, we may have a concept named $area$ which specifies a set of areas in computer science. Suppose we have another concept $expert$ which is a set of names of experts in computer science. We can have a role $expertIn$ which relates $expert$ to $area$. For instance, $expertIn(John,$ *"Semantic Web"*$)$ means "John is an expert in the Semantic Web".

The ABox contains assertions about named individuals.

A dl-knowledge base can also reason about the knowledge stored in the TBox and ABox, although its reasoning ability is a bit too limited for some practical applications. For example, the system can determine whether a description is consistent or whether one description subsumes another description.

The knowledge in both the TBox and ABox are represented as formulas of the first order language but they are restricted to special forms so that efficient reasoning is guaranteed. For our purpose, we deal with a simple description logic called $\mathcal{S}AL$. The formulas in $\mathcal{S}AL$ are called *concept descriptions*. Elementary descriptions consists of both *atomic concepts* and *atomic roles*. Complex concepts are built inductively as follows (in the rest of this subsection, A is an atomic concept, C and D are concept descriptions, R is a role): A (atomic concept); $\top$ (universal concept); $\bot$ (bottom concept); $\neg A$ (atomic negation); $C \sqcap D$ (intersection); $C \sqcup D$ (union). Note that we can use $\sqcup$ and $\sqcap$ to represent $\forall R.C$ (value restriction) and $\exists R.C$ (existential quantification).

To define a formal semantics of concept descriptions, we need the notion of *interpretation*. An interpretation $\mathcal{I}$ of $\mathcal{S}AL$ is a pair $(\Delta, \cdot^{\mathcal{I}})$ where Δ is a non-empty set called the *domain* and $\cdot^{\mathcal{I}}$ is an interpretation function which associates each atomic concept A with a subset $A^{\mathcal{I}}$ of Δ and each atomic role R with a binary relation $R^{\mathcal{I}} \subseteq \Delta \times \Delta$. The function $\cdot^{\mathcal{I}}$ can be naturally extended to complex descriptions:

- $\top^{\mathcal{I}} = \Delta$
- $\bot^{\mathcal{I}} = \emptyset$
- $(\neg A)^{\mathcal{I}} = \Delta - A^{\mathcal{I}}$
- $(C \sqcap D)^{\mathcal{I}} = C^{\mathcal{I}} \cap D^{\mathcal{I}}$
- $(C \sqcup D)^{\mathcal{I}} = C^{\mathcal{I}} \cup D^{\mathcal{I}}$

A *terminology axiom* is of the form $C \sqsubseteq D$ or $C \equiv D$ where C and D are concepts (roles). An interpretation $\mathcal{I}$ satisfies $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$; it satisfies $C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$.

2.2 Extended Logic Programs

We deal with extended logic programs [9] whose rules are built from some atoms where default negation not and strong negation $\neg$ are allowed. A literal is either an atom a or its strong negation $\neg a$[3]. For any atom a, we say a and $\neg a$ are complementary literals.

If l is a literal, then $not\ l$ is called a *negative literal*. For any set S of literals, $not\ S = \{not\ l \mid l \in S\}$.

An *extended logic program* is a finite set of rules of the following form

$$l_0 \leftarrow l_1, \ldots, l_m, not\ l_{m+1}, \ldots, not\ l_n \tag{1}$$

where l_0 is either a literal or empty, each l_i is a literal for $i = 1, \ldots, n$, and $0 \leq m \leq n$. If l_0 is empty, then the rule is a *constraint*.

If a rule of form (1) contains no default negation, it is called *positive*; P is a *positive program* if every rule of P is positive.

If a rule of form (1) contains only negative literals, it is called *negative*; P is a *negative program* if every rule of P is negative.

Given a rule r of form (1), $head(r) = l_0$ and $body(r) = body^+(r) \cup not\ body^-(r)$ where $body^+(r) = \{l_1, \ldots, l_m\}$, $body^-(r) = \{l_{m+1}, \ldots, l_n\}$. The set $head(P)$ consists of all literals appearing in rule heads of P.

In the rest of this section we assume that P is an extended logic program and S is a set of ground literals. A rule r in P is satisfied by S, denoted $S \models r$, iff "if $body^+(r) \subseteq S$ and $body^-(r) \cap S = \emptyset$, then $head(r) \in S$". S is a model of P, denoted $S \models P$ if every rule of P is satisfied by S.

The answer set semantics The *reduct* of logic program P on a set S of literals, written P^S, is obtained as follows:

- Delete every r from P such that there is a $not\ q \in body^-(r)$ with $q \in S$.
- Delete all negative literals from the remaining rules.

