Evolution of Ontologies using ASP

Max Ostrowski[1] and Giorgos Flouris[2] and Torsten Schaub[1] and Grigoris Antoniou[2,3]
[1]*Universität Potsdam* [2]*FORTH-ICS* [3]*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.

1 Introduction

The Semantic Web's vision, originally proposed in (Berners-Lee et al. 2001), is an extension of the current web that provides a common framework, including tools, languages and methodologies, to allow information on the web to be both understandable by humans and processable by machines; this enables people and machines to work in cooperation and process meaning and information on the web. 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 (McBride et al. 2004; Brickley and Guha 2004) 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 (Serfiotis et al. 2005; Motik et al. 2007; Lausen et al. 2008; Cali et al. 2009; Tao et al. 2010) 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 declarative formalisms for representing constraints on top of RDF/S data. In this paper, we will consider DED constraints (Deutsch 2009), which form a subset of first-order logic and have been shown to allow the representation of many useful types of constraints on ontologies; 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. RDF/S KBs, being representations of the real world, are often subject to change for various reasons, including changes in the modeled world, new information on the domain, newly-gained access to information previously unknown or classified, and other eventualities (Umbrich et al. 2010; Stojanovic et al. 2002; Flouris et al. 2008). Therefore, an important task towards the realization of the Semantic Web is the introduction of techniques that allow the efficient and intuitive changing of ontologies in the presence of constraints. Given the constraints, one should make sure that the evolution result is valid, i.e., it does not violate any constraints. This is often called the *Principle of Validity* (Alchourron et al. 1985). In addition, the *Principle of Success* (Alchourron et al. 1985) 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* (Alchourron et al. 1985), 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 (Konstantinidis et al. 2008), 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 (Konstantinidis et al. 2008) regarding the problem of ontology evolution in the presence of integrity 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 algorithms. Using first-order logic programs is a smart way to represent the problem while remaining highly flexible, especially in the set of constraints related to the ontology.

The objective of the present work is to recast the problem of ontology evolution 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 (Gebser et al.). Our work is based on the approach presented in (Konstantinidis et al. 2008), 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 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 describe the problem of ontology evolution and briefly present the solution proposed in (Konstantinidis et al. 2008). In Section 3, we present ASP. Section 4 is the main section of this paper, 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 (McBride et al. 2004; Brickley and Guha 2004) 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 (representing collections of resources), properties (which are binary relations between resources), and individuals (which are resources). We use the symbol $type(u)$ to denote the type of a resource u, stating whether it is a class, a property, etc. RDF/S also allows for the description of various predefined relations between resources, like the domain and range of properties, subsumption relationships between classes and between properties, and instantiation relationships between individuals and classes, or between pairs of individuals and properties. RDF/S KBs are commonly represented as labeled graphs, whose nodes are resources and edges are relations (see Fig. 1 for an example). In that figure, a, b, and c are classes, whereas 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

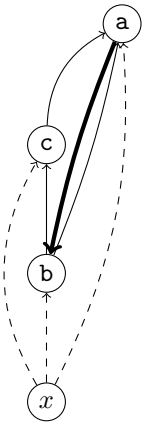

Fig. 1: A knowledge base with change (change appearing as bold arrow)

As explained above, 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 often called the Principle of Validity (Alchourron et al. 1985). 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* (Alchourron et al. 1985), 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* (Alchourron et al. 1985), stating 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. In this work, we determine the notion of "closeness" that is used in the Principle of Minimal Change using a relation; the function and use of this relation is described in detail later.

2.3 Formal Setting

To address the problem of ontology evolution, we use the general approach presented in (Konstantinidis et al. 2008). In that work, 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 name and $\vec{x}$ is a vector of constants. Constants represent resources in RDF/S parlance, and for each type of relation-

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	$c\_Inst(x, c)$

Table 1. *Representation of RDF/S Triples Using Predicates*

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 2. *Ontological Constraints*

ship that can appear in an RDF/S KB, a different predicate is defined. For example, the triple $(a, rdfs{:}subClassOf, b)$, 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, while variables will start with an upper case letter. Table 1 shows some of the predicates we use and their intuitive meaning (see (Konstantinidis et al. 2008) for a complete list).

