Formalizing the Evolution Process

Giorgos Flouris[1], George Konstantinidis[1,2]

[1] Institute of Computer Science, FORTH
[2] Computer Science Department, University of Crete
{fgeo, gconstan}@ics.forth.gr

Abstract. This work focuses on identifying and formalizing the process underlying the development of an evolution algorithm. The main argument elaborated in this work is that the development of an algorithm dealing with the incorporation of new knowledge in a logical structure (such as an ontology) is based on a pattern consisting of discrete and well-defined steps which can be formalized, described and studied independently. After a short literature review on the current state-of-the-art in the field of ontology evolution, we describe the aforementioned pattern and propose a formalization of it; this allows us to develop a generic "algorithmic pattern" which can be made specific and be applied to different representation languages or application scenarios, and lead to specific ontology evolution algorithms applicable to specific contexts. We then apply our formalization to the problem of updating RDF-based ontologies and develop an algorithm for this case.

Introduction

The problem of change management has been attracting a lot of attention in various contexts. The field of *belief change* (also known as *belief revision*) addresses the problem of understanding the process of change in standard logics [8]. Recently, a number of works have been tackling the problem in other contexts [5], [11], [12]; one related research field is *ontology evolution*, which refers to the process of modifying an ontology in response to a change in the domain or its conceptualization [6].

Our original motivation for this work was the development of an algorithm for updating RDF [13] ontologies. This effort led to the realization that the process of change is based on a pattern shared by most change algorithms, even though this pattern is often hidden in their implementation; this similarity offers an opportunity for abstraction. We argue that change algorithms are different manifestations of the same underlying mechanism and that, by studying the process of change at a more fundamental level, we will be able to better understand the algorithms' properties. Such a study may also serve as an aid towards the development of new change algorithms.

This paper elaborates on this idea, by studying the problem of change at three different levels. The first level is very generic and discusses an approach that could be applicable in practically any update context. This approach results to a generic algorithmic pattern, a "meta-algorithm" to which practical change algorithms can be reduced. The second level is more implementation-oriented, in the sense that it gives us hints on how the meta-algorithm could be implemented. At that level, we develop a general-purpose algorithm which could, in principle, be applied to several different contexts, once some generic, application-dependent parameters are set, so it is also quite general. The third level is most specific and can be viewed as a proof of concept for the other levels. At that level, we describe, based on the above patterns, the implementation of an algorithm for updating RDF [13] ontologies; this algorithm can be viewed as an instantiation of the more general algorithmic patterns of the other levels.

Preliminaries

Belief Change Principles

The field of belief change is a mature field addressing the problem of change for standard logics. Most of the works in the field study the problem at an abstract level and have given several interesting results, including the formulation of a number of intuitive desiderata (update principles) for a change operation. More specifically, given a *Knowledge Base (KB)* and an update, the following principles were formulated [4]:

- Principle of Adequacy of Representation, which states that the representation of the result of the change should be the same as the KB in the input
- Principle of Irrelevance of Syntax, which states that it is the logical force (rather than the syntactic form) of the KB and the update that should be taken into account in deciding the result of an update
- Principle of Maintenance of Consistency, which states that the resulting KB should be consistent
- Principle of Primacy of New Information, which states that the update takes precedence over existing information
- Principle of Persistence of Prior Knowledge, which states that as much as possible from the existing knowledge should be retained in the result
- Principle of Fairness, which states that the update algorithm should be well-defined, deterministic and that its result should be reproducible and non-arbitrary

The above constraints have been largely adopted, even though there have also been attempts to relax them; these attempts have led to various interesting research fields, such as non-prioritized belief revision [9]. Another attempt to introduce generic principles for change resulted to a similar, but more formal set of constraints, the AGM postulates, and the most influential belief change theory, the AGM theory [1].

Ontology Evolution

The current state of the art in ontology evolution, as well as an extensive list of related tools can be found in [6]. Here, we will consider four different tools, namely, Protégé [14], OilEd [2], OntoStudio (formerly OntoEdit) [16] and KAON [7].

Protégé is an ontology editor, which has emerged as a popular tool for ontology design, creation, evolution and management. OilEd is also an ontology editor often used for ontology evolution. Unlike Protégé, OilEd is rather restrictive in the sense that it disallows changes that cause any invalidity in the ontology, rather than taking action against it; in addition, it supports less update operations than Protégé.

KAON and OntoStudio are examples of more specialized ontology evolution tools, where the user can set pre-defined evolution strategies that control how indirect changes (side-effects) will be determined, thus allowing the tool to perform some changes automatically. In this respect, OntoStudio provides more options for parameterization, but uses a more restricted model and supports less operations than KAON.

RDF and RDFS

In this paper, RDF [13] will be used as the language in which our generic algorithmic pattern will be applied as a proof of concept (at the third, most specific level of analysis). We will use RDF under RDFS semantics [3], as described and formalized in [15]. Briefly, knowledge in RDF is described using triples of the form *(subject, predicate, object)*, where *predicate* is used to assign some property with value *object* on the *subject*, e.g., (myCar, hasColor, Red). A vocabulary of classes and properties is also provided: classes are used to classify resources, whereas properties define relationships between resources. All resources are assigned a Universal Resource Identifier (URI), which is used for unambiguous reference to the resource.

RDFS defines a number of special constructs carrying useful semantics, such as the *rdfs:subclassOf* predicate, used to denote that some class is subsumed by another class. Per the RDFS specification the *rdfs:subclassOf* predicate is transitive [3], so we can use this semantics to infer new knowledge. In [15], the semantics of RDF/S were recast, adding typing information, validity and inference rules, as well as a formalization based on First-Order Logic (FOL) predicates.