Notice that P^S is a set of rules without any negative literals. Thus P^S may have no model or have a unique minimal model, which coincides with the set of literals that can be derived by resolution.

S is an *answer set* of P if S is the minimal model of P^S.

A logic program may have zero, one or more answer sets. We use $\parallel P \parallel$ to denote the collection of answer sets of P.

A program is *consistent* if it has at least one answer set.

Two logic programs P and P' are *equivalent*, denoted $P \equiv P'$, if they have the same answer sets.

As usual, B_P is the *Herbrand base* of logic program P, that is, the set of all (ground) literals in P.

[3] We use the same sign "$\neg$" to represent the negation in description logic and the strong negation in extended logic programs.

3 Answer Sets for dl-Programs

The dl-program introduced in this section is a generalization of the description logic program introduced in [7]. This language is a combination of the description logic $\mathcal{SAL}$ and extended logic programs, which allows to build logic programs on top of description logic (and thus some description logic-based web ontology languages like OWL). We will also define the answer set semantics for dl-programs.

3.1 Syntax

Informally, a dl-program consists of some dl-knowledge bases $\mathsf{L} = \{L_1, \ldots, L_s\}$ and a logic program P whose rule bodies may contain queries to some knowledge base (or its update) in L.

A *dl-query* $Q[t]$ is either a concept, or its negation, or a role, or its negation where t is a term. A *dl-atom* has the form $DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ where L is a dl-knowledge base, each S_i is either a concept or a role, each P_i is a predicate; each $\circ$ is either $\oplus$ or $\ominus$, and $Q[t]$ is a dl-query. Intuitively, $S_i \oplus P_i$ and $S_i \ominus P_i$ implement views of inserting and deleting the objects satisfying the property P_i (see Section 3.2 for formal definition).

Note that we do not include the third operator in [7] because it can be represented by $\ominus$. Thus we do not need the notion of monotonicity of dl-atoms.

In the ontology $MUSICIAN$, if we are not interested in Jazz music, dl-atom $DL[Singer \ominus jazz(x), Cui]$ can be used. When there is no confusion, the L in the dl-atom can be omitted.

Definition 1. *A* dl-rule *is of the form*

$$a \leftarrow b_1, \ldots, b_r, not\ b_{r+1}, \ldots, not\ b_n$$

where $a, b_{r+1}, \ldots, b_n$ *are ordinary atoms; each of* $b_1, \ldots, b_r$ *can be either an ordinary atom or a dl-atom.*

A dl-program *is a pair* (L, P) *where* L *is a set of knowledge bases in description logic and* P *is a finite set of dl-rules. Sometimes, we just say* P *is a dl-program if there is no confusion caused.*

The dl-programs here are a bit different from the programs introduced in [7] in that a multitude of dl-knowledge bases can be queried in the same program. This is more useful for Web-based ontology representation.

A dl-program (L, P) is *positive* if it does not contain negation as failure "not ".

The *dl-base* D_P of a dl-program P is defined as the set of all ground ordinary literals in P (including the literals appearing in dl-atoms). A ground instance of a rule $r \in P$ is a rule obtained by replacing every variable in r by a constant. $ground(P)$ denotes the set of all ground instances of P. An *interpretation* I of a dl-program P is a consistent set of literals in D_P.

Let us consider the following example, which extends a scenario from http://www.kr.tuwien.ac.at/staff/roman/asp_sw/.

Example 1. Suppose that we have an ontology called ARTISTS. According to the ontology, Artists are either Singers or Painters. Some artists in the ontology are "Jodie Nash", "Vincent Van Gogh", "Luciano Pavarotti".

We also have some rules for formalizing commonsense knowledge.

If someone is an artist and there is no evidence to show that he is a painter, then he is not a painter:

$$r_1 : \neg painter(A) \leftarrow DL[L, artist](A), not\ painter(A)$$

Similarly, if someone is an artist and there is no evidence to show that he is a singer, then he is not a singer:

$$r_2 : \neg singer(A) \leftarrow DL[L, artist](A), not\ singer(A).$$

Single out cases when an artist ought to be a painter or singer (but not necessarily both)

$$r_3 : painter(A) \leftarrow DL[L \oplus Wynne(X), artist](A), DL[L, painter](A)$$

Here $L \oplus Wynne(X)$ guarantees that a Wynne Prize Winner is treated as an artist in case he is not included the ontology.

$$r_4 : singer(A) \leftarrow DL[L \ominus Jazz(X), artist](A), DL[L, singer](A).$$

Here $DL[L \ominus Jazz(X)$ shows that we are not interested in Jazz music.