We assume closed world, i.e., we infer that $\mathcal{K} \nvdash p(\vec{x})$ whenever $p(\vec{x}) \notin \mathcal{K}$. A *change* $\mathcal{C}$ is a request to add/remove some facts to/from the KB. For practical purposes, a change is modeled as a set of (possibly negated) ground facts, where positive ground facts correspond to additions, and negated ones correspond to deletions.

Ontological constraints are modeled using DED rules (Deutsch 2009), which allow for formulating various useful constraints, such as primary and foreign key constraints (used, e.g., in (Lausen et al. 2008)), acyclicity and transitivity constraints for properties (as in (Serfiotis et al. 2005)), and cardinality constraints (used in (Motik et al. 2007)). 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 (Konstantinidis et al. 2008). 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

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

Table 3. *Facts from example in Fig. 1*

$c\_IsA(a,b)$ 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, $\mathcal{C}$ is not a valid change, because $\mathcal{K}_1$ violates rule $R13$; thus, it cannot be returned as an evolution result. 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). This is an extra modification, that is not part of the original change, but is, in a sense, enforced by it and the related principles; such extra modifications are called *side-effects*. After applying this side-effect, we would get $\mathcal{K}_2 = \mathcal{K}_0 \cup \{c\_IsA(a,b)\} \setminus \{c\_IsA(b,a)\}$.

We note that $\mathcal{K}_2$ is no good either, because $R12$ is violated. To resolve this, we need to apply more side-effects. In that case, we have two options, either removing $c\_IsA(b,c)$ or $c\_IsA(c,a)$. This leads to two alternative solutions, namely $\mathcal{K}_{3.1} = \mathcal{K}_0 \cup \{c\_IsA(a,b)\} \setminus \{c\_IsA(b,a), c\_IsA(b,c)\}$ and $\mathcal{K}_{3.2} = \mathcal{K}_0 \cup \{c\_IsA(a,b)\} \setminus \{c\_IsA(b,a), c\_IsA(c,a)\}$. Again, adding $c\_IsA(b,a)$ is not an option, because its addition was dictated in a previous step.

Once again $\mathcal{K}_{3.1}, \mathcal{K}_{3.2}$ are not valid, so one more step is required: for $\mathcal{K}_{3.1}$, $R12$ is violated (because $c\_IsA(c,a), c\_IsA(a,b) \in \mathcal{K}_{3.1}$ but $c\_IsA(c,b) \notin \mathcal{K}_{3.1}$), so we add $c\_IsA(c,b)$ (the other options are overruled from previous steps) resulting to $\mathcal{K}_{4.1}$; for $\mathcal{K}_{3.2}$, $R12$ is violated again (because $c\_IsA(a,b), c\_IsA(b,c) \in \mathcal{K}_{3.2}$ but $c\_IsA(a,c) \notin \mathcal{K}_{3.2}$), so we add $c\_IsA(a,c)$ (the other options are overruled from previous steps) resulting to $\mathcal{K}_{4.2}$.

Now $\mathcal{K}_{4.1}, \mathcal{K}_{4.2}$ are both possible results for the evolution, as they satisfy the principles of success and validity. It remains to determine which of the two is "preferable", in the sense of the principle of minimal change, i.e., which of the two is "closer" to $\mathcal{K}_0$.

As explained above, to determine this, we need to develop an ordering that would "rank" $\mathcal{K}_{4.1}, \mathcal{K}_{4.2}$ in terms of "closeness" to $\mathcal{K}_0$. Note that $\mathcal{K}_{4.1}$ differs from $\mathcal{K}_0$ in having added $c\_IsA(a,b), c\_IsA(c,b)$ and deleted $c\_IsA(b,a), c\_IsA(b,c)$, whereas $\mathcal{K}_{4.2}$ differs from $\mathcal{K}_0$ in having added $c\_IsA(a,b), c\_IsA(a,c)$ and deleted $c\_IsA(b,a), c\_IsA(c,a)$. We express this by using difference sets, called *deltas*, containing the ground facts that need to be added and the (negated) ground facts that need to be deleted to get from one KB to another (denoted by $\Delta(\mathcal{K}, \mathcal{K}')$). In our example, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) = \{c\_IsA(a,b), c\_IsA(c,b), \neg c\_IsA(b,a), \neg c\_IsA(b,c)\}$, $\Delta(\mathcal{K}_0, \mathcal{K}_{4.2}) = \{c\_IsA(a,b), c\_IsA(a,c), \neg c\_IsA(b,a), \neg c\_IsA(c,a)\}$. Thus, the "ranking of closeness" is essentially reduced to ranking $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}), \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$; note that both deltas have the same size (and this occurs in many change scenarios), so ranking should be based on more subtle differences, like the severity of changes in each delta, not just their cardinality.

