Evolution of Ontologies using ASP

Max Ostrowski[1], Giorgos Flouris[2], Torsten Schaub[3], and Grigoris Antoniou[4]

1,3 Universität Potsdam
2 FORTH-ICS
4 University of Crete

Abstract

RDF/S ontologies are often used in e-science to express domain knowledge regarding the respective field of investigation (e.g., cultural informatics, bioinformatics etc). Such ontologies need to change often to reflect the latest scientific understanding on the domain at hand, and are usually associated with constraints expressed using various declarative formalisms to express domain-specific requirements, such as cardinality or acyclicity constraints. Addressing the evolution of ontologies in the presence of ontological constraints imposes extra difficulties, because it forces us to respect the associated constraints during evolution. While these issues were addressed in previous work, this is the first work to examine how ASP techniques can be applied to model and implement the evolution process. ASP was chosen for its advantages in terms of a principled, rather than ad hoc implementation, its modularity and flexibility, and for being a state-of-the-art technique to tackle hard combinatorial problems. In particular, our approach consists in providing a general translation of the problem into ASP, thereby reducing it to an instance of an ASP program that can be solved by an ASP solver. Our experiments are promising, even for large ontologies, and also show that the scalability of the approach depends on the morphology of the input.

1998 ACM Subject Classification I.2.4 Semantic Networks

Keywords and phrases Ontology evolution, Evolution in the presence of constraints, incremental ASP application

Digital Object Identifier 10.4230/LIPIcs.ICLP.2011.16

1 Introduction

Semantic Web [3], aims to extend the current web so as to allow information to be both understandable by humans and processable by machines. Ontologies describe our understanding of the physical world in a machine-processable format and form the backbone of the Semantic Web. They are usually represented using the RDF/S [13, 4] language; in a nutshell, RDF/S permits the representation of different types of resources like individuals, classes of individuals and properties between them, as well as basic taxonomic facts (such as subsumption and instantiation relationships).

Several recent works [16, 14, 12, 5, 19] have acknowledged the need for introducing constraints in ontologies. Given that RDF/S does not impose any constraints on data, any application-specific constraints (e.g., functional properties) or semantics (e.g., acyclicity in subsumptions) can only be captured using constraints on top of RDF/S data. In this paper, we consider DED constraints [8], which form a subset of first-order logic and have been shown to allow capturing many useful types of constraints; we will consider populated ontologies represented using RDF/S, and use the term *RDF/S knowledge base* (KB) to denote possibly interlinked and populated RDF/S ontologies with associated (DED) constraints.

© Max Ostrowski, Giorgos Flouris, Torsten Schaub and Grigoris Antoniou;
licensed under Creative Commons License NC-ND
Technical Communications of the 27th International Conference on Logic Programming (ICLP'11).
Editors: John P. Gallagher, Michael Gelfond; pp. 16–27

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

An important task towards the realization of the Semantic Web is the introduction of techniques that allow the efficient and intuitive evolution of KBs in the presence of constraints. Note that a *valid* evolution result should satisfy the constraints; this is often called the *Principle of Validity* [1]. In addition, the *Principle of Success* [1] should be satisfied, which states that the change requirements take priority over existing information, i.e., the change must be applied in its entirety. The final important requirement is the *Principle of Minimal Change* [1], which states that, during a change, the modifications applied upon the original KB must be minimal. In other words, given many different evolution results that satisfy the principles of success and validity, one should return the one that is "closer" to the original KB, where "closeness" is an application-specific notion. The above non-trivial problem was studied in [11], resulting in a general-purpose changing algorithm that satisfies the above requirements. Unfortunately, the problem was proven to be exponential in nature, so the presented general-purpose algorithmic solution to the problem (which involved a recursive process) was inefficient.

ASP is a flexible and declarative approach to solve NP-hard problems. The solution that was presented in [11] regarding the problem of ontology evolution in the presence of constraints can easily be translated into a logic program with first-order variables; this is the standard formalism that is used by ASP, which is then grounded into a variable free representation by a so called *grounder* that is then solved by a highly efficient Boolean *solver*. As it is closely related to the SAT paradigm, knowledge about different techniques for solving SAT problems are incorporated into the ASP algorithms. Using first-order logic programs is a smart way to represent the evolution problem while remaining highly flexible, especially with respect to the set of constraints related to the ontology.

The objective of the present work is to recast the problem of ontology evolution with constraints in terms of ASP rules, and use an efficient grounder and ASP solver to provide a modular and flexible solution. In our work, we use *gringo* for the grounding and *clasp* for the solving process as they are both state-of-the-art tools to tackle ASP problems [9]. Our work is based on the approach presented in [11], and uses similar ideas and notions. The main contribution of this work is the demonstration that ASP can be used to solve the inherently difficult problem of ontology evolution with constraints in a decent amount of time, even for large real-world ontologies. ASP was chosen for its advantages in terms of a principled, rather than ad hoc implementation, its modularity and flexibility, and for being a state-of-the-art technique to tackle hard combinatorial problems.

In the next section we present the problem of ontology evolution in the presence of constraints, and the solution proposed in [11]. In Section 3, we present ASP. Section 4 is the main section, where our formulation of the problem in terms of an ASP program is presented and explained. This approach is refined and optimized in Section 5. We present our experiments in Section 6 and conclude in Section 7.

