

Ontology Evolution: A Framework and its Application to RDF

George Konstantinidis
Department of Computer
Science, University of Crete
and Institute of Computer
Science, FO.R.T.H., Heraklion,
Greece
gconstan@csd.uoc.gr

Giorgos Flouris
Institute of Computer Science,
FO.R.T.H., Heraklion, Greece
fgeo@ics.forth.gr

Grigoris Antoniou
Vassilis Christophides
Institute of Computer Science,
FO.R.T.H. and Department of
Computer Science, University
of Crete, Heraklion, Greece
{antoniou,christop}@ics.forth.gr

ABSTRACT

The algorithms dealing with the incorporation of new knowledge in an ontology often share a rather standard process of dealing with changes. This process consists of the determination of the allowed change operations, the identification of the inconsistencies that could be caused by each such operation as well as the various alternatives to deal with each such inconsistency, and, finally, some (manual or automatic) selection mechanism that allows the determination of the “best” of these alternatives. Unfortunately, most ontology evolution algorithms implement these steps using a case-based, ad-hoc methodology, which is cumbersome and error-prone. In this paper we propose a general framework for ontology change management that generalizes the methodology employed by existing tools. The introduction of this framework allows us to devise a whole class of ontology evolution algorithms, which, due to their formal underpinnings, avoid many of the problems exhibited by ad-hoc frameworks. We exploit this framework by implementing a specific ontology evolution algorithm for RDF ontologies.

1. INTRODUCTION

Change management is a key component of any knowledge-intensive application. The same is true for the Semantic Web, where knowledge is usually expressed in terms of ontologies and refined through various methodologies using *ontology evolution* techniques. The most critical part of an ontology evolution algorithm is the determination of *what* can be changed and *how* each change should be implemented. The main argument of this paper is that this determination can be split into the following 5 steps, which, although unrecognized, are shared by many evolution frameworks:

1. *Model Selection.* The allowed changes are constrained by the expressive power of the ontology representation

model. Thus, the selection of the model may have profound effects on what can be changed and constitutes an important parameter of the evolution algorithm.

2. *Supported Operations.* In step 2, the allowed change operations upon the ontology are specified.
3. *Consistency Model.* Problems related to the consistency of the resulting ontology may arise whenever a change operation is executed; such problems depend on the consistency model assumed for ontologies.
4. *Inconsistency Resolution.* This step determines, for each supported operation and inconsistency problem, the different (alternative) actions that can be performed to restore the consistency of the ontology.
5. *Action Selection.* In step 5, a selection process determining the most preferable among the various potential actions identified in the previous step is devised.

The first two steps determine what can be changed and correspond to the *change capturing phase* introduced in [9]; the last three steps correspond to the *semantics of change phase* of [9] and indicate how changes should be implemented. Unfortunately, most of the existing frameworks (e.g., [3, 4, 7, 10]) address ontology evolution issues related to the above 5 steps in an ad-hoc way. As we will see in section 2, this approach causes a number of problems (e.g., reduced flexibility, limited evolution primitives etc), so evolution algorithms could benefit a lot from the formalization of the aforementioned change management process.

In section 3, we introduce a general framework that models the various steps of this process. We exhibit the merits of our framework via the development of a general-purpose algorithm for RDF [8] ontology updates. Our framework allows us to deal with arbitrary change operations (rather than a predetermined set). In addition, it considers all the inconsistencies related to each change and all the possible ways to deal with them. Finally, it provides a parameterizable method to select the “best” option to deal with an inconsistency, according to some metric. The formal nature of the process allows us to avoid resorting to the tedious and error-prone case-based reasoning that is necessary in other frameworks for determining inconsistencies and solutions to them, and provides a uniform way to select the “best” option out of the list of available ones, using some total ordering. We propose one specific ordering for our

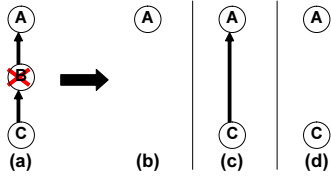


Figure 1: Three alternatives for deleting a class

RDF algorithm and demonstrate how we can devise certain special purpose algorithms (similar to the existing ad-hoc frameworks) for coping with RDF changes, which, due to their formal underpinnings and their compatibility with the general framework, enjoy the interesting properties of the framework described above.

2. EVOLUTION PROCESS

2.1 Model and Operations

The first step towards developing an evolution algorithm is to determine the underlying representation model for the evolving ontology (step 1). Most systems assume a language supporting the basic constructs used in ontology development, like class and property IsAs, instantiation relationships and domain and range restrictions for properties. The selection of the representation model obviously affects the operations that can be supported; for example, OntoStudio (formerly OntoEdit [10]) does not support property subsumption relations so all related changes are similarly overruled.