In (Konstantinidis et al. 2008), a specific intuitive ordering is described (denoted by $<_{\mathcal{K}_0}$, where $\mathcal{K}_0$ the original KB). In a nutshell, the available predicates are ordered in terms of severity using a special ordering, denoted by $<_{pred}$; 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, a delta Δ_1 is preferable than Δ_2 (i.e., $\Delta_1 <_{\mathcal{K}_0} \Delta_2$) 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 two deltas contain an equal number of ground facts per predicate (as in the case of our example with $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}), \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$), 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 $\mathcal{K}_{4.2}$), per the principle of minimal change. The ordering between ground facts that allows this kind of comparison is denoted by $<_G$. Formally: $\Delta(\mathcal{K}_0, \mathcal{K}_{4.1}) <_{\mathcal{K}_0} \Delta(\mathcal{K}_0, \mathcal{K}_{4.2})$; note that the subscript $\mathcal{K}_0$ in the ordering is important, because the position of constants in the hierarchy of the original KB ($\mathcal{K}_0$) is considered. For a more formal and detailed presentation of the ordering, we refer the reader to (Konstantinidis et al. 2008).

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* (Gebser et al.) (extending the language of *lparse* (Syrjänen)) and introduce only informally the basics of ASP. A comprehensive, formal introduction to ASP can be found in (Baral 2003).

We consider extended logic programs as introduced in (Simons et al. 2002). 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

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

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 $\ell_1, \ldots, \ell_k$ 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 (Simons et al. 2002), 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 vari-

able must occur in a positive body literal. Formally, we only rely on the function $ground$ to denote the set of all ground instances, $ground(\Pi)$, 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 (Syrjänen ; Gebser et al.). 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 (Gebser et al.) 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. To explain this in more detail, let us denote by ∇ the set of potential side effects of a change $\mathcal{C}$. In general, 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 $pred(p, \vec{U})$ for $pred(p_1, \vec{U}), \ldots, pred(p_n, \vec{U})$, for any predicate name $pred$ where p is the set of atoms $p_1(\vec{U}), \ldots, p_{body}(\vec{U})$.

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:

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 $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$.

	$kb(ci,(x))$.	
$kb(cs,(a))$.	$kb(cs,(b))$.	$kb(cs,(c))$.
$kb(c\_IsA,(b,c))$.	$kb(c\_IsA,(b,a))$.	$kb(c\_IsA,(c,a))$.
$kb(c\_Inst,(x,b))$.	$kb(c\_Inst,(x,c))$.	$kb(c\_Inst,(x,a))$.
	$changeAdd(c\_IsA,(a,b))$.	

Table 4. *Instance from example in Fig. 1*

Our goal is to find a $\subset$-minimal set of potential side-effects ∇, which means that there exists no set of potential side-effects ∇' such that $\nabla' \subset \nabla$. 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 redundant predicate $nAvail$ (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 ontological constraints which determine whether there is an invalidity, we use the ASP rules to determine how to handle this invalidity. So now consider a constraint $r \in \mathcal{R}$ as defined in Section 2.3. For r, we define a set