Suppose we have another dl-knowledge base L_1 to check if A is a singer:

$$r_5 : singer(A) \leftarrow DL[L_1, singer](A).$$

Let L be the ontology ARTIST and $P = \{r_1, r_2, r_3, r_4, r_5\}$. Then $(\{L\}, P)$ is a dl-program.

The above dl-program intends to provide a compact representation for an ontology which extends the following owl-ontology (we omit an ontology "Awards" containing "Wynne" and the class "Jazz" of "Artist").

```
<!DOCTYPE rdf:RDF [] > <rdf:RDF
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xmlns="file://artist#"
  xmlns:base="file://artist">

  <owl:Ontology rdf:ID="artist"/>

  <owl:Class rdf:ID="Painter" />
  <owl:Class rdf:ID="Singer" />
```

```
<owl:Class rdf:ID="Artist">
  <owl:unionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Painter" />
    <owl:Class rdf:about="#Singer" />
  </owl:unionOf>
</owl:Class>

<Artist rdf:ID="Jodie_Nash"/>
<Painter rdf:ID="Vincent_Van_Gogh"/>
<Singer rdf:ID="Luciano_Pavarotti"/>

  </rdf:RDF>
```

3.2 Semantics

The semantics for a dl-program (L, P) is defined by the *dl-answer sets*, which modify and generalize the corresponding notion in [7].

An interpretation I of a dl-program P is a consistent set of ground literals. So we also allow dl-atoms in an interpretation.

The following interpretations for operators $\oplus$ and $\ominus$ is expected to remedy some problems in the original definition.

Definition 2. *Let I be an interpretation of the dl-program P and $DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ is a ground dl-atom in P.*

Denote $\odot^I$ the interpretation obtained from $\cdot^I$ by replacing $(S_i)^I$

- *with $(S_i)^I \cup \{t \mid P_i(t)\}$ if S_i is a concept and $\circ = \oplus$;*
- *with $(S_i)^I - \{t \mid P_i(t)\}$ if S_i is a concept and $\circ = \ominus$;*
- *with $(S_i)^I \cup \{(t_1, t_2) \mid P_i(t_1, t_2)\}$ if S_i is a role and $\circ = \oplus$;*
- *with $(S_i)^I - \{(t_1, t_2) \mid P_i(t_1, t_2)\}$ if S_i is a role and $\circ = \ominus$;*

The resulting semantic relation is denoted $\models_{dl}$.

$L \models DL[L, S_1 \circ P_1, \ldots, S_m \circ P_m, Q](t)$ if and only if $L \models_{dl} Q(t)$.

Note that every positive dl-program P has a least model, denoted $M(P)$. A general dl-program can be reduced into a positive dl-program with respect to an interpretation.

Let P be a dl-program and I a set of atoms in P. The reduct P^I of P on I is the positive dl-program obtained from $ground(P)$ by the following three ordered steps :

1. adding a rule $Q(t) \leftarrow$ for each dl-atom d in the body of a rule in $ground(P)$ such that $L \models Q(t)$, where $Q(t)$ is the query for d;
2. deleting every rule r in $ground(P)$ such that $b \in I$ for some $not\ b$ in the body of r and
3. deleting every $not\ b$ in the remaining rules.

The first condition says a dl-atom must be true in the model if it occurs in the program and can be derived from the corresponding dl-knowledge base; the second and third conditions are inherited from the definition of standard answer sets.

Definition 3. *Let P be a dl-program and S a set of literals in P. S is a dl-answer set if $M(P^S) = S$.*

A dl-answer set S is *consistent* if (1) there is no atom a such that both a and $\neg a$ in S and (2) if d is a dl-atom and $d \in S$, then $L \not\models_{dl} \neg Q(t)$ where $Q(t)$ is the query of d.

A dl-program may have zero, one or more dl-answer sets. We use $\parallel P \parallel$ to denote the collection of answer sets of P.

Consider the dl-program $(\{L, L_1\}, P)$ again where L_1 is a dl-knowledge base which includes "Tweety" as a singer. Then this dl-program has the unique dl-answer set which contains some information like $painter(VincentVanGogh)$ and $singerTweety$.

A dl-program is *consistent* if it has at least one consistent dl-answer set.

Two dl-programs P and P' are *equivalent*, denoted $P \equiv P'$, if they have the same dl-answer sets.