2 Problem Statement

2.1 RDF/S

The RDF/S [13, 4] language uses triples of the form (subject, predicate, object) to express knowledge. RDF/S permits the specification of various entities (called *resources*), which may be classes (i.e., collections of resources), properties (i.e., binary relations between resources), and individuals (i.e., atomic entities). We use the symbol $type(u)$ to denote the type of a resource u (class, property, individual). RDF/S supports various predefined relations between resources, like the domain and range of properties, subsumption relationships between classes

Table 1 Representation of RDF/S Triples Using Predicates

RDF/S triple	Intuitive meaning	Predicate
c rdf:type rdfs:Class	*c* is a class	$cs(c)$
x rdf:type rdfs:Resource	*x* is an individual	$ci(x)$
c_1 *rdfs:subClassOf* c_2	IsA between classes	$c_IsA(c_1, c_2)$
x rdf:type c	class instantiation	$c_Inst(x, c)$

and between properties, and instantiation relationships between individuals and classes, or between pairs of individuals and properties. RDF/S associates such relations with semantics, e.g., subsumption is transitive.

RDF/S KBs are commonly represented as labeled graphs, whose nodes are resources and edges are relations (see Fig. 1). In Fig. 1, a, b, and c are classes and x is an individual. Solid arrows represent subsumption relationships between classes (e.g., b is a subclass of c), and dashed arrows represent instantiation relationships (e.g., x is an instance of b). The bold arrow represents the change we want to make, namely to make a a subclass of b.

2.2 Ontology Evolution Principles

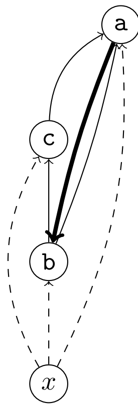

Figure 1 A knowledge base with change (appearing as bold arrow)

In the presence of constraints in the ontology, one should make sure that the evolution result is valid, i.e., it does not violate any constraints. This is called the Principle of Validity [1]. Manually enforcing this principle is an error-prone and tedious process. The objective of this work is to assist knowledge engineers in applying their changes in an automated manner, while making sure that no invalidities are introduced in the KB during the evolution.

In addition to the Validity Principle, two other principles are usually considered. The first is the *Principle of Success* [1], stating that the required changes take priority over existing information, i.e., the change must be applied in its entirety. The second is the *Principle of Minimal Change* [1], which requires that the modifications applied upon the original KB to accommodate the change must be minimal. Thus, if there are several different results that satisfy the principles of success and validity, one should return the one that is “closer” to the original KB, i.e., causes the least important modifications. Note that the importance of modifications (i.e., “closeness”) is an application-specific notion; in this work, we model “closeness” using a relation; details on this relation will be given later.

2.3 Formal Setting

To address the problem of ontology evolution, we use the general approach presented in [11]. An RDF/S KB $\mathcal{K}$ is modeled as a set of ground facts of the form $p(\vec{x})$ where p is a predicate and $\vec{x}$ is a vector of constants. Constants represent resources in RDF/S parlance, and each predicate represents one type of RDF/S relationship (e.g., domain, range, subsumption etc). For example, the triple $(a, \mathit{rdfs}{:}\mathit{subClassOf}, b)$, which denotes that a is a subclass of b, is represented by the ground fact $c_IsA(a, b)$. For the rest of the paper, predicates and constants will start with a lower case letter, whereas variables will start with an upper case letter. Table 1 shows some of the predicates we use and their intuitive meaning (see [11] for a complete list).

We assume closed world, i.e., $\mathcal{K} \nvdash p(\vec{x})$ whenever $p(\vec{x}) \notin \mathcal{K}$. A *change* $\mathcal{C}$ is a request to

Table 2 Ontological Constraints

ID, Constraint	Intuitive Meaning
$R5$: $\forall U, V$ $c_IsA(U,V) \rightarrow cs(U) \wedge cs(V)$	Class subsumption
$R12$: $\forall U, V, W$ $c_IsA(U,V) \wedge c_IsA(V,W) \rightarrow c_IsA(U,W)$	Class IsA transitivity
$R13$: $\forall U, V$ $c_IsA(U,V) \wedge c_IsA(V,U) \rightarrow \bot$	Class IsA irreflexivity

Table 3 Facts from example in Fig. 1

$\mathcal{K}_0$		$ci(x)$	
	$cs(a)$	$cs(b)$	$cs(c)$
	$c_IsA(b,c)$	$c_IsA(b,a)$	$c_IsA(c,a)$
	$c_Inst(x,b)$	$c_Inst(x,c)$	$c_Inst(x,a)$
$\mathcal{C}$		$c_IsA(a,b)$	

add/remove fact(s) to/from the KB, and it is modeled as a set of positive/negative ground facts.

Ontological constraints are modeled using DED rules [8], which allow for formulating various useful constraints, such as primary and foreign key constraints (used, e.g., in [12]), acyclicity and transitivity constraints for properties (as in [16]), and cardinality constraints (used in [14]). Here, we use the following simplified form of DEDs, which still includes the above constraint types:

$$\forall \vec{U} \bigvee_{i=1,\ldots,head} \exists \vec{V_i} q_i(\vec{U},\vec{V_i}) \leftarrow e(\vec{U}) \wedge p_1(\vec{U}) \wedge \cdots \wedge p_{body}(\vec{U}),$$

where $e(\vec{U})$ is a conjunction of (in)equality atoms. We denote by p the facts $p_1(\vec{U}), \ldots, p_{body}(\vec{U})$ and by q the facts $q_1(\vec{U},\vec{V_1}), \ldots, q_{head}(\vec{U},\vec{V}_{head})$. Table 2 shows some of the constraints used in this work; for a full list, refer to [11]. We say that a KB $\mathcal{K}$ *satisfies* a constraint r (or a set of constraints $\mathcal{R}$), iff $\mathcal{K} \vdash r$ ($\mathcal{K} \vdash \mathcal{R}$). Given a set of constraints $\mathcal{R}$, $\mathcal{K}$ is *valid* iff $\mathcal{K} \vdash \mathcal{R}$.