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

Table 5. *Potential Side-effects of example in Fig. 1*

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 $ground(\mathcal{I}(\mathcal{K},\mathcal{C}) \cup \{(1)\ldots(11)\})$, 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 and should be returned, per the Principle of Minimal Change. So, the solutions must be checked wrt to the ordering $<_{\mathcal{K}}$. For the most important criteria $<_{pred}$ we generate minimize statements (see Section 3) that minimize the number of occurrences of the applied side-effects. For adding classes, the corresponding minimize statement looks as follows:

$$minimize[kb'(cs, X)@L :{\sim}kb(cs, X)]. \tag{19}$$

Here L denotes the level of the minimize constraint. So several minimize constraints can be combined and the order of the minimize statements is respected. For the ordering $<_G$, we need to compare the distance of the entries (using their subsumption relation) to the top of the hierarchy. Even though this distance can be computed using a grounder like *gringo*, for simplicity we rely on a precomputed predicate $lowestDist(type(X), X, N)$ which gives the distance N for each entry X. The corresponding minimization statement is:

$$minimize[kb'(cs, X) = -N@L' : lowestDist(cs, X, N) :{\sim}kb(cs, X)]. \tag{20}$$

Here $-N$ denotes the weight of the literal in the statement. By using negative weights we do maximization instead of minimizing. We clearly want to maximize the distance to the top of the hierarchy. As these entities are less "general" and a change is therefore less severe. Again, $L' = L - 1$ denotes the level of the minimize constraint, so minimizing the number of additions is more important than maximizing the distance.

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 $<_{pred}$, we use the symbol $level(p)$ for each fact $p(\vec{x})$ to denote the severeness of its application (per $<_{pred}$). We start with the least severe impact (level 1), incrementally assigning higher numbers for more severe changes. For example, adding a class subsumption c_IsA is on level 7 ($level(c\_IsA) = 7$), removing such a subsumption has $level(\neg c\_IsA) = 15$ and adding a class (cs) has $level(cs) = 31$. For a complete list of such levels, see (Konstantinidis et al. 2008).

Per definition of the ordering, a set of side-effects that does not contain any fact $p(\vec{x})$ with $level(p) > k$ is "better" than a solution that uses at least one side-effect $p'(\vec{x'})$ such that $level(p') > k$. 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 $level(p) \leq k$. To exploit this idea, we associate each constraint r with a *level*, denoted by $level(r)$. The level of r is the maximum of the levels of the predicates q_h, p_j that appear in the lhs of the corresponding ASP rules of r, namely (8) and (9), (10), (11) respectively (see Section 4).

To express the corresponding ASP rules, we need some general rules. First, (21) stipulates the fact $step$ for the current level; as we move to subsequent levels, $step$ is updated to indicate the level we processed or are just processing. Rule (22) stipulates the fact $oldStep(l)$ for all already processed levels l.

$$step(k). \tag{21}$$

$$oldStep(k-1) \leftarrow k > 1. \tag{22}$$

To denote potential side-effects of a given level k, we use the predicates $newPadd(k, p, \vec{x})$ and $newPdel(k, p, \vec{x})$ (unlike $pAdd$, $pDelete$, which contain potential side-effects of all

levels). The ASP rules representing Condition I (originally expressed using (3) and (4) for $pAdd$, $pDelete$) are expressed for the new predicates as follows:

$$newPdel(k, T, \vec{X}) \leftarrow changeDel(T, \vec{X}), k = 1, kb(T, \vec{X}). \tag{23}$$
$$newPadd(k, T, \vec{X}) \leftarrow changeAdd(T, \vec{X}), k = 1, \sim kb(T, \vec{X}). \tag{24}$$

The predicates $pAdd$ and $pDelete$ represent the potential side-effects already found in all levels before k. Therefore the new additions from the last processed level are added to the $pAdd$ and $pDelete$ predicate ((25) to (26)), respectively.

$$pAdd(T, \vec{X}) \leftarrow newPadd(k-1, T, \vec{X}). \tag{25}$$
$$pDelete(T, \vec{X}) \leftarrow newPdel(k-1, T, \vec{X}). \tag{26}$$

The redundant predicates $nNewAvail$ and $newAvail$ contain everything that is newly available or not available at level k ((27) and (28)).

$$nNewAvail(k, T, \vec{X}) \leftarrow newPdel(k, T, \vec{X}). \tag{27}$$
$$newAvail(k, T, \vec{X}) \leftarrow newPadd(k, T, \vec{X}). \tag{28}$$