4 Forgetting in dl-Programs

In this section we introduce the notion of forgetting for dl-programs. That is, we want to define what it means to forget about (or discard) a literal l in a dl-program P. The intuition behind the forgetting theory is to obtain a dl-program which is equivalent to the original dl-program if we ignore the existence of the literal l.

4.1 Forgetting Ordinary Atoms

It is easy to forget a literal l in a set X of literals, that is, just remove l from X if $l \in X$. This notion of forgetting can be easily extended to subsets. A set X' is an l-subset of X if $X' - \{l\} \subseteq X - \{l\}$. Similarly, a set X' is a true l-subset of X if $X' - \{l\} \subset X - \{l\}$.

Two sets X and X' of literals are l-equivalent, denoted $X \sim_l X'$, iff $(X - X') \cup (X' - X) \subseteq \{l\}$.

Given a consistent dl-program P and an ordinary ground literal l, we could define a result of forgetting about l in P as a dl-program P' whose dl-answer sets are exactly $\parallel P \parallel -l = \{X - \{l\} \mid X \in \parallel P \parallel\}$. However, such a notion of forgetting cannot even guarantee the existence for some simple programs as illustrated in [17]. So *we need a notion of minimality of dl-answer sets which can naturally combine the definition of dl-answer sets, minimality and forgetting together.*

Definition 4. *Let P be a consistent dl-program, l an ordinary ground literal in P and X a set of ground literals.*

1. *We say X is* l-minimal *in a collection $\mathcal{S}$ of sets of ground literals if $X \in \mathcal{S}$ and there is no $X' \in \mathcal{S}$ such that X' is a true l-subset of X . In particular, if $\mathcal{S}_P$ is the set of models of P, then we say X is an* l-minimal model *of dl-program P if X is a model of P and it is l-minimal in $\mathcal{S}_P$.*
2. *X is a* dl-answer set of P by forgetting *l (briefly,* l-answer set*) if X is the l-minimal model of the reduct P^X.*

The above definition is a filter for dl-answer sets rather than a new semantics.

Having the notion of minimality about forgetting an ordinary ground literal, we are now in a position to define the result of forgetting about a literal in a dl-program.

Definition 5. *Let P be a consistent dl-program and l be an ordinary ground literal. A dl-program P' is a result of* forgetting about l in P *if the following conditions are satisfied:*

1. $D_{P'} \subseteq D_P - \{l\}$.
2. *For any set X' of ground literals, X' is a dl-answer set of P' iff there is an l-answer set X of P such that $X' \sim_l X$.*

Notice that the first condition implies that l does not appear in P'. In particular, no new symbol is introduced in P'.

Suppose we have a dl-knowledge base L which contains some concepts "bird", "parrot" and "penguin". An ontology "BIRD" is specified as a dl-program (L, P) where $P = P_1 \cup P_2$, P_2 contains no information about "penguin" and P_1 consists of the following rules:

$$\begin{aligned} bird(A) &\leftarrow penguin(A) \\ flies(A) &\leftarrow bird(A), not\ penguin(A) \\ \neg flies(A) &\leftarrow penguin(A) \\ penguin(Tweety) &\leftarrow \end{aligned}$$

If we do not want to import the concept "penguin", we can discard the information on "penguin" by forgetting and get $\mathsf{forget}((L, P), penguin) = \{flies(A) \leftarrow bird(A)\} \cup P_2$.

A dl-program P may have different dl-programs as results of forgetting about the same ordinary ground literal l. However, it follows from the above definition that any two results of forgetting about the same literal in P are equivalent under dl-answer sets.

Proposition 1. *Let P be a dl-program and l an ordinary ground literal in P. If P' and P'' are two results of forgetting about l in P, then P' and P'' are equivalent (i.e. they have the same dl-answer sets).*

We use $\mathsf{forget}(P, l)$ to denote the result of forgetting about l in P.

To compute $\mathsf{forget}(P, l)$, we can easily adapt the corresponding algorithms in [17] to dl-programs.

Similarly, we can forget a set of ordinary literals F in a dl-program P and thus define $\mathsf{forget}(P, F)$.

4.2 Forgetting dl-Atoms

To discard or forget an unwanted ground dl-atom d in a dl-program P, we need to remove all the effects caused by d in both P and L. This can be accomplished by the following steps:

Step 1. Forget d in dl-knowledge base L by removing all those concepts, roles and terminology axioms of L in which $Q(t)$ occurs. Denote the resulting dl-knowledge base $L - d$.