The Meta-Algorithm (First Level)

Basic Notions and General Update Principles

In our effort to study the change management process in the most general manner possible (first level), we will assume that knowledge is stored into some generic KB K, which can be any set of elements taken from a particular pool of elements (the *language* L). A language is part of some *logic*, which equips the elements of the language with semantics. An *update* U is a request to add and/or remove certain elements of the language from the KB. Our desiderata for an update algorithm are inspired by the principles of [4] (and [1]), and are described below:

- *Principle of Algorithmic Adequacy*, which states that the update operation should be a well-defined and deterministic algorithm, whose output should be a KB. This corresponds to the Principles of Adequacy of Representation and Fairness [4].
- *Principle of Irrelevance of Syntax*, stating that the logical force (rather than the syntax) of the KB (and update) should be considered in deciding the update result.

- *Principle of Validity*, which states that the resulting KB should be valid per the underlying constraints. This principle overrides the Principle of Maintenance of Consistency [4], in the sense that the notion of validity is more general than the notion of consistency, and it can be used also to capture user-defined constraints.
- *Principle of Success*, which states that whatever is dictated by the update should be executed. This principle is equivalent to the Principle of Primacy of New Information [4], but the more common (e.g., [1], [8]) term "success" has been used.
- *Principle of Minimal Change*, which states that as much as possible of the knowledge in the original KB should be retained. This principle implies that we should minimize the number and the "severeness" of the changes actually executed. The Principle of Minimal Change is equivalent to the Principle of Persistence of Prior Knowledge [4], but the term "Minimal Change" has prevailed (see [1], [8]).

The intuitions behind the above principles are straightforward. The Principle of Algorithmic Adequacy guarantees that the update result is reproducible, and that the result of the update is a KB which can be queried (and also updated) using the same processes as the original KB. The Principle of Irrelevance of Syntax dictates that the updates are based on semantic, rather than syntactic, considerations. The Principle of Validity guarantees that the resulting KB satisfies the same invariants (validity constraints) as the original KB (so that the update algorithm respects the invariants that have been imposed by the logic's semantics and the application). The Principle of Success dictates that the update request takes precedence over other knowledge; the update algorithm cannot choose to ignore any part of the update request. Finally, the Principle of Minimal Change "protects" the knowledge already in the KB, by dictating that the changes caused by the update should be as few and as "mild" as possible. The latter principle is different from the others in the sense that it imposes some kind of optimization constraint: it states that some quantity (the information loss) should be minimized. In this respect, it implies some kind of preference among the different possible update results, so it actually dictates the expected result of the update.

Informal Description of the Update Mechanism

The above principles imply certain facts on the update mechanism. First, during the execution of an update, one needs to, at least, add and remove the requested (by the update) elements from the KB (Principle of Success). Simply applying the update upon the KB would rarely succeed to satisfy the principles though; in practice, there are two factors which make the problem non-trivial, namely validity and inference.

When a validity context is in place (as is the case in all interesting logics), the aforementioned direct update execution is (usually) overruled by the Principle of Validity, as there is no guarantee that the result will, in general, be valid.

Inference (also a critical part of any interesting logic) affects the result as well, because its presence implies that not only explicit information should be considered. For example, the removal of an element, even if seemingly successful, could violate the Principle of Success if said element is a consequence of the remaining elements. Moreover, the addition (or removal) of some element may result to an invalidity which is not directly visible, but can only be detected once inferred knowledge is considered. Another implication is related to the Principle of Irrelevance of Syntax which implies that two logically equivalent KBs (or updates) should give the same result in the update operation; thus, it is the *closure* of the KB (i.e., both the explicit and the implicit elements) that should be considered for the update result.

In all the above cases, we need to reconsider the original result by executing more changes (called *side-effects*) than the ones directly dictated by the update (called *effects*); these side-effects should be as few and as mild as possible, per the Principle of Minimal Change. Thus, the general process (meta-algorithm) that we could use in order to determine the result of an update is as follows:

1. Execute the original update upon the original KB by adding and removing elements as requested by the update
2. Check whether the result is a valid KB and whether the inference mechanism does not cause any problems with respect to the Principle of Success
3. If the conditions of step 2 are true, no further action is required, so there will be no side-effects; otherwise, we need to determine the mildest possible set of side-effects that, upon execution, would render the conditions of step 2 true
4. Apply the selected side-effects of step 3 upon the result of step 1

Note that the execution of the above steps depends on various parameters that define the setting (context) upon which the algorithm is executed. This setting involves the determination of the following:

1. Language, i.e., the set of elements of which the KB and the update are comprised
2. Allowable KBs and updates, as some KBs or updates may not be supported (e.g., we may disallow updating a non-valid KB, or disallow certain types of updates)
3. Logic, by defining the inference mechanism and the validity context of the logic

All these parameters determine the context upon which the algorithm is applied, so they are called the *context of application* of the algorithm. The algorithm itself is, in most part, straightforward. The direct execution of an update or its side-effects (steps 1 and 4 respectively) is trivial, and the verification of the conditions in step 2 are given by the definition of the context of application. The only difficult step of the algorithm is step 3, in which one should determine the mildest possible set of side-effects to apply; this step involves some kind of *selection mechanism* determining the mildest out of the various potential side-effects that would render the conditions of step 2 true (which are determined by the context of application). Thus, given a context of application, the expected result of a change algorithm is determined by the selection mechanism that identifies the "best" out of the potential side-effects.

Formalizing the Setting: The Context of Application

The context of application determines the setting upon which the algorithm is applied; its basic ingredient is the language L, i.e., the set of elements which are used to form KBs and updates. In our context, L can be any non-empty set of elements. Given L, a KB can be easily formalized as a set of elements from the language L, i.e., as a set $K \subseteq L$ ($K \in 2^L$). An update U can be formalized as two sets of elements, namely, the elements to add and the elements to remove. Thus, U is a pair (U^+, U^-), where U^+, $U^- \subseteq L$ are consisting of the elements to add and remove (respectively) from the KB (i.e., $U \in 2^L \times 2^L$). We set $\mathbf{K}=2^L$, the set of all KBs, and $\mathbf{U}=2^L \times 2^L$, the set of all updates.