Finally, $nAvail$ and $avail$ "accumulate" the available/not available facts ((29) to (31)).

$$nAvail(T, \vec{x}) \leftarrow nNewAvail(k, T, \vec{x}). \tag{29}$$
$$avail(T, \vec{x}) \leftarrow kb(T, \vec{x}). \tag{30}$$
$$avail(T, \vec{x}) \leftarrow newAvail(k, T, \vec{x}). \tag{31}$$

For our *conditions* II, III and IV we can distinguish three different cases depending on the level of the corresponding ASP rules. The ASP rules for constraints r with

- $level(r) = k$ are grounded for all potential side-effects (new and old ones) and entities in the KB.
- $level(r) \leq k$, are only grounded if either of the potential side-effects on the rhs of the rule r is newly added on the level k.
- $level(r) < k$ are grounded if at least one of the potentially *available*/not *available* entries has been added at step k.

These three steps are now described in detail for each condition. For Condition II, the rule sets (32), (33), and (34) correspond to the set (8) (Section 4). The rule set (32) produces the corresponding set of potential side-effects for the current incremental step k,

$$\begin{aligned} newPadd(k, q_h, (\vec{U}, \vec{V_h})) \leftarrow & \sim pAdd(q_h, (\vec{U}, \vec{V_h})), avail(s_l(p), \vec{U}), pAdd(p_l, \vec{U}), \\ & e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V_h})), dom(type(\vec{V_h}), \vec{V_h}), \\ & step(level(add, q_h)), \sim step(level(add, q_h) + 1). \end{aligned} \tag{32}$$

for all $1 \leq l \leq body$, $1 \leq h \leq head$. For constraints r with $level(r) \leq k$, the following rule (33) captures Condition II and checks whether one of the potential side-effects on the rhs of the rule is newly added on the level k by using $newPadd$ in the body of the rule. We only add $step(level(r))$ to the rule to consider all levels,

$$\begin{aligned} newPadd(k, q_h, (\vec{U}, \vec{V_h})) \leftarrow & \sim pAdd(q_h, (\vec{U}, \vec{V_h})), avail(s_l(p), \vec{U}), \\ & newPadd(k, p_l, \vec{U}), e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V_h})), \\ & dom(type(\vec{V_h}), \vec{V_h}), step(level(add, q_h)). \end{aligned} \tag{33}$$

for all $1 \leq l \leq body$, $1 \leq h \leq head$. To capture Condition II for all constraints r with $level(r) < k$, the rule (34) checks if at least one of the potentially *available*/not *available*

entries has been added at step k. We add $oldStep(level(r))$ to the rules, to consider only $levels < k$,

$$\begin{aligned} newPadd(k, q_h, (\vec{U}, \vec{V}_h)) \leftarrow & \sim pAdd(q_h, (\vec{U}, \vec{V}_h)), avail(s_l(p), \vec{U}), pAdd(p_l, \vec{U}), \\ & e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V}_h)), dom(type(\vec{V}_h), \vec{v}_h), \\ & 1\{newAvail(k, s_l(p), \vec{U})\}, oldStep(level(add, q_h)). \end{aligned} \tag{34}$$

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

For Condition III, rules (35) to (40) correspond to the rules expressed in (9) and (10).

We use $step(level(r)), \sim step(level(r)+1)$ in the rule body, to denote that we only want to consider the last (current) incremental step k in rules (35) and (36),

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), pAdd(p_l, \vec{U}) \\ & kb(p_j, \vec{U}), e(\vec{U}), nAvail(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & step(level(delete, p_j)), \sim step(level(delete, p_j)+1). \end{aligned} \tag{35}$$

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), pAdd(p_l, \vec{U}), \\ & kb(p_j, \vec{U}), e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & step(level(delete, p_j)), \sim step(level(delete, p_j)+1). \end{aligned} \tag{36}$$

for all $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$. For constraints r with $level(r) \leq k$, the following rule set (37) and (38) capture Condition III and check whether one of the potential side-effects on the rhs of the rule is newly added on level k