Step 2. Replace each occurrence of the dl-atom d by d' in P where d' is obtained from d by replacing L with $L - d$. The resulting dl-program is denoted P'.

Step 3. Forget the dl-atom d' in P' by treating d' as an ordinary atom.

However, a concept can also be forgotten from the knowledge base L but we will discuss it else where.

5 Merging and Aligning Ontologies

In recent years, researchers have developed many ontologies. These different groups of researchers are now beginning to work with each other, so they must bring together ontologies from different sources. Approaches to this problem usually fall into one of the two categories:

- merging the ontologies to create a single coherent ontology.
- aligning the ontologies by establishing links between them to reuse information from one another.

In this section we show how to merge and align ontologies in dl-programs by forgetting.

For simplicity, throughout the discussion, we assume that only two ontologies are being merged or aligned.

5.1 Merging Ontologies by Forgetting

When two ontologies are merged, a new ontology is created, which is a merged version of the original ontologies. Usually, overlapping domains are kept in the merged ontology. Some algorithms like SMART [15] tried to automate parts of the merging process of ontologies. However, their languages are relatively simple and thus reasoning is almost not involved.

As shown in the algorithm of SMART, those concepts and roles to be merged can be specified by users and/or automatic processes. For instance, Linguistically similar names can be found automatically. Linguistic similarity can be determined in a couple of different ways including by synonymy or shared substrings.

Let O_1 and O_2 be two ontologies expressed as dl-programs. Suppose we have determined two sets of literals F_1 and F_2 for O_1 and O_2, respectively. F_1 and F_2 correspond to certain concepts that need to be handled separately in the merging. A literal is put into F_1 or/and F_2 due to a number of reasons. However, in an expressive language like dl-programs, these two concepts may be related to some other concepts by terminology axioms. Thus we may have to deal with some issues related to reasoning like conflict resolving and consistency maintaining. Another possibility is that some concepts are useless and we want to discard them as mentioned in the introduction. The second scenario can be satisfactorily handled by the following algorithm.

Algorithm 1 Input*: Two ontologies O_1 and O_2 (in dl-programs).*
Output*: A merged ontology O.*
Process*:*

Step 1. *Determine the sets F_1 and F_2 of literals that need to be handled separately.*
Step 2. *Compute* $\mathsf{forget}(O_i, F_i)$ *for* $i = 1, 2$.
Step 3. *F_1 and F_2 are handled by user and thus O_0 is obtained.*
Step 4. *Merged ontology:* $O = O_0 \cup \mathsf{forget}(O_1, F_1) \cup \mathsf{forget}(O_2, F_2)$.

As for Step 3, it depends on the application. There may be a couple of possible different approaches. For example, we may remove some literals from F_1 and/or F_2; we may replace some literal of one F_i with a literal in the other; or we may even replace two literals $l_i \in F_i$, $i = 1, 2$ with a new literal.

5.2 Aligning Ontologies by Forgetting

In alignment, the two original ontologies persist but one of the aligned ontologies (say O_1, more general) is preferred over another (say O_2, more specific), with links established between their concepts and roles. These links can be represented as a mapping or a view (virtual ontology). However, the domain-specific ontology O_2 does not become part of the more general ontology O_1; rather O_2 is a separate ontology that includes O_1 and uses O_1's top-level distinctions. For example, many ontologies in the domain of military are structured around a central ontology, the CYC knowledge base [13]. The developers of these domain-specific ontologies then align their ontologies to CYC by establishing links into CYC's upper- and middle-level ontologies [8]. However, in many cases the alignment cannot be done by only some simple links. As in the case of merging, conflicts and/or inconsistencies may arise when aligning two ontologies. In particular, we have to deal with their subsumption relations. For example, if we have two ontologies ARTIST, where a concept "Singer" is included, and MUSICIAN, where a concept "singer" is included, we may wish to merge "singer" and "Singer". Moreover, ARTIST is established by the Australian Association of Arts and MUSICIAN by the College of Music at Griffith University. Since both "Singer" and "singer" may be related to some other concepts in their ontologies by roles and axioms, we cannot simply replace "singer" by "Singer".

By employing the notion of forgetting, the tasks of conflict resolution and consistency maintenance during aligning can be done automatically.