Normally, an update algorithm should be able to support any update operation (from **U**) upon any KB (from **K**). In practice however, there are various combinations that are disallowed by specific algorithms for various reasons, such as implementation efficiency and/or simplicity, good computational complexity, difficulty in determining the most “rational” result, inability to satisfy certain invariants for particular combinations of KBs/updates etc. Usually, invalid KBs are overruled as input to the algorithms, as invalid KBs (often) carry no logical meaning, so updating them would necessarily involve a number of questionable and arbitrary decisions by the algorithm. Certain (difficult) types of updates are also overruled in many cases, the most important reasons being the difficulty of implementing an algorithm to handle them, or the inability to satisfy certain invariants of the algorithm for certain cases.

The supported combinations are formalized using a non-empty set $\mathbf{D}\subseteq 2^L\times 2^L\times 2^L$, corresponding to the allowable pairs of KBs (from 2^L) and updates (from $2^L\times 2^L$); **D** is called the *domain of application* of the update algorithm. Note that this definition of **D** is very flexible because it allows, e.g., to support different updates per KB. For $K\in\mathbf{K}$, $U=(U^+,U^-)\in\mathbf{U}$, we will write $(K,U)\in\mathbf{D}$ to denote that $(K,U^+,U^-)\in\mathbf{D}$.

The language L is assumed to be equipped with some generic *consequence operator* Cn, which is a function $Cn:2^L\rightarrow 2^L$; intuitively, a consequence operator returns the closure of a given set of elements. Usually, a consequence operator is assumed to satisfy iteration, inclusion and monotony [5], but this assumption is not necessary here. The pair (L,Cn) forms the underlying *logic*.

Finally, the logic is assumed to be equipped with some validity context which is formalized as a non-empty set $\mathbf{V}\subseteq 2^L$, containing all the valid KBs from **K**. The set **V** is used to invalidate particular sets of elements that correspond to a non-meaningful or inconsistent KB per the semantics of the underlying logic; however, it could also be used in order to impose particular user-defined constraints on the elements (data), such as business rules or integrity constraints.

Summarizing, the context of application is a 4-tuple $<L,\mathbf{D},Cn,\mathbf{V}>$, where:

- $L\neq\emptyset$ is the language
- $\mathbf{D}\subseteq 2^L\times 2^L\times 2^L$, $\mathbf{D}\neq\emptyset$ is the domain of application of the update algorithm
- $Cn:2^L\rightarrow 2^L$ is the consequence operator of the logic
- $\mathbf{V}\subseteq 2^L$, $\mathbf{V}\neq\emptyset$ is the set of valid KBs that the logic admits

Formalizing the Update Algorithm: The Selection Mechanism

Step 1 of the presented meta-algorithm dictates a straightforward application of an update $U=(U^+,U^-)$ upon a KB K without any further considerations (validity etc); this is called the *raw application* (+) of U upon K and is defined as: $K+U=(K\cup U^+)\setminus U^-$.

Usually, the raw application of an update satisfies the Principle of Success, as the elements in the update are added and removed to the KB as requested. Note however, that this is not always true, as the inference mechanism may re-introduce some removed element. In addition, if there is some element $x\in U^+\cap U^-$, then + does not satisfy the Principle of Success; in fact, there is no way to satisfy success for such an update, as it requires the same element to be added and removed at the same time. Such updates (for which $U^+\cap U^-\neq\emptyset$) are called *inherently infeasible*.

Similarly, there are updates which cannot be applied upon a KB in a satisfactory way, even if not inherently infeasible. Such updates will be called *infeasible*. For example, an update requiring the addition of two conflicting elements would be infeasible. Often, infeasible (and inherently infeasible) updates are excluded from the domain of application (**D**).

As already mentioned, the most critical (and difficult) part of the update algorithm is the selection mechanism, which is used to determine (in step 3) the set of side-effects to apply upon the original KB, based on the impact that each option would have upon the KB. A set of side-effects is actually another update, i.e., a set of requests for adding and removing elements. Note that the impact of an update may differ depending on the KB upon which it is applied; furthermore, the selection should also consider the direct effects, because the decision (per the Principle of Minimal Change) should be based on the impact of the entire set of changes that will eventually be applied upon the original KB (i.e., effects plus side-effects).

Note that the selected side-effects (and the original effects) are ultimately applied (using +) upon the original KB, so the selection of side-effects actually determines the update result as well. Moreover, given that this selection is the only non-trivial step of the meta-algorithm, one can say that the design of an update algorithm is essentially reduced to the design of the mechanism that determines the side-effects to apply, depending on the KB, the update and, of course, the context of application.

The decision on the side-effects to apply is actually a selection over a pool of different possible sets of side-effects that satisfy the other constraints (basically, Success and Validity). The standard way to implement this is, first, to determine the pool of options (sets of side-effects) that could (potentially) be selected, and, second, to use some kind of filtering (or ranking mechanism) to select the preferred one. Thus, the selection mechanism can be formalized as a function σ that takes a non-empty set of updates in its input (corresponding to the potential sets of effects and side-effects to apply, i.e., the pool of options) and returns one member of the set that corresponds to the preferred update; formally, $\sigma:2^{\mathbf{U}}\setminus\varnothing\rightarrow\mathbf{U}$, such that $\sigma(S)\in S$ for all $S\in 2^{\mathbf{U}}\setminus\varnothing$.

Given a context of application $<L,\mathbf{D},Cn,\mathbf{V}>$, an update algorithm can be defined as a function $\bullet:\mathbf{D}\rightarrow\mathbf{K}$. The input to the function is a KB (say $K\in\mathbf{K}=2^L$) and an update (say $U\in\mathbf{U}=2^L\times 2^L$) which are allowed by the domain of application (i.e., $(K,U)\in\mathbf{D}$); we will use infix notation ($K\bullet U$) to denote the result of the function •.