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), newPadd(k, p_l, \vec{U}), \\ & kb(p_j, \vec{U}), e(\vec{U}), nAvail(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & step(level(delete, p_j)). \end{aligned} \tag{37}$$

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), newPadd(k, p_l, \vec{U}), \\ & kb(p_j, \vec{U}), e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & step(level(delete, p_j)). \end{aligned} \tag{38}$$

for all $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$. The rule sets (39) and (40) check if at least one of the potentially *available*/not *available* entries has been added at step k,

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), pAdd(p_l, \vec{U}), \\ & kb(p_j, \vec{U}), e(\vec{U}), nAvail(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & 1\{newAvail(k, s_j(s_l(p)), \vec{U}), \\ & \quad nNewAvail(k, q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h)\}, \\ & oldStep(level(delete, p_j)). \end{aligned} \tag{39}$$

$$\begin{aligned} newPdel(k, p_j, \vec{U}) \leftarrow & \sim pDelete(p_j, \vec{U}), avail(s_j(s_l(p)), \vec{U}), pAdd(p_l, \vec{U}), \\ & kb(p_j, \vec{U}), e(\vec{U}), \sim kb(q_h, (\vec{U}, \vec{V}_h)) : dom(type(\vec{V}_h), \vec{V}_h), \\ & 1\{newAvail(k, s_j(s_l(p)), \vec{U})\}, oldStep(level(delete, p_j)). \end{aligned} \tag{40}$$

for all $1 \leq l, j \leq body$, $l \neq j$, $1 \leq h \leq head$.

Condition IV is captured using the rules (41) to (43) that correspond to the rules (11) in Section 4. For the current incremental step k we use:

$$\begin{aligned} newPdel(k, p_l, \vec{U}) \leftarrow & \sim pDelete(p_l, \vec{U}), e(\vec{U}), avail(s_l(p), \vec{U}), kb(p_l, \vec{U}), \\ & pDelete(q_h, \vec{U}, \vec{V}_h) : dom(type(\vec{V}_h), \vec{V}_h), \\ & step(level(delete, p_l)), \sim step(level(delete, p_l)+1). \end{aligned} \tag{41}$$

for all $1 \leq l \leq body$, $1 \leq h \leq head$. For constraints r with $level(r) \leq k$, the rule (42)

$c\_IsA(a,b)$	$c\_IsA(c,c)$	$c\_IsA(c,b)$
$c\_IsA(b,b)$	$c\_IsA(a,a)$	$c\_IsA(a,c)$

Table 6. *Potential Side-effects of example in Fig. 1 to level* 7

checks whether one of the potential side-effects on the rhs of r is newly added on level k,

$$\begin{aligned} newPdel(k,p_l,\vec{U}) \leftarrow & \sim pDelete(p_l,\vec{U}), e(\vec{U}), avail(s_l(p),\vec{U}), kb(p_l,\vec{U}), \\ & newPdel(k,q_h,\vec{U},\vec{V_h}) : dom(type(\vec{V_h}),\vec{V_h}), \\ & step(level(delete,p_l)). \end{aligned} \tag{42}$$

for all $1 \leq l \leq body$, $1 \leq h \leq head$. To capture Condition IV for all constraints r with $level(r) < k$, the rule (34) checks if at least one of the potentially *available*/not *available* entries has been added at step k,

$$\begin{aligned} newPdel(k,p_l,\vec{U}) \leftarrow & \sim pDelete(p_l,\vec{U}), e(\vec{U}), avail(s_l(p),\vec{U}), kb(p_l,\vec{U}), \\ & pDelete(q_h,\vec{U},\vec{V_h}) : dom(type(\vec{V_h}),\vec{V_h}), \\ & 1\{newAvail(k,s_l(p),\vec{U})\}, oldStep(level(delete,p_l)). \end{aligned} \tag{43}$$