Algorithm 2 Input*: Two ontologies O_1 and O_2 (in dl-programs) where O_1 is preferred to O_2.*
Output*: Aligned ontology O.*
Process*:*

Step 1. *Determine the set F_2 of atoms that will be aligned in O_2.*
Step 2. *Compute* $\mathsf{forget}(O_2, F_2)$.
Step 3. *Aligned ontology:* $O = O_1 \cup \mathsf{forget}(O_2, F_2)$.

6 Conclusions

The language of dl-programs is a latest effort in developing an expressive representation for Web-based ontologies. It allows to build answer set programming (ASP) on top of description logic and thus some attractive features of ASP can be employed in the design of the Semantic Web architecture. Often, an ontology on the Web is based on more than one knowledge base. In this paper we have generalized dl-programs by allowing multiple knowledge bases and then accordingly, defined the answer set semantics for the dl-programs. The notion of forgetting has been proved an extremely useful technique for updating knowledge bases, constraint problem solving and query answering [12, 17, 18]. In this paper we have imported the notion of forgetting into dl-programs. We have also applied the technique of forgetting to two important tasks of representing ontologies, that is, merging and aligning ontologies. In particular, we have introduced two algorithms for these two tasks. This is only preliminary report of our work. There are a couple of issues to be pursued in the future:

– More constructs in ASP can be introduced into dl-programs, like disjunction and preference. The major difficulty in allowing these constructs is how to design corresponding algorithms for forgetting.
– It is also important to see if more expressive description logic can be allowed in the forgetting of dl-programs.
– The two algorithms for merging and aligning ontologies need further improvement. We are also planning to implement them and apply to some practical application domains.

References

1. G. Antoniou and G. Wagner. A rule-based approach to the semantic web (preliminary report). In *Proceedings of the 2nd Workshop on Rules and Rule Markup Languages for the Semantic Web (RuleML2003)*, pages 111–120, 2003.
2. F. Baader, D. Calvanese, D.McGuinness, D. Nardi, and P. Patel-Schneider. *The Description Logic Handbook*. Cambridge University Press, 2002.
3. D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. Daml+oil reference description. http://www.w3.org/tr/2001/note-daml+oil-reference-20011218.html, W3C Note, 18 December 2001.
4. M. Dean, D. Connolly, F. van Harmelen, J. Hendler, I. Horrocks, D. McGuinness, P. Patel-Schneider, and L. Stein. Owl web ontology language reference. http://www.w3.org/tr/2004/rec-owl-ref-20040210/, 3C Recommendation, 10 February 2004.
5. F. Donini, M. Lenzerini, D. Nardi, and A. Schaerf. AL-log: Integrating datalog and description logics. *Journal of Intelligent Information Systems*, 10(3):227–252, 1998.
6. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. A kr system dlv: Progress report, comparisons and benchmarks. In *Proceedings of the Sixth International Conference on the Principles of Knowledge Representation and Reasoning*, pages 406–417. Morgan Kaufmann Publishers, 1998.
7. T. Eiter, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. In *Proceedings of the 9th International Conference on Principles of Knowledge Representation and Reasoning*, pages 141–151, 2004.
8. R. Fikes and A. Farquhar. Large-scale repositories of highly expressive reusable knowledge. *IEEE Intelligent Systems*, 14(2):73–79, 1999.
9. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.
10. B. Grau, B. Parsia, and E. Sirin. Combining owl ontologies using e-connections. Technical Report TR-2005-01, University of Maryland Institute for Advanced Computer Studies (UMIACS), January 2005.
11. B. Grosof, I. Horrocks, R. Volz, and S. Decker. Description logic programs: Combining logic programs with description logics. In *Proceedings of the 12th International World Wide Web Conference*, pages 48–57, 2003.
12. J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.
13. D. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of ACM*, 38(11):33–38, 1995.
14. I. Niemelä and P. Simons. Smodels: An implementation of the stable model and well-founded semantics for normal logic programs. In J. Dix, U. Furbach, and A. Nerode, editors, *Proceedings of the Fourth International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 420–429. Springer-Verlag, 1997.

15. N. Noy and M. Musen. An algorithm for merging and aligning ontologies: Automation and tool support. In *Proceedings of the Workshop on Ontology Management at AAAI-99*, 1999.
16. T. Swift. Deduction in ontologies via asp. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer-Verlag, 2004.
17. K. Wang, A. Sattar, and K. Su. A theory of forgetting in logic programming. In *Proceedings of the AAAI National Conference on Artificial Intelligence*. AAAI Press, 2005.
18. Y. Zhang, N. Foo, and K. Wang. Solving logic program conflicts through strong and weak forgettings. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 627–632. the Professional Book Centre, USA, 2005.