Now consider the KB $\mathcal{K}_0$ and the change of Fig. 1, which can be formally expressed using the ground facts of Table 3. To satisfy the principle of success, we should add $c_IsA(a,b)$ to $\mathcal{K}_0$, getting $\mathcal{K}_1 = \mathcal{K}_0 \cup \{c_IsA(a,b)\}$. The result ($\mathcal{K}_1$) is called the *raw application* of $\mathcal{C}$ upon $\mathcal{K}_0$, and denoted by $\mathcal{K}_1 = \mathcal{K}_0 + \mathcal{C}$. $\mathcal{C}$ is called a *valid change* w.r.t. $\mathcal{K}_0$ iff $\mathcal{K}_0 + \mathcal{C}$ is valid. In our example, this is not the case, because $\mathcal{K}_1$ violates $R13$; thus, it does not constitute an acceptable evolution result. The form of the violated rule implies that the only possible solution to this problem is to remove $c_IsA(b,a)$ from $\mathcal{K}_1$ (removing $c_IsA(a,b)$ is not an option, because its addition is dictated by the change – cf. the Principle of Success). This is an extra modification, that is not part of the original change, but is, in a sense, enforced by it; such extra modifications are called *side-effects*.

We note that the result, $\mathcal{K}_2 = \mathcal{K}_0 \cup \{c_IsA(a,b)\} \setminus \{c_IsA(b,a)\}$ is no good either, because $R12$ is violated, so, we need to repeat the above process recursively for $\mathcal{K}_2$. Note that $R12$ can be resolved in more than one ways, each of which should be evaluated independently; this fact leads to a recursive tree of resolutions (and side-effects). Eventually, after possibly several recursive steps, we will reach one or more valid KBs (leaves in the resolution tree);

these are possible results for the evolution, as they satisfy the principles of success and validity. In our example, these are: $\mathcal{K}_{4.1} = \mathcal{K}_0 \cup \{c_IsA(a,b), c_IsA(c,b)\} \setminus \{c_IsA(b,a), c_IsA(b,c)\}$ and $\mathcal{K}_{4.2} = \mathcal{K}_0 \cup \{c_IsA(a,b), c_IsA(a,c)\} \setminus \{c_IsA(b,a), c_IsA(c,a)\}$.

It remains to determine the "preferable" KB, i.e., the one that is "closest" to $\mathcal{K}_0$. To do so, we first determine the "distance" between KBs using difference sets, called *deltas*, which contain the positive/negative ground facts that need to be added/removed from one KB to get to the other (denoted by $\Delta(\mathcal{K}, \mathcal{K}')$). In our example, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) = \{c_IsA(a,b), c_IsA(c,b), \neg c_IsA(b,a), \neg c_IsA(b,c)\}$, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.2}) = \{c_IsA(a,b), c_IsA(a,c), \neg c_IsA(b,a), \neg c_IsA(c,a)\}$. Then, we can determine "closeness" using an ordering that ranks $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}), \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$; both deltas have the same size (and this occurs often), so ranking cannot be based on cardinality, but should also consider more subtle differences, like the severity of changes.

Here, we consider the ordering defined in [11], which is denoted by $<_{\mathcal{K}_0}$, where $\mathcal{K}_0$ the original KB. To define $<_{\mathcal{K}_0}$, we first order the available predicates in terms of severity ($<_{pred}$); for example, the addition of a class (predicate *cs*) is more important than the addition of a subsumption (predicate *c_IsA*), i.e., $c_IsA <_{pred} cs$. Then, Δ_1 is preferable than Δ_2 (denoted by $\Delta_1 <_{\mathcal{K}_0} \Delta_2$) iff the most important predicate (per $<_{pred}$) appears less times in Δ_1. In case of a tie, the next most important predicate is considered, and so on. If the deltas contain an equal number of ground facts per predicate, the ordering considers the constants involved: a constant is considered more important if it occupies a higher position in its corresponding subsumption hierarchy in the original KB. In this respect, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1})$ causes less important changes upon $\mathcal{K}_0$ than $\Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$, because the former affects b, c ($c_IsA(c,b), \neg c_IsA(b,c)$) whereas the latter affects c, a ($c_IsA(a,c), \neg c_IsA(c,a)$); this means that $\mathcal{K}_{4.1}$ is a preferred result (over $\mathcal{K}_{4.2}$), as $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) <_{\mathcal{K}_0} \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$. The ordering between ground facts that allows this kind of comparison is denoted by $<_G$. For a more formal and detailed presentation of the ordering, we refer the reader to [11].

We denote the evolution operation by $\bullet$. In our example, we get $\mathcal{K}_0 \bullet \mathcal{C} = \mathcal{K}_{4.1}$. Note that $\mathcal{K}_0 \bullet \mathcal{C}$ results from applying the change, $\mathcal{C}$, and its most preferable side-effects upon $\mathcal{K}_0$.