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

We define the operator $\mathcal{T}(\mathcal{K},\mathcal{C},k)$, as $\mathcal{T}(\mathcal{K},\mathcal{C},0) = ground(\mathcal{I}(\mathcal{K},\mathcal{C}))$ 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 $ground(\mathcal{T}(\mathcal{K},\mathcal{C},k-1) \cup \{(21),\ldots,(43)\})$. $\mathcal{T}(\mathcal{K},\mathcal{C},n)$ 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 set shown in Table 6 and $\mathcal{T}(\mathcal{K},\mathcal{C},15)$ gives us the same set of side-effects as already show in 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). Formally, for all DED constraints of the form

$$\forall\vec{U} \bigvee\nolimits_{i=1,\ldots,head} \exists\vec{V_i} q_i(\vec{U},\vec{V_i}) \leftarrow e(\vec{U}), p(\vec{U}),$$

where $p(\vec{U})$ is a single atom, we generate the following rules:

$$\begin{aligned} changeAdd(q_j,(\vec{U},\vec{V_j})) \leftarrow & e(\vec{U}), changeAdd(p,\vec{U}), \\ & \sim kb(q_j,(\vec{U},\vec{V_j})), dom(type(\vec{V_j}),\vec{V_j}). \end{aligned} \tag{44}$$

$$\begin{aligned} changeDel(p,\vec{U}) \leftarrow & e(\vec{U}), kb(p,\vec{U}), \\ & changeDel(q_j,\vec{U},\vec{V_j}) : dom(type(\vec{V_j}),\vec{V_j}). \end{aligned} \tag{45}$$

for all $1 \leq j \leq head$. 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 $changeDel(c\_IsA,(b,a))$.

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

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

Table 7. *(a) Go Benchmark* *(b) CIDOC Benchmark*

6 Experiments

For our experiments, we considered two real-world ontologies of different size and structure in order to study the effects of both the size and of the morphology of the ontology on the performance of our approach. The first considered ontology is Gene Ontology (GO (Consortium 2000)), a large ontology from the bio-informatics, consisting of roughly 458.000 entries (28.000 classes, 370.000 relations, and 60.000 instances). The second ontology is CIDOC (CIDOC 2010), a medium-sized ontology from the cultural domain, consisting of roughly 1500 entries (with about 250 properties, 80 classes and 700 relations). As we can see, GO's emphasis is on classes, while CIDOC contains many properties.

To generate the changes, we took the original knowledge base $\mathcal{K}$ (GO or CIDOC), randomly selected 6 entries $I \subseteq \mathcal{K}$ 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 entries from $\mathcal{K}'$ (as deletions) and the 6 entries in I (as additions). The change $\mathcal{C}$ for our benchmark was created by randomly selecting n entries 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 ($1 \leq n \leq 6$), giving a total of 600 runs. The benchmark was run on a machine with 4×4 CPU's, 3.4Ghz each and was restricted to 4 GB of RAM Our implementation uses a single threaded version of *gringo*3.0.4 and *clasp*2.0.0RC1. A timeout of 3600 seconds was imposed on the experiments, so that any run that reached this timeout was stopped.

Table 7 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 inherently intractable nature of the evolution problem, can evolve in a decent amount of time, and has very few timeouts. On the other hand, CIDOC exhibits a seemingly strange behaviour: it has lots of timeouts, but very fast execution (on average) 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 specific form of the KB or change. Given that this behaviour is much less apparent in GO, we conclude that it is due to the existence of many properties in CIDOC; in fact, it turns out that property-related constraints increase the number of potential side-effects, thereby increasing the computational effort required, but only if such rules are violated. As a result, for the updates that included many property-related changes the execution time increases, often causing timeouts, whereas for updates that include few property-related changes the

execution time is rather small. Given that GO contains no properties, the execution times are much more smooth (few timeouts, but worse average performance). If we concentrate on the average level where a solution was found, we see a strong correlation between the level and the average time reported. Also, we note that as the size of the change increases, so does the level, and the average required time.

7 Summary and Outlook

In this paper, we studied how we can solve the problem of ontology evolution in the presence of ontological constraints, using ASP rules. Based on the setting and solution proposed in (Konstantinidis et al. 2008), we recast the problem and the corresponding algorithm in terms of ASP rules, thereby reducing the problem to an ASP program that can be solved by an efficient and optimized ASP reasoner. This resulted to a flexible approach for changing ontologies.