As already mentioned, an update function • is usually assumed to satisfy certain principles, whose formalization is given below:

- Principle of Algorithmic Adequacy: the operation • is a function and for all $K\in\mathbf{K}$, $U\in\mathbf{U}$, such that $(K,U)\in\mathbf{D}$, it holds that $K\bullet U\in\mathbf{K}$.
- Principle of Irrelevance of Syntax: consider $K_1,K_2\in\mathbf{K}$ and $U_1=(U_1^+,U_1^-)\in\mathbf{U}$, $U_2=(U_2^+,U_2^-)\in\mathbf{U}$, such that:
  - (K_1,U_1), $(K_2,U_2)\in\mathbf{D}$
  - $Cn(K_1)=Cn(K_2)$
  - For all $u\in U_1^+$ there is a $u'\in U_2^+$ such that $Cn(\{u\})=Cn(\{u'\})$
  - For all $u\in U_1^-$ there is a $u'\in U_2^-$ such that $Cn(\{u\})=Cn(\{u'\})$
  - For all $u\in U_2^+$ there is a $u'\in U_1^+$ such that $Cn(\{u\})=Cn(\{u'\})$
  - For all $u\in U_2^-$ there is a $u'\in U_1^-$ such that $Cn(\{u\})=Cn(\{u'\})$

  Then, it holds that $Cn(K_1\bullet U_1)=Cn(K_2\bullet U_2)$.

- Principle of Validity: for all $K \in \mathbf{K}$, $U \in \mathbf{U}$, such that $(K,U) \in \mathbf{D}$ it holds that $K \bullet U \in \mathbf{V}$.
- Principle of Success: for all $K \in \mathbf{K}$, $U=(U^+,U^-) \in \mathbf{U}$, such that $(K,U) \in \mathbf{D}$ it holds that $U^+ \subseteq Cn(K \bullet U)$ and $U^- \cap Cn(K \bullet U)=\varnothing$.
- Principle of Minimal Change: for all $K \in \mathbf{K}$, $U=(U^+,U^-) \in \mathbf{U}$, such that $(K,U) \in \mathbf{D}$, for the selection function σ, and for the set $S=\{U_0=(U_0^+,U_0^-) \in \mathbf{U} \mid K+U_0 \in \mathbf{V}, U^+ \subseteq Cn(K+U_0), U^- \cap Cn(K+U_0)=\varnothing\}$, it holds that $S \neq \varnothing$ and $Cn(K \bullet U)=Cn(K+\sigma(S))$.

Some notes on the definitions of the principles are in order. First, it is easy to see that the Principle of Algorithmic Adequacy essentially requires $\bullet$ to be a function, so it is satisfied by the definition of $\bullet$. The Principle of Irrelevance of Syntax requires a strong relation between the two updates (U_1,U_2) in order to be applicable: there should exist a 1-1 and onto relationship connecting logically equivalent elements of U_1^+ and U_2^+ (and U_1^- and U_2^-); on the other hand, the KBs K_1, K_2 are merely required to be logically equivalent (as a whole). The strong requirement on the updates is based on the observation that, for example, removing two elements u_1, u_2 is not necessarily the same as removing the elements' consequence (i.e., an element u_3 implied by u_1, u_2).

The Principle of Validity is quite straightforward and requires the result to be valid. Similarly, the Principle of Success requires all the added and none of the removed elements to be part of the result. Finally, the Principle of Minimal Change requires that the result is in accordance with the selection function σ. More specifically, the result should be equivalent to the result of the raw application of the update $U'=\sigma(S)$ upon K, where U' is the update (effects and side-effects) that the selection function σ would select out of the pool of potential updates which satisfy the principles of Success and Validity (and are given by the set S). Note that $S=\varnothing$ if and only if there is no way to perform the update, i.e., if U is infeasible; thus, the Principle of Minimal Change cannot be applied for infeasible updates.

The function σ is assumed to capture a notion of minimality (of the impact of the changes) and is defined independently of the update algorithm, so the update algorithm is assumed to, in a sense, implement the selection function (rather than the other way around). Given a context of application and a selection function, the aforementioned principles uniquely determine (modulo logical equivalence) the result of an update algorithm that satisfies the principles. To see that, note that the set S in the definition of the Principle of Minimal Change is uniquely determined by the input of the algorithm (i.e., K,U) and the context of application, so the result of the update $(K \bullet U)$ is uniquely determined (modulo logical equivalence) by the selection made by σ upon S. This justifies our claim that an update algorithm implements a selection function. An update algorithm satisfying our principles will be called a *rational update algorithm associated with the selection function σ* (the reference to σ may be omitted if obvious from the context).

Implementing the Meta-Algorithm (Second and Third Level)

The meta-algorithm and the formalizations presented in the previous section (first level of abstraction) don't give any hints on how this algorithm could be implemented in any specific context. In this section, we will describe a proposal towards implementing the meta-algorithm (second level of abstraction). Our proposal is a general-purpose algorithm, which is very general and applicable to a multitude of contexts. Moreover, it is directly implementable, because it is based on a number of parameters, different values of which would make the algorithm applicable for different contexts. For illustration purposes, we will, in parallel, present one possible parameterization of the algorithm, which makes it suitable for RDF ontologies. This algorithm constitutes our third level of abstraction (most specific) and can be viewed as a proof of concept for the more abstract (generic) algorithmic patterns of the other two levels.

The Context of Application

In order to guarantee uniformity at this level of abstraction, we will restrict the language L to be of a particular form. More specifically, we will assume that we are given a finite and non-empty set of predicates (of various arities), say **P**, as well as a non-empty set of constants, say Σ, and that the language L consists of all the ground relational atoms that can be generated from the given predicates and constants, i.e., $L=\{P(x) \mid P\in\mathbf{P}$, P is a predicate of arity $n\geq 0$ and $x=(x_1,\ldots,x_n)\in\Sigma^n\}$. Note that L does not use the full power of predicate logic, as it does not allow operands (e.g., $\neg$, $\wedge$, $\forall$) in the KBs (and updates); such operands will be used in the validity rules (see below).