3 Answer Set Programming (ASP)

In what follows, we rely on the input language of the ASP grounder *gringo* [9] (extending the language of *lparse* [18]) and introduce only informally the basics of ASP. A comprehensive, formal introduction to ASP can be found in [2].

We consider extended logic programs as introduced in [17]. A *rule* r is of the following form:

$$h \leftarrow b_1, \ldots, b_m, \sim b_{m+1}, \ldots, \sim b_n.$$

By $head(r) = h$ and $body(r) = \{b_1, \ldots, b_m, \sim b_{m+1}, \ldots, \sim b_n\}$, we denote the *head* and the *body* of r, respectively, where "$\sim$" stands for default negation. The head H is an atom a belonging to some alphabet $\mathcal{A}$, the falsum $\bot$, or a cardinality constraint $L\,\{\ell_1, \ldots, \ell_k\}\,U$. In the latter, $\ell_i = a_i$ or $\ell_i = \sim a_i$ is a *literal* for $a_i \in \mathcal{A}$ and $1 \leq i \leq k$; L and U are integers providing a lower and an upper bound. Such a constraint is true if the number of its satisfied literals is between L and M. Either or both of L and U can be omitted, in which case they are identified with the (trivial) bounds 0 and ∞, respectively. A rule r such that $head(r) = \bot$ is an *integrity constraint*; one with a cardinality constraint as head is called a *choice rule*. Each body component B_i is either an atom or a cardinality constraint for $1 \leq i \leq n$. If $body(r) = \emptyset$, r is called a *fact*, and we skip "$\leftarrow$" when writing facts below. In addition to

rules, a logic program can contain `#minimize` statements of the form

$$\#\texttt{minimize}[\ell_1 = w_1@L_1, \dots, \ell_k = w_k@L_k].$$

Besides literals ℓ_j and integer weights w_j for $1 \leq j \leq k$, a `#minimize` statement includes integers L_j providing priority levels. A `#minimize` statement distinguishes optimal answer sets of a program as the ones yielding the smallest weighted sum for the true literals among $\ell_1, \dots, \ell_k$ sharing the same (highest) level of priority L, while for $L' > L$ the sum equals that of other answer sets. For a formal introduction, we refer the interested reader to [17], where the definition of answer sets for logic programs containing extended constructs (cardinality constraints and minimize statements) under "choice semantics" is defined.

Likewise, first-order representations, commonly used to encode problems in ASP, are only informally introduced. In fact, *gringo* requires programs to be *safe*, that is, each variable must occur in a positive body literal. Formally, we only rely on the function *ground* to denote the set of all ground instances, $\mathit{ground}(\Pi)$, of a program Π containing first-order variables. Further language constructs of interest, include conditional literals, like "$a\!:\!b$", the range and pooling operator ".." and ";" as well as standard arithmetic operations. The ":" connective expands to the list of all instances of its left-hand side such that corresponding instances of literals on the right-hand side hold [18, 9]. While ".." allows for specifying integer intervals, ";" allows for pooling alternative terms to be used as arguments within an atom. For instance, $p(1..3)$ as well as $p(1;2;3)$ stand for the three facts $p(1)$, $p(2)$, and $p(3)$. Given this, $q(X)\!:\!p(X)$ results in $q(1), q(2), q(3)$. See [9] for detailed descriptions of the input language of the grounder *gringo*.

4 Evolution using ASP

4.1 Potential Side-Effects

In order to determine the result of updating a KB, we need to determine the side-effects that would resolve any possible validity problems caused by the change. The general idea is simple: since the original KB is valid, a change causes a violation if it adds/removes a fact that renders some constraint invalid. Let us denote by ∇ the set of potential side effects of a change $\mathcal{C}$. Given a set of facts $\mathcal{C}$, we will write $\mathcal{C}^+/\mathcal{C}^-$ to denote the positive/negative facts of $\mathcal{C}$ respectively. First of all, we note that ∇ will contain all facts in $\mathcal{C}$, except those already implied by $\mathcal{K}$, i.e., if $p(\vec{x}) \in \mathcal{C}^+$ and $p(\vec{x}) \notin \mathcal{K}$, then $p(\vec{x}) \in \nabla^+$, and if $\neg p(\vec{x}) \in \mathcal{C}^-$ and $p(\vec{x}) \in \mathcal{K}$ then $\neg p(\vec{x}) \in \nabla^-$ (Condition I). The facts in the set $\nabla^+ \cup \mathcal{K}$ are called *available*. This initial set of effects may cause a constraint violation. Note that a constraint r is violated during a change iff the right-hand-side (rhs) of r becomes true and the left-hand-side (lhs) is made false. Thus, if a potential addition ∇^+ makes the rhs of r true, and lhs is false, then we have to add some fact from the lhs of the implication to the potential positive side-effects (to make lhs true) (Condition II), *or* remove some fact from rhs (to make it false) (Condition III). If a removal in ∇^- makes the lhs of r false, and all other facts in rhs are available (so rhs is true), we have to remove some fact from rhs (to make it false) (Condition IV). To do that, we first define a select function $s_i(X) = X \setminus \{X_i\}$ on a set X of *atoms*, to remove exactly one element of a set. So we can then refer to the element X_i and the rest of the set $s_i(X)$ separately. Abusing notation, we write $\mathit{pred}(p, \vec{U})$ for $\mathit{pred}(p_1, \vec{U}), \dots, \mathit{pred}(p_n, \vec{U})$, for any predicate name *pred* where p is the set of atoms $p_1(\vec{U}), \dots, p_{body}(\vec{U})$.