Given that the problem is inherently exponential in nature (Konstantinidis et al. 2008), the reported times (Table 7) for the evolution of two real-world ontologies are decent. To the best of our knowledge, there is no comparable approach, because the approach presented in (Konstantinidis et al. 2008) 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 to perform the change (which is not the case in our implementation).

An interesting side-product of our approach is that the validity of an ontology can be checked and/or imposed by simply applying the empty change upon it, so we plan to explore this option in the future for *repairing ontologies*. In addition, we will consider further optimizations that will improve the efficiency of the proposed modeling, such as using incremental ASP solvers such as *iclingo* (Gebser et al. 2008). Finally, given that the application of many small changes is usually faster than the application of a single large one (because it leads to much smaller sets of potential side-effects), we plan to consider methodologies for decomposing changes into smaller, independent sets that can be applied in a sequence (rather than in a bulk manner), without jeopardizing the correctness and optimality of the result.

References

ALCHOURRON, C. E., GARDENFORS, P., AND MAKINSON, D. 1985. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic 50*, 510–530.

BARAL, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press.

BERNERS-LEE, T., HENDLER, J. A., AND LASSILA, O. 2001. The semantic web. *Scientific American 284,* 5, 34–43.

BRICKLEY, D. AND GUHA, R. 2004. Rdf vocabulary description language 1.0: Rdf schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210`.

CALI, A., GOTTLOB, G., AND LUKASIEWICZ, T. 2009. Datalog±: A unified approach to ontologies and integrity constraints. In *Proceedings of the International Conference on Database Theory (ICDT-09)*.

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

CONSORTIUM, T. G. O. 2000. Gene ontology: tool for the unification of biology. In *Nature genetics*. Vol. 25. 25–29.

DEUTSCH, A. 2009. Fol modeling of integrity constraints (dependencies). *Encyclopedia of Database Systems*, 1155–1161.

FLOURIS, G., MANAKANATAS, D., KONDYLAKIS, H., PLEXOUSAKIS, D., AND ANTONIOU, G. 2008. Ontology change: Classification and survey. *The Knowledge Engineering Review 23,* 2, 117–152.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. A user's guide to gringo, clasp, clingo, and iclingo. Available at http://potassco.sourceforge.net.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND THIELE, S. 2008. Engineering an incremental ASP solver. In *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, M. Garcia de la Banda and E. Pontelli, Eds. Lecture Notes in Computer Science, vol. 5366. Springer Verlag, 190–205.

KONSTANTINIDIS, G., FLOURIS, G., ANTONIOU, G., AND CHRISTOPHIDES, V. 2008. A formal approach for rdf/s ontology evolution. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08)*. 405–409.

LAUSEN, G., MEIER, M., AND SCHMIDT, M. 2008. Sparqling constraints for rdf. In *Proceedings of 11th International Conference on Extending Database Technology (EDBT-08)*. 499–509.

MCBRIDE, B., MANOLA, F., AND MILLER, E. 2004. Rdf primer. www.w3.org/TR/rdf-primer.

MOTIK, B., HORROCKS, I., AND SATTLER, U. 2007. Bridging the gap between owl and relational databases. In *Proceedings of 17th International World Wide Web Conference (WWW-07)*. 807–816.

SERFIOTIS, G., KOFFINA, I., CHRISTOPHIDES, V., AND TANNEN, V. 2005. Containment and minimization of rdf/s query patterns. In *Proceedings of the 4th International Semantic Web Conference (ISWC-05)*.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

STOJANOVIC, L., MAEDCHE, A., MOTIK, B., AND STOJANOVIC, N. 2002. User-driven ontology evolution management. In *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management (EKAW-02)*.

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

TAO, J., SIRIN, E., BAO, J., AND MCGUINNESS, D. 2010. Extending owl with integrity constraints. In *Proceedings of the 23rd International Workshop on Description Logics (DL-10)*. CEUR-WS 573.

UMBRICH, J., HAUSENBLAS, M., HOGAN, A., POLLERES, A., AND DECKER, S. 2010. Towards dataset dynamics: Change frequency of linked open data sources. In *Proceedings of the WWW2010 Workshop on Linked Data on the Web (LDOW2010)*.