Restricting L to be of this form is not as severe a restriction as it looks, because many languages can be represented this way. For example, in the case of RDF, where the language is composed of triples, we can define the set of constants Σ to be the set of URIs and develop a set of predicates (i.e., the set **P**), to capture the knowledge carried by the triples (in a manner similar to [15]); using **P** and Σ, L can be easily defined as above. The semantics of the predicates in **P** have been selected in order to capture the semantics of the various triple types that can be formed in RDF (see [10]); for example, the semantics of the "subclass of" relationship is captured by the binary predicate C_IsA. This way, the semantics of the triple (A, rdfs:subclassOf, B) is captured by the relational atom C_IsA(A,B). More generally, we can define a mapping that allows the transformation of any RDF KB (i.e., any set of RDF/S triples) into a set of relational atoms from L with the same semantics (see [10] for details on the mapping).

Our algorithm will be able to handle all possible combinations of $(K,U)\in\mathbf{K}\times\mathbf{U}$, provided that K is a valid KB and U is not infeasible. The former restriction (K is valid) is due to the fact that it is unclear how to update an invalid KB. The latter restriction (U is not infeasible) allows the satisfaction of our principles. Thus, we define $\mathbf{D}=\{(K,U^+,U^-) \mid K\in\mathbf{V}, U=(U^+,U^-)\in\mathbf{U}$, U is not infeasible$\}$. Determining whether U is infeasible is not simple, so our algorithm detects U as infeasible in the process of updating. For the RDF-specific algorithm, **D** is defined in the same way.

Validity and inference are intermingled in our framework. Validity is encoded via a set of rules, so **V** contains all the KBs satisfying the rules. For technical reasons, such axioms are restricted to be disjunctive embedded dependencies (DEDs), which is a general class of FOL axioms; see [11] for details.

The inference mechanism is also handled using a set of DED rules; however, instead of using these rules as inference rules and encoding them inside the Cn function, we use them as validity rules. This way, a KB is required to be closed with respect to inference in order to be valid. This fact simplifies things in various ways.

For one, the definition of Cn is now straightforward, because there are no “inference rules” or “inferred knowledge” in the strict sense, so we can set Cn(K)=K for all $K \in \mathbf{K}$. Secondly, this option greatly simplifies the definition of validity rules themselves, because, in a closed KB, we only have to check the explicit knowledge in order to verify validity; for non-closed KBs, implicit knowledge should also be considered. For example, in the RDF framework, there is a rule that overrules cyclic subsumption relationships (explicit or implicit). Given that subsumption relationships are transitive, in a non-closed KB we would have to check for cycles of arbitrary length, making the rule quite complicated. For closed KBs, we only have to check for cycles of length 2, so the rule becomes: $\forall x,y$ C_IsA(x,y)$\rightarrow\neg$C_IsA(y,x).

Moreover, the approach of encoding the inference as validity rules simplifies all checking for the effects of inference with respect to the Principle of Success. Given that all valid KBs are also closed, we only have to check the explicit knowledge in order to verify that the Principle of Success is satisfied in a valid KB. Finally, this approach guarantees that the Principle of Irrelevance of Syntax is satisfied, as all operations are performed upon the KBs’ closures, and equivalent (and valid) KBs are equal.

Another important remark on the validity rules is that they assume closed-world semantics. This fact, along with the fact that L contains only relational atoms and that there is no “real inference”, implies that a positive ground fact P(x) is implied by a valid KB K if and only if $P(x) \in K$, whereas a negated ground fact $\neg P(x)$ is implied by a valid KB K if and only if $P(x) \notin K$. This simplifies checking for the validity of a rule; for example, in order for the rule $\forall x,y$: C_IsA(x,y)$\rightarrow\neg$C_IsA(y,x) to be satisfied by a KB K, it is enough that C_IsA(y,x)$\notin$K whenever C_IsA(x,y)$\in$K.

The validity (and inference) rules employed for the RDF case are not presented here due to lack of space; the reader is referred to [10] for a full list.

The Selection Mechanism and the Update Algorithm

One could imagine various ways to define the selection mechanism that is necessary for the development of the update algorithm. Our choice is the introduction of a generic “cost model” that assigns a cost to each update (set of side-effects). We chose to model the cost qualitatively, i.e., through a relation $<:\mathbf{U}\times\mathbf{U}$ that determines whether the cost of an update is “greater” than the cost of another. Given $<$, the function σ can be defined as: $\sigma(S)=U$ such that $U \in S$ and $U \leq U_0$ for all $U_0 \in S$. Since the cost of an update depends also on the KB upon which it is applied, a more accurate definition would be to say that the cost model is a family of relations $\{<_K \mid K \in \mathbf{V}\}$; however, K is often obvious from the context, so we will omit the subscript.

The above definition comes with a number of problems. On the theoretical side, the relation must be such that any set of updates S should have a unique minimum, otherwise the selection function would be ill-defined. On the implementation side, it does not give us any algorithmic method to compute the set of potential updates S (which, per the definition of the Principle of Minimal Change, is the set of all updates that would satisfy Success and Validity); moreover, S could be infinite, so our definition forces us to compare a possibly infinite number of updates which is not possible from an implementation point of view. Those problems can be all resolved by carefully selecting < so that it satisfies certain properties and by providing an algorithm that determines S in a clever way, taking into account the properties of <.