Formally, a set ∇ is a *set of potential side-effects* for a KB $\mathcal{K}$ and a change $\mathcal{C}$, if the following *conditions* are all true:

Table 4 Instance from example in Fig. 1

	$kb(ci,(x)).$	
$kb(cs,(a)).$	$kb(cs,(b)).$	$kb(cs,(c)).$
$kb(c_IsA,(b,c)).$	$kb(c_IsA,(b,a)).$	$kb(c_IsA,(c,a)).$
$kb(c_Inst,(x,b)).$	$kb(c_Inst,(x,c)).$	$kb(c_Inst,(x,a)).$
	$changeAdd(c_IsA,(a,b)).$	

I $x \in \nabla$ if $x \in \mathcal{C}^+$ and $x \notin \mathcal{K}$ *or* $x \in \mathcal{C}^-$ and $\neg x \in \mathcal{K}$,
II $\forall \vec{V}_h\ q_h(\vec{U},\vec{V}_h) \in \nabla^+$ if $s_l(p(\vec{U})) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \nabla^+$ and $q_h(\vec{U},\vec{V}_h) \notin \mathcal{K}$
III $\neg p_j(\vec{U}) \in \nabla^-$ if $s_j(s_l(p(\vec{U}))) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \nabla^+$ and $p_j(\vec{U}) \in \mathcal{K}$ and for all $\vec{V}_h$ either $\neg q_h(\vec{U},\vec{V}_h) \in \nabla^-$ or $q_h(\vec{U},\vec{V}_h) \notin \mathcal{K}$
IV $\neg p_l(\vec{U}) \in \nabla^-$ if $s_l(p(\vec{U})) \subseteq \nabla^+ \cup \mathcal{K}$ and $p_l(\vec{U}) \in \mathcal{K}$ and $\forall \vec{V}_h\ \neg q_h(\vec{U},\vec{V}_h) \in \nabla^-$,

for each constraint r defined in Section 2.3 and for all variable substitutions for $\vec{U}$ wrt $E(\vec{U})$ and for all $1 \leq l,j \leq body$, $l \neq j$, $1 \leq h \leq head$.

Our goal is to find a $\subset$-minimal set of potential side-effects ∇. We do this using the grounder *gringo*, which ground-instantiates a logic program with variables. We create a logic program where the single solution is the subset minimal set of potential side-effects ∇.

To build a logic program, we first have to define the inputs to the problem, called *instance*. An instance $\mathcal{I}(\mathcal{K},\mathcal{C})$ of a KB $\mathcal{K}$ and a change $\mathcal{C}$ is defined as a set of facts

$$\begin{aligned}\mathcal{I}(\mathcal{K},\mathcal{C}) = &\{kb(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{K}\}\\ &\cup \{changeAdd(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{C}^+\}\\ &\cup \{changeDel(p,\vec{x}) \mid p(\vec{x}) \in \mathcal{C}^-\}.\end{aligned}$$

In the above instance, predicate *kb* contains the facts in the KB, whereas predicates *changeAdd*, *changeDel* contain the facts that the change dictates to add/delete respectively. Note that this representation forms a twist from the standard representation, since a ground fact $p(\vec{x}) \in \mathcal{K}$ is represented as $kb(p,\vec{x})$ (same for the change). The representation of the KB $\mathcal{K}$ in Fig. 1 and its demanded change $\mathcal{C}$ can be found in Table 4.

Furthermore we have to collect all resources available in the KB (1) or newly introduced by the change (2). So the predicate *dom* associates a resource to its type,

$$dom(type(X_i),X_i) \leftarrow kb(T,\vec{X}). \tag{1}$$

$$dom(type(X_i),X_i) \leftarrow changeAdd(T,\vec{X}). \tag{2}$$

for all $X_i \in \vec{X}$. The following two rules ((3) and (4)) correspond to Condition I above, stating that the effects of $\mathcal{C}$ should be in ∇ (unless already in $\mathcal{K}$). The predicates *pAdd* and *pDelete* are used to represent potential side effects (additions and deletions respectively), i.e., facts in the sets ∇^+, ∇^-.

$$pDelete(T,\vec{X}) \leftarrow changeDel(T,\vec{X}), kb(T,\vec{X}). \tag{3}$$

$$pAdd(T,\vec{X}) \leftarrow changeAdd(T,\vec{X}), {\sim}kb(T,\vec{X}). \tag{4}$$

To find those facts that are added due to subsequent violations, we define, for the set $\nabla^+ \cup \mathcal{K}$, the predicate *avail* in (5) and (6). For negative potential side-effects ∇^- we use a predicate

Table 5 Potential side-effects of example in Fig. 1

$c_IsA(a,b)$	$c_IsA(c,c)$	$c_IsA(c,b)$
$c_IsA(b,b)$	$c_IsA(a,a)$	$c_IsA(a,c)$
$\neg c_IsA(b,a)$	$\neg c_IsA(b,c)$	$\neg c_IsA(c,a)$

nAvail (7).