Further restrictions to the allowable changes may appear in step 2, where various design decisions may disallow certain operations, despite the fact that they could, potentially, be supported by the underlying ontology model. For example, OntoStudio does not allow the manipulation of implicit knowledge, whereas Oiled [4] does not support any operation that would render the ontology inconsistent (i.e., it does not take any actions to restore consistency, but rejects the entire operation instead).

In general, change operations can be either *elementary* (involving a change in a single ontology construct) or *composite* ones (involving changes in multiple constructs). Since composite operations can involve changes in an arbitrary number of constructs, there is an infinite number of them [5]. Although there are conditions under which composite operations can be decomposed into a series of elementary ones [5], for ad-hoc systems this is not of much help, as the decomposition of a non-supported operation into a series of supported ones (even if possible) should be done manually by the ontology engineer.

The above observations indicate an important inherent problem with ad-hoc algorithms, which can only deal with a predefined (and finite) set of supported operations, determined at design time. Therefore, any such algorithm is limited, because it can only support some of the potential changes upon an ontology, namely the ones that are considered more useful for practical purposes.

2.2 Inconsistencies and Solutions

One of the basic requirements for a change operation is that the result of its application should be a *consistent* ontology, according to the consistency model defined in step

3; this principle is necessary in order for the resulting ontology to make sense. On the other hand, we require that the resulting ontology implements the change operation requested; the latter requirement will be called *success* and implies that the change request should be satisfied (if possible). Both principles are motivated by research on the related field of belief revision [1].

These two requirements are, in some cases, conflicting: the naive raw application of a change operation upon an ontology (which is the minimum requirement for success) may result to an inconsistent ontology, thus violating the consistency principle. This problem can be overcome by initiating additional actions, in the form of additional change operations (side-effects) upon the ontology, that would restore consistency (step 4). It can be shown that, for certain updates, no additional actions could restore consistency. For such updates it is not possible for both principles (success, consistency) to be satisfied, so these operations are rejected; such updates are called *infeasible*.

As a case study, let us consider the change operation depicted in Fig. 1(a), where the ontology engineer expresses the desire to delete a class (B) which happens to subsume another class (C). It is obvious that, once class B is deleted, the IsAs relating B with A and C would refer to a non-existent class (B), so they should be removed; the consistency model should capture this case, and attempt to resolve it. One possible result of this process, employed by Protégé [7], is shown in Fig. 1(b); in that evolution context, a class deletion causes the deletion of its subclasses as well. This is not the only possibility though; figures 1 (c) and (d), present other potential results of this operation, where in (c), B 's subclasses are re-connected to its father, while in (d), the implicit IsA from C to A is not taken into account. KAON [3], for example, would give either of the three as a result, depending on a user-selected parameter.

In this particular example, both systems detect the inconsistency caused by the operation and actively take action against it; however, the consistency model employed by different systems may be different in general. Moreover, notice that an inconsistency is not caused by the operation itself, but by the combination of the current ontology state and the operation (e.g., if B was not in any way connected to A and C , its deletion would cause no problems). Therefore, in order for a mechanism to propose solutions against inconsistencies, both should be taken into account. Notice that the mechanism employed by Protégé, in Fig. 1, identifies only a single set of side-effects, while KAON identifies three different reactions. This is not a peculiarity of this example; the inconsistency resolution mechanism employed by Protégé identifies only a single solution per inconsistency; this is not true for KAON and OntoStudio.

2.3 Selection Mechanism

The last component of an evolution algorithm (step 5) is a selection mechanism that would identify the most adequate inconsistency resolution action out of the possible ones (identified in step 4). Such a mechanism is not necessary for systems that identify only a single possible action, like Protégé, but it is critical for other systems. KAON, for example, provides a set of parameters (called *evolution strategies*) which allow the ontology engineer to tune the system's behavior and, implicitly, indicate what is the appropriate inconsistency resolution action for implementation

per case. OntoStudio provides a similar customization over its change strategies.

Notice that our preference among resulting ontologies reflects in a preference among side-effects of the corresponding update operations. For instance if we prefer the result of Fig. 1(c), we can equivalently say that we prefer the addition of the subsumption relation shown in (c) together with the deletion of the two initial IsAs, as a side-effect to this operation, over the deletion of the two initial IsAs and class C , shown in (b), or just the deletion of the two relations, as in (d). Therefore, the evolution process can be tuned by introducing a preference ordering upon the operation’s side-effects that would dictate the related choice (evolution strategy). Given that the determination of the alternative side-effects depends on both the update and the ontology, there is an infinite number of different side-effects that may have to be compared. Thus, we are faced with the challenge of introducing a preference mechanism that will be able to compare any imaginable pair of side-effects.