Towards this end, we can initially note that certain elements of an update should not contribute to its cost; more specifically, a request to add an element in a KB that already contains it, or to remove an element that is not in the KB to begin with, should not add up to the cost. Such elements in an update are called *void* and, even though they can make a difference on the update result, they should have no effect on the cost of the update itself. This requirement is called *conflict sensitivity* and can be formalized by requiring that the cost of an update is determined solely by its non-void elements. More formally, consider an update $U=(U^+,U^-)$; we define the *update's actual effects*, denoted by AE(U), to be the update $AE(U)=(U_0^+,U_o^-)$ such that $U_0^+=U^+\setminus K$, $U_0^-=U^-\cap K$. For two updates $U_1=(U_1^+,U_1^-)$, $U_2=(U_2^+,U_2^-)$, conflict sensitivity requires that $U_1 \leq U_2$ if and only if $AE(U_1) \leq AE(U_2)$.

In order for a unique minimum to exist, it should be the case that any two sets of potential effects and side-effects should be comparable, thus < should satisfy *totality*: for any two updates U_1, U_2, it should hold that either $U_1 \leq U_2$ or $U_2 \leq U_1$.

For the same reasons, we shouldn't be able to identify two different updates U_1, U_2 whose cost is the same, i.e., the relation should be antisymmetric. However, due to the conflict sensitivity requirement, we should relax this condition, by saying that two updates could have the same cost, but only if their actual effects are the same. This property is called *partial antisymmetry* (or delta antisymmetry [11]): for any two updates U_1,U_2 it holds that $U_1 \leq U_2$ and $U_2 \leq U_1$ if and only if $AE(U_1)=AE(U_2)$.

Intuitively, we expect that the addition of side-effects to an update could never reduce its cost. This property is called *monotonicity*: for any two updates $U_1=(U_1^+,U_1^-)$, $U_2=(U_2^+,U_2^-)$, such that $U_1^+ \subseteq U_2^+$ and $U_1^- \subseteq U_2^-$ it holds that $U_1 \leq U_2$.

Finally, it makes sense for the < relation to satisfy *transitivity*. Thus, for any three updates U_1, U_2, U_3 such that $U_1 \leq U_2$ and $U_2 \leq U_3$ it holds that $U_1 \leq U_3$.

All five of the above properties make intuitive sense, so we require that any given ordering relation should satisfy them. For one such relation, suitable for the RDF case, refer to [10]. Most importantly, these properties are also crucial to guarantee that a unique minimum update will exist for any set of updates S. Moreover, they allow us to create an algorithm that will determine the set of potential updates S, and, at the same time, prune this (possibly infinite) set into an, as small as possible, set that will definitely contain the (unique) minimum update.

Let's first see how the set of potential updates (effects and side-effects) can be determined. Consider a KB K and an update $U=(U^+,U^-)$. Let us assume, initially, that U contains only non-void updates, i.e., AE(U)=U. In this case, by the Principle of Success, any potential set of effects and side-effects $U_0=(U_0^+,U_o^-)$ should contain at least as many elements as U, i.e., $U_0^+ \supseteq U^+$, $U_0^- \supseteq U^-$. Given the monotonicity property, the addition of any element to U^+ or U^- could not lead to a better update, in terms of cost, therefore it makes sense to add elements to U on a need-to-have basis; any non-necessary addition would lead to a non-optimum update. In this respect, a necessary addition is one that somehow contributes towards the resolution of some invalidity that would arise if we applied U upon K as-is (i.e., some invalidity in K+U).

Thus, if K+U is valid, then there is nothing to be added to U, and U is the only element of S. If K+U is not valid, then it invalidates at least one of the rules. The rules can be written in conjunctive normal form (CNF), and, given the fact that they are DEDs, K+U will invalidate at least one of the conjunctions in the rule, for at least one set of constants. This gives us directly the possible actions that can restore validity.

As an example, take the RDF rule $\forall x,y$ C_IsA(x,y)$\rightarrow$CS(x)$\wedge$CS(y), which states that a class IsA relationship (C_IsA(x,y)) should be applied between classes only, so both x and y should be classes (CS(x)$\wedge$CS(y)). This rule can be equivalently rewritten (in CNF) as: ($\forall x,y$ $\neg$C_IsA(x,y)$\vee$CS(x)) $\wedge$ ($\forall x,y$ $\neg$C_IsA(x,y)$\vee$CS(y)). If it is invalidated by K+U, then at least one of the operands of the conjunction will be invalidated. For example, we might have defined an IsA relationship C_IsA(A,B), but not defined A as a class (i.e., CS(A)$\notin$K+U). That would violate the first operand of the conjunction, for x=A, y=B. It is clear that there are only two minimal ways to restore this violation: either by removing the problematic IsA (which corresponds to the addition of C_IsA(A,B) to U^-) or by adding A as a class (which corresponds to the addition of CS(A) to U^+); this is directly derivable from the rule's syntactical form.

Out of the two different options (i.e., removing the IsA, or adding A as a class), we cannot, a priori, know which one would lead to the best update. Thus, we have to explore both options. To do that, we spawn two different potential results, namely $U_1=(U^+,U^-\cup\{$C_IsA(A,B)$\})$ and $U_2=(U^+\cup\{$CS(A)$\},U^-)$, and repeat the process for each one of K+U_1, K+U_2 independently, as if U_1 (or U_2) was the original update.

The recursion can stop in two different ways. The first case is when the current update is inherently infeasible. If, for example, the original update requested the addition of C_IsA(A,B), then update U_1 would be inherently infeasible, and this option would be overruled and no longer explored. The second case is when the current update is such that its raw application upon K gives a valid KB. In this case, we don't need to add any more side-effects and the current update is added to the pool of potential updates S. The algorithm described above can be found in Table 1 below.

If all different recursion paths lead to inherently infeasible updates, or if the original update is inherently infeasible, then the update is infeasible and an error is reported. Otherwise, as the updates are returned, they are compared using < and the less costly one is kept; this is possible, given the totality property, and safe, given the transitivity property (i.e., we can't drop any update that will turn out to be minimal later). Partial antisymmetry implies that if we can't determine the less costly one, then it is because those updates are the same as far as the actual changes are concerned; therefore, we can arbitrarily select any of the two without affecting the final result.