$$avail(T,\vec{X}) \leftarrow kb(T,\vec{X}). \tag{5}$$

$$avail(T,\vec{X}) \leftarrow pAdd(T,\vec{X}). \tag{6}$$

$$nAvail(T,\vec{X}) \leftarrow pDelete(T,\vec{X}). \tag{7}$$

At a next step, we need to include the ontological constraints $\mathcal{R}$ into our ASP program, by creating the corresponding ASP rules. Unlike standard ontological constraints which determine whether there is an invalidity, the ASP rules are used to determine how to handle an invalidity. So now consider a constraint $r \in \mathcal{R}$ as defined in Section 2.3. For r, we define a set of rules ((8)) that produce the set of potential side-effects according to Condition II.

$$pAdd(q_h,(\vec{U},\vec{V}_h)) \leftarrow e(\vec{U}), avail(s_l(p),\vec{U}), pAdd(p_l,\vec{U}), \sim kb(q_h,(\vec{U},\vec{V}_h)), dom(type(\vec{V}_h),\vec{V}_h). \tag{8}$$

for all $1 \leq l \leq body$ and $1 \leq h \leq head$. Similarly, to capture Condition III, we need two sets of rules ((9) and (10)), since we do not want to do this only for negative side-effects *nAvail* on the lhs of the rule, but also for facts that are not in the KB $\mathcal{K}$,

$$pDelete(p_j,\vec{u}) \leftarrow e(\vec{U}), avail(s_j(s_l(p)),\vec{U}), pAdd(p_l,\vec{U}), kb(p_j,\vec{U}), nAvail(q_h,(\vec{U},\vec{V}_h)) : dom(type(\vec{V}_h),\vec{V}_h). \tag{9}$$

$$pDelete(p_j,\vec{U}) \leftarrow e(\vec{U}), avail(s_j(s_l(p)),\vec{U}), pAdd(p_l,\vec{U}), kb(p_j,\vec{U}), \sim kb(q_h,(\vec{U},\vec{V}_h)) : dom(type(\vec{V}_h),\vec{V}_h). \tag{10}$$

for all $1 \leq l,j \leq body$, $l \neq j$, $1 \leq h \leq head$. The last Condition IV can be expressed by the following rule set (11)

$$pDelete(p_l,\vec{U}) \leftarrow e(\vec{U}), avail(s_l(p),\vec{U}), kb(p_l,\vec{U}), pDelete(q_h,\vec{U},\vec{V}_h) : dom(type(\vec{V}_h),\vec{V}_h). \tag{11}$$

for all $1 \leq l \leq body$ and $1 \leq h \leq head$.

▶ Proposition 1. Given a KB $\mathcal{K}$ and a change $\mathcal{C}$, and let A be the unique answer set of the stratified logic program $ground(\mathcal{I}(\mathcal{K},\mathcal{C}) \cup \{(1)\dots(11)\})$, then $\nabla = \{p(\vec{x}) \mid pAdd(p,\vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid pDelete(p,\vec{x}) \in A\}$ is a subset minimal set of potential side-effects of the KB $\mathcal{K}$ and the change $\mathcal{C}$.

For our example in Fig. 1, this results in the set of potential side-effects in Table 5. Note that the potential side-effects contain all possible side-effects, including side-effects that will eventually not appear in any valid change.

4.2 Solving the Problem

Note that the set of potential side-effects computed above contains all options for evolving the KB. However, some of the potential changes in *pAdd*, *pDelete* are unnecessary; in our

running example, the preferred solution was $\{c_IsA(a,b), \neg c_IsA(b,a), \neg c_IsA(b,c)\}$ (see Section 2), whereas Table 5 contains many more facts.

To compute the actual side-effects (which is a subset of the side-effects in *pAdd*, *pDelete*), we use a generate and test approach. In particular, we use the predicate $add(p, \vec{x})$ and $delete(p', \vec{x}')$ to denote the set of side-effects $p(\vec{x}) \in \Delta(\mathcal{K}, \mathcal{K}')$ (respectively $\neg p'(\vec{x}') \in \Delta(\mathcal{K}, \mathcal{K}')$) and use choice rules to guess side-effects from *pAdd*, *pDelete* to *add*, *delete* respectively (see (12), (13) below).

$$\{add(T, \vec{X}) : pAdd(T, \vec{X})\}. \tag{12}$$

$$\{delete(T, \vec{X}) : pDelete(T, \vec{X})\}. \tag{13}$$

Our changed KB is expressed using predicate kb' and is created in (14) and (15) consisting of every entry from the original KB that was not deleted and every entry that was added.

$$kb'(T, \vec{X}) \leftarrow kb(T, \vec{X}), {\sim}delete(T, \vec{X}). \tag{14}$$

$$kb'(T, \vec{X}) \leftarrow add(T, \vec{X}). \tag{15}$$

Moreover, we have to ensure that required positive (negative) changes $\mathcal{C}$ are (not) in the new KB respectively (*Principle of Success*) ((16) and (17)).

$$\leftarrow changeAdd(T, \vec{X}), {\sim}kb'(T, \vec{X}). \tag{16}$$

$$\leftarrow changeDel(T, \vec{X}), kb'(T, \vec{X}). \tag{17}$$

To ensure the *Principle of Validity* we construct all constraints from the DEDs $\mathcal{R}$, using the following transformation for each $r \in \mathcal{R}$:

$$\leftarrow kb'(p, \vec{U}), {\sim}1\{kb'(q_i, (\vec{U}, \vec{V_i}) : dom(type(\vec{V_i}), \vec{V_i}))\}, e(\vec{U}). \tag{18}$$

for all $1 \leq i \leq head$. Rule (18) ensures that if the rhs of a constraint is true wrt to the new KB and the lhs if false, then the selected set of side-effects is no valid solution.

▶ Proposition 2. Given a KB $\mathcal{K}$, a change $\mathcal{C}$ and a set of potential side-effects ∇, we define a set of facts $\nabla' = \{pAdd(p, \vec{x}) \mid p(\vec{x}) \in \nabla\} \cup \{pDelete(p, \vec{x}) \mid \neg p(\vec{x}) \in \nabla\}$. Let A be the answer set of the logic program $ground(\mathcal{I}(\mathcal{K}, \mathcal{C}) \cup \nabla' \cup \{(12) \ldots (18)\})$, then $\Delta(\mathcal{K}, \mathcal{K}') = \{p(\vec{x}) \mid add(p, \vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid delete(p, \vec{x}) \in A\}$ is a valid change of KB $\mathcal{K}$.

4.3 Finding the Optimal Solution

The solutions contained in *add*, *delete* are all valid solutions, per the above proposition, but only one of them is optimal, per the Principle of Minimal Change. So, the solutions must be checked wrt to the ordering $<_{\mathcal{K}}$. We generate minimize statements for the criteria $<_{pred}$ and $<_G$ (see Section 3). Several minimize constraints can be combined and the order of the minimize statements is respected. As *gringo* allows hierarchical optimization statements, we can easily express the whole ordering $<_{\mathcal{K}}$ in a set of optimize statements $\mathcal{O}$.

▶ Proposition 3. Given a KB $\mathcal{K}$, a change $\mathcal{C}$ and a set of potential side-effects ∇, we define a set of facts $\nabla' = \{pAdd(p, \vec{x}) \mid p(\vec{x}) \in \nabla\} \cup \{pDelete(p, \vec{x}) \mid \neg p(\vec{x}) \in \nabla\}$. Let A be the answer set of the logic program $ground(\mathcal{I}(\mathcal{K}, \mathcal{C}) \cup \nabla' \cup \{(12) \ldots (18)\})$, which is minimal wrt the optimize statements $\mathcal{O}$ then $\Delta(\mathcal{K}, \mathcal{K}') = \{p(\vec{x}) \mid add(p, \vec{x}) \in A\} \cup \{\neg p(\vec{x}) \mid delete(p, \vec{x}) \in A\}$ is the unique valid minimal change of KB $\mathcal{K}$.

5 Refinements

In this section, we refine the above direct translation, in order to increase the efficiency of our logic program. Our first optimization attempts to reduce the size of the potential side-effects ∇, whereas the second takes advantage of deterministic consequences of certain side-effects to speed-up the process.

5.1 Incrementally Computing Side-Effects

As the set of potential side-effects directly corresponds to the search space for the problem (see (12), (13) in Section 4), we could improve performance if a partial set of potential side-effects that contains the minimal solution was found, instead of the full set. According to the ordering of the solutions $<_{pred}$, a set of side-effects that does not contain any fact with a level greater than k is "better" than a solution that does. Thus, we split the computation of the possible side-effects into different parts, one for each level of $<_{pred}$ optimization. We start the computation of possible side-effects with $k = 1$, only adding facts of level 1 to repair our KB. If with this subset of possible side-effects no solution to the problem can be found, we increase k by one and continue the computation, reusing the already computed part of the potential side-effects. For grounding, this means we only want to have the possibility to find potential side-effects $p(\vec{x})$ of a level less than or equal to k. The corresponding ASP rules can be found in the extended version of this paper [15] and are denoted by I.

We define the operator $\mathcal{T}(\mathcal{K}, \mathcal{C}, k)$, as $\mathcal{T}(\mathcal{K}, \mathcal{C}, 0) = \mathit{ground}(\mathcal{I}(\mathcal{K}, \mathcal{C}))$ and $\mathcal{T}(\mathcal{K}, \mathcal{C}, k)$ where $k > 0$ is the set of facts of the unique answer set of the logic program $\mathit{ground}(\mathcal{T}(\mathcal{K}, \mathcal{C}, k - 1) \cup \{I\})$. $\mathcal{T}(\mathcal{K}, \mathcal{C}, n)$ produces a subset of the potential side-effects only using repairs up to level n. Given our example in Fig. 1, $\mathcal{T}(\mathcal{K}, \mathcal{C}, 7)$ gives us the first two rows of Table 5.

5.2 Exploiting Deterministic Side-Effects

A second way to improve performance is to consider deterministic side-effects of the original changes. As an example of a deterministic side-effect, suppose that the original change includes the deletion of a class a (corresponding to the side-effect $\neg cs(a)$). Then, per rule R5 (cf. Table 2), all class subsumptions that involve a must be deleted as well (corresponding to the side-effect $\neg c_IsA$). Therefore, the latter side-effect(s) are a necessary (deterministic) consequence of the former, so they can be added to the set of side-effects right from the beginning (at level 1). For the detailed logic program we refer to [15]. In this way we extend our change by deterministic consequences, to possibly reduce the number of incremental steps. For our example in Fig. 1 this results in the additionally required $\mathit{changeDel}(c_IsA, (b, a))$.