If U also contains void updates, the process is identical; note however that void updates may affect the result in the sense that they could contradict some side-effect, thus causing us to invalidate (reject) some (otherwise valid) recursion path(s).

Update(K,U), where K is the KB to update, U the update
1. If U is inherently infeasible, return INFEASIBLE
2. If K+U$\in$**V**, then return U
3. Otherwise, find a rule r that is invalidated by K+U
4. For each possible side-effect that would restore the validity of r, spawn a different update U_i, such that U_i is equal to U extended with the new side-effect
5. For each U_i, set U_i'=Update(K, U_i) and return the minimal (per <) of U_i'

Table 1 Update algorithm (recursive function)

In practice, the implementation of the algorithm can admit a number of optimizations. The most obvious one dictates an initial depth-first search for some update that would be put in S. After we find one such update and put it in our pool S, we can compare (in step 1) the currently considered update with the one that is currently identified as the best one. If the current update is already more costly than the currently best one, then it doesn't make sense to explore the given recursive path any longer, as any new side-effects would only increase further the cost of the current update. This allows us to prune several recursive paths early enough and improve the efficiency of the algorithm. Other, heuristic-based optimizations are also possible, but such optimizations are application-dependent, as they are based on the particular rules and ordering used; a description of the optimizations that we used for the RDF case is beyond the scope of this paper, but can be found in [10].

To guarantee correctness, we must show that the set S produced by the above process will include the minimal (per $<$) update. This is true because elements are added as side-effects only if they are necessary to resolve some invalidity. Thus, the recursive process will return all the minimal (with respect to $\subseteq$) updates that would satisfy Success and Validity; by monotonicity, the minimal one (per $<$), will be one of these.

Moreover, we can verify that the algorithm satisfies the principles. The Principle of Algorithmic Adequacy is guaranteed, as the algorithm will return, in a deterministic way, a single result (update) which is then raw applied upon the KB, unless the original KB is invalid (which is tested as a precondition) or the update is infeasible (which is tested during the recursion). The Principle of Irrelevance of Syntax is guaranteed by the validity rules themselves and the Principle of Validity. The Principle of Validity is guaranteed by the recursive process, in particular by the fact that an update is returned as a potential result only if its raw application to K would lead to a valid KB. The Principle of Success is guaranteed by the fact that all updates returned by the algorithm build upon U, so the actual update performed will include U. Finally, the Principle of Minimal Change is guaranteed by the fact that S contains the minimal (per $<$) update and that the contents of S are filtered out using $<$ (which is used to define σ).

It remains to show that the algorithm terminates. This is true if and only if the number and the size of the explored recursive branches is finite; equivalently, the number of updates that are put in S should be finite. Unfortunately, this is not guaranteed by the properties already described; termination depends on the rules themselves, as well as on the ordering. For the particular validity rules and ordering that we used in the RDF case [10], it can be shown that the minimal update cannot contain references to more than one constants (i.e., URIs) which are not in the KB or the update. Given that this set of URIs is finite, and that the set of predicates **P** is also finite, it follows that there is a finite set of elements that can be considered as candidates for inclusion in the final set of effects and side-effects. Thus, all updates in S are bounded in terms of the elements that they can potentially contain, so S is finite.

Discussion and Comparison With Related Work

Current ontology evolution algorithms employ an approach similar to the described one to address the problem of evolution: first, they identify their setting (context of application), by determining the language and the validity and inference rules that they want to impose, as well as the supported updates and KBs; then, for each possible update, they identify the various validity problems that its direct application would cause per KB; finally, they identify (select) the best way to resolve such problems.

The main (and crucial) difference between these approaches and ours, is that, in our approach, only the context of application is determined at design time; both the identification of the validity problems (depending on the KB and the update), as well as the determination of the optimum way to resolve such problems, are made at run time, using logic which is inbuilt in the algorithm.

On the other hand, the approaches currently used in the literature perform all the above steps at design time. In particular, for each update in the pre-selected pool of supported updates, an exhaustive analysis of the various relevant states (properties) of the KB is performed. For each such state, the possible invalidities are considered, as well as the optimum way to resolve each such invalidity. At the end of the process, an extensive list of cases that contains the supported updates, the relevant KB states (per update) and the respective actions (per update and KB state) that the update algorithm should take in order to execute the update upon the KB under question is produced. Finally, this list is used as a guide in order to implement the update algorithm.

This ad-hoc approach has both advantages and disadvantages. The main advantage is that the determination of the side-effects and the related actions are hard-coded inside the algorithm; thus, we expect an ad-hoc algorithm to be more efficient than our general-purpose one. This has been addressed in our case by doing the same process (developing special-purpose algorithms) for the most interesting updates [11].

On the other hand, the ad-hoc approach also causes a number of problems. First of all, each invalidity, as well as each of the possible solutions, needs to be considered individually, using a highly tedious, case-based reasoning which is error-prone and gives no formal guarantee that the cases and options considered are exhaustive. As an answer to such problems, the designers are often forced to support only a few different updates (usually only atomic ones); in our more general approach, any update, containing any number and/or type of atomic add/remove requests, can be handled.

Similarly, the nature of the selection mechanism cannot guarantee that the selections (regarding the proper side-effects) that are made for different operations exhibit a consistent overall behaviour. This is necessary in the sense that the side-effect selections made for different operations (and for different KBs) should be based on an operation-independent global policy regarding changes. Such a global policy is difficult to implement and enforce in an ad-hoc system.

Furthermore, any minor change on our thinking regarding how to address the updates should force a redesign of the entire algorithm. In our case, we can change the validity rules and the selection function quite easily. Such changes will not affect the core of the algorithm and can even be part of user's input (they may invalidate some of the application-specific optimizations though).