6 Experiments

We experimented with two real-world ontologies of different size and structure, namely GO [7] and CIDOC [6] (∼458.000/∼1.500 facts). GO's emphasis is on classes, whereas CIDOC contains many properties. To generate the changes, we took each ontology $\mathcal{K}$, randomly selected 6 facts $I \subseteq \mathcal{K}$, and deleted I from $\mathcal{K}$, resulting in a valid KB $\mathcal{K}'$. We then created our "pool of changes", $I_{\mathcal{C}}$, which contains 6 randomly selected facts from $\mathcal{K}'$ (deletions) and the 6 facts from I (additions). The change $\mathcal{C}$ was a random selection of n facts from $I_{\mathcal{C}}$ ($1 \leq n \leq 6$). Our experiment measured the time required to apply $\mathcal{C}$ upon $\mathcal{K}'$. The above process was repeated 100 times for each n ($1 \leq n \leq 6$). The benchmark was run on a machine with 4×4 CPUs, 3.4Ghz each and was restricted to 4 GB of RAM. Our implementation uses

Table 6 (a) GO benchmark

n	times	level	timeouts
1	123.3	2.37	0
2	243.7	4.72	0
3	454.6	8.50	0
4	619.0	11.94	0
5	711.1	13.44	2
6	756.1	14.27	6

(b) CIDOC benchmark

n	times	level	timeouts
1	2.2	10.70	0
2	3.3	16.28	20
3	3.4	16.15	30
4	7.0	16.96	51
5	3.5	16.19	66
6	7.3	18.00	76

*gringo*3.0.4 and *clasp*2.0.0RC1. A timeout of 3600 seconds was imposed on each run. Table 6 contains the results of our experiments in GO and CIDOC respectively. Each row in the table contains the experiments for one value of n (size of $\mathcal{C}$) and shows the average CPU time (in seconds) of all runs that did not reach the timeout (column "times"), the average level of incremental grounding where the solution was found ("level") and the number of timeouts ("timeouts").

The results of our experiments are encouraging. GO, despite its large size and the intractable nature of the evolution problem, can evolve in a decent amount of time, and has very few timeouts. On the other hand, CIDOC has lots of timeouts, but very fast execution when no timeout occurs. This indicates that the deviation of execution times, even for KBs/changes of the same size, is very large for CIDOC, i.e., the performance is largely affected by the morphology of the input. This behaviour is much less apparent in GO, and is caused by the existence of many properties in CIDOC. Any violated property-related constraint greatly increases the number of potential side-effects. Thus, for updates causing many property-related violations, the execution time increases, often causing timeouts. Given that GO contains no properties, the execution times are more smooth. Another observation is that there is a strong correlation between the level, the average time reported and the size of the change.

7 Summary and Outlook

We studied the problem of ontology evolution in the presence of ontological constraints. Based on the setting and solution proposed in [11], we recast the problem and reduced it to an ASP program that can be solved by an optimized ASP reasoner. Given that the problem is inherently exponential in nature [11], the reported times (Table 6) for the evolution of two real-world ontologies (GO/CIDOC) are decent. To the best of our knowledge, there is no comparable approach, because the approach presented in [11] did not report any experiments, and other similar approaches either do not consider the full set of options (therefore returning a suboptimal evolution result), or require user feedback. An interesting side-product of our approach is that we can *repair ontologies* by simply applying the empty change upon them; we plan to explore this idea as a future work. We will also consider additional optimizations using incremental ASP solvers such as *iclingo* [10].

References

1. C. E. Alchourron, P. Gardenfors, and D. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50:510–530, 1985.
2. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving.* Cambridge University Press, 2003.

3 Tim Berners-Lee, James A. Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, 2001.

4 D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210`, 2004.

5 A. Cali, G. Gottlob, and T. Lukasiewicz. Datalog±: A unified approach to ontologies and integrity constraints. In *Proceedings of the International Conference on Database Theory (ICDT-09)*, 2009.

6 CIDOC. *The CIDOC Conceptual Reference Model*, 2010. `cidoc.ics.forth.gr/official_release_cidoc.html`.

7 The Gene Ontology Consortium. Gene ontology: tool for the unification of biology. In *Nature genetics*, volume 25, pages 25–29, 2000.

8 Alin Deutsch. Fol modeling of integrity constraints (dependencies). *Encyclopedia of Database Systems*, pages 1155–1161, 2009.

9 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`.

10 M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In M. Garcia de la Banda and E. Pontelli, editors, *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*, pages 190–205. Springer Verlag, 2008.

11 George Konstantinidis, Giorgos Flouris, Grigoris Antoniou, and Vassilis Christophides. A formal approach for rdf/s ontology evolution. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08)*, pages 405–409, 2008.

12 G. Lausen, M. Meier, and M. Schmidt. Sparqling constraints for rdf. In *Proceedings of 11th International Conference on Extending Database Technology (EDBT-08)*, pages 499–509, 2008.

13 B. McBride, F. Manola, and E. Miller. Rdf primer. `www.w3.org/TR/rdf-primer`, 2004.

14 B. Motik, I. Horrocks, and U. Sattler. Bridging the gap between owl and relational databases. In *Proceedings of 17th International World Wide Web Conference (WWW-07)*, pages 807–816, 2007.

15 M. Ostrowski, G. Flouris, T. Schaub, and G. Antoniou. Evolution of ontologies using ASP. Technical Report TR-415, FORTH-ICS, 2011.

16 G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *Proceedings of the 4th International Semantic Web Conference (ISWC-05)*, 2005.

17 P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

18 T. Syrjänen. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

19 J. Tao, E. Sirin, J. Bao, and D.L. McGuinness. Extending owl with integrity constraints. In *Proceedings of the 23rd International Workshop on Description Logics (DL-10)*. CEUR-WS 573, 2010.