Popular systems face a lot of limitations due to the above problems. For example, OilED deals only with a very small fraction of the operations that could be defined using its underlying language as any change operation that would be triggering side-effects is unsupported. In Protégé, the design choice to support a large number of operations has forced its designers to limit the flexibility of the system by offering only one way of realizing a change (i.e., the user cannot adjust the selection mechanism);

in OntoStudio, they are relieved of dealing with the complexity of the aforementioned case-based reasoning as the severe limitations on the expressiveness of the underlying language constrain drastically the number of supported operations, as well as the possible cases to consider. Finally, in KAON, they provide a feature that allows a limited parameterization of the selection mechanism; however, some possible side-effects are missing (ignored) for certain operations, while the selection process implied by KAON's parameterization may exhibit non-uniform behaviour in some cases. For example, implicit knowledge is considered/retained when removing a class, but not when removing an IsA, even under the same parameterization [11].

As an extra advantage, our approach has resulted to an adjustable general-purpose algorithm that can support different languages (this would require some extra design work to determine the predicates to use, the validity rules etc). It makes more sense for the general-purpose algorithm to be applied for simpler languages, such as those used in standard ontology evolution. This is true because our encoding of the language into relational atoms may be difficult for complicated logics; in such cases, we should try a different implementation of the meta-algorithm.

Conclusion and Future Work

We studied the process of change at three different levels with varying degrees of abstraction. At the first, most abstract level, we discussed the general properties that an update algorithm should satisfy and provided a meta-algorithm to deal with changes. At the second level, we provided a general-purpose algorithm for implementing the meta-algorithm, which is based on some parameters (e.g., predicates, validity rules, ordering). The third level is most specific and is actually an implementation of the general-purpose algorithm for the case of RDF, as a proof of concept. The RDF-specific algorithm has been implemented in a large scale real-time system, as part of the ICS-FORTH Semantic Web Knowledge Middleware (SWKM), which includes a number of web services for managing RDF/S KBs.

The main difference of our approach with respect to existing ones is that it delegates the error-prone and time-consuming task of identifying and resolving the various invalidities that could occur during a change operation from design-time to run-time. This way, our methodology exhibits a faithful behaviour with respect to the various choices (parameters) involved, regardless of the particular KB or update at hand. It lies on a formal foundation, issuing a solid and customizable method to handle any type of change on any KB, including operations not considered at design time. In addition, it avoids resorting to the error-prone per-case reasoning of other systems, as all the alternative side-effects of an update can be derived automatically in an exhaustive and provably correct manner.

Future work includes detailed experimental evaluation of the performance of the RDF-specific algorithm and the incorporation of techniques and heuristics that would further speed it up. Our general idea could also be applied, with minor changes, to address the problem of ontology debugging, which is the process of identifying and removing undesirable logical contradictions from an ontology [6]; this application is planned for future work.

Acknowledgements

This work was partially supported by the EU projects CASPAR (FP6-2005-IST-033572) and KP-Lab (FP6-2004-IST-4). Part of this paper is the result of joint work with Vassilis Christophides and Grigoris Antoniou.

References

1. C. Alchourron, P. Gärdenfors, and D. Makinson. "On the Logic of Theory Change: Partial Meet Contraction and Revision Functions". In *Journal of Symbolic Logic*, 50:510-530, 1985.
2. S. Bechhofer, I. Horrocks, C. Goble, and R. Stevens. "OilEd: A Reason-able Ontology Editor for the Semantic Web". In *Proceedings of the 24th German / 9th Austrian Conference on Artificial Intelligence (KI-01)*, 2001.
3. D. Brickley and R.V. Guha. "RDF Vocabulary Description Language 1.0: RDF Schema". W3C Recommendation, 2004. http://www.w3.org/TR/rdf-schema/
4. M. Dalal. "Investigations into a Theory of Knowledge Base Revision: Preliminary Report". In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pp. 475-479, 1988.
5. G. Flouris. *On Belief Change and Ontology Evolution*. Doctoral Dissertation, Department of Computer Science, University of Crete, 2006.
6. G. Flouris, D. Manakanatas, H. Kondylakis, D. Plexousakis, and G. Antoniou. "Ontology Change: Classification and Survey". In *Knowledge Engineering Review (KER)*, 23(2):117-152, 2008.
7. T. Gabel, Y. Sure, and J. Voelker. "D3.1.1.a: KAON – Ontology Management Infrastructure". SEKT deliverable, 2004.
8. P. Gardenfors. "Belief Revision: An Introduction". In P. Gardenfors (ed.), *Belief Revision*, pages 1-20, Cambridge University Press, 1992.
9. S.O. Hansson. "A Survey of Non-Prioritized Belief Revision". In *Erkenntnis*, 50:413-427, 1999.
10. G. Konstantinidis. *Belief Change in Semantic Web Environments*. Master Thesis, Computer Science Department, University of Crete, 2008.
11. G. Konstantinidis, G. Flouris, G. Antoniou, and V. Christophides. "A Formal Approach for RDF/S Ontology Evolution". In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI-08)*, pp. 405-409, 2008.
12. M. Langlois, R.H. Sloan, B. Szorenyi, and G. Turan. "Horn Complements: Towards Horn-to-Horn Belief Revision". In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI-08)*, pp. 466-471, 2008.
13. E. Miller, R. Swick, and D. Brickley. "Resource Description Framework (RDF)". W3C Semantic Web Activity, 2006. http://www.w3.org/RDF/
14. N. Noy, R. Fergerson, and M. Musen. "The Knowledge Model of Protégé-2000: Combining Interoperability and Flexibility". *In Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management: Methods, Models, and Tools (EKAW-00)*, pp. 17-32, 2000.
15. 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.
16. Y. Sure, J. Angele, and S. Staab. "OntoEdit: Multifaceted Inferencing for Ontology Engineering". In *Journal on Data Semantics*, 1(1):128-152, 2003.