It is worth noting here the connection of this preference ordering with the well-known belief revision principle of Minimal change [1] which states that the resulting ontology should be as “close” as possible to the original one. In this sense, the preference ordering could be viewed as implying some notion of relative distance between different results and the original ontology, as identified by the preference between these results’ corresponding side-effects.

2.4 Discussion

To the best of authors’ knowledge, all currently implemented systems employ ad-hoc mechanisms to resolve the issues described above. The designers of these systems have determined, in advance, the possible inconsistencies that could occur, the various alternatives for handling any such possible inconsistency, and have already pre-selected the preferable option (or options, for flexible systems like KAON) for implementation per case; this selection (or selections) is hard-coded into the systems’ implementations.

This approach causes a number of problems. First of all, each inconsistency, as well as each of the possible solutions to each of them, 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. Similarly, the nature of the selection mechanisms cannot guarantee that the selections (regarding the proper side-effects) that are made for different operations exhibit a “consistent” overall behavior. This is necessary in the sense that the side-effect selections made in different operations (and on different ontologies) 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.

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 upon its modeling as any change operation that would be triggering side-effects is unsupported; e.g., the operation of Fig. 1 is rejected. 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; 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 under-

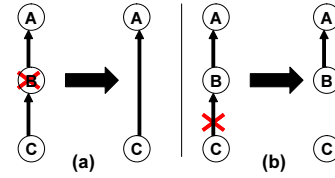


Figure 2: Implicit knowledge handling in KAON

lying model constrain drastically the number of supported operations. Finally, in KAON, some possible side-effects are missing (ignored) for certain operations, while the selection process implied by KAON’s parameterization may exhibit inconsistent or non-uniform behavior in some cases. As an example, consider Fig. 2, in which the same evolution strategy was set in both (a) and (b); despite this, the implicit IsA from C to A is only considered/retained in case (a).

Table 1 summarizes some of the key features of ontology evolution systems, categorized according to the 5-step process introduced in this paper, and shows how each feature is realized in each of the four systems discussed here, as well as in our framework, described in section 3 below.

We argue that many of the problems identified in this section could be resolved by introducing an adequate evolution framework that would allow the description of an algorithm in more formal terms, as a modular sequence of choices regarding the model used, the supported operations, the consistency model, the identification of plausible side-effects and the selection mechanism. Such a framework would allow justified reasoning on the system’s behavior, without having to resort to a case-by-case study of the various possibilities. To the best of the authors’ knowledge, there is no implemented system that follows this policy. In the next section we describe such a framework and specialize it to the case of RDF ontologies.

3. A FORMAL FRAMEWORK

Our evolution framework consists of a fine-grained modeling of ontologies (step 1), a description of how both elementary and composite operations can be handled in a unifying way (step 2), a consistency model formalized using integrity rules (step 3) which also allow us to document how side-effects are generated (step 4), and, finally, a selection mechanism based on an ordering of the side-effects in terms of some metric of “minimality” according to the principle of minimal change (step 5). This framework will be instantiated to refer to RDF updating, but can be used essentially for any language, by tuning the various parameters involved.

3.1 Model, Operations and Consistency

The representation model we use in this paper is the RDF language, in particular the model described in [8]. For ease of representation, RDF constructs will not be represented in the standard way, but we will use an alternative representation, which in short amounts to mapping each statement of RDF to a FOL predicate (see Table 2); this way, a class IsA between A and B , for example, would be mapped to the predicate: $C.IsA(A, B)$, while the domain of a property P , say C , would be denoted by $Domain(P, C)$. Note that the standard alternative mapping of RDF to FOL (e.g., for IsA: $\forall xA(x) \rightarrow B(x)$) does not allow us to map assertions of the form “ C is a class”, and, consequently, does not allow us

Table 1: Ontology Evolution in Systems

		Protégé	KAON	OntoStudio	OilED	Our Framework	
Change Representation	Fine-grained Model	✓	✓	×	✓	✓	
	Supported Operations	Elementary (consistent w.r.t its model)	✓	✓	✓	×	✓
		Composite	×	×	×	×	✓
Semantics of Change	Consistency	Transparent consistency context	×	×	✓	✓	✓
		Total set of possible inconsistencies	×	×	✓	×	✓
	Solutions to Inconsistencies	None				✓	
		One	✓				
		More than one alternatives		✓			
		All possible			✓		✓
	Selection mechanism	None (fixed)	✓			✓	
		Per-case parameterizable		✓	✓		
Globally parameterizable						✓	

Table 2: RDF facts to FOL predicates

Predicates	Intuitive meaning
$CS(C)$	C is a class in the RDF graph
$PS(P)$	P is a property in the RDF graph
$CI(x)$	x is a class instance in the RDF graph
$Domain(P, C)$	The domain of P is C
$Range(P, C)$	The range of P is C
$C.IsA(C_1, C_2)$	C_1 is a direct/indirect subclass of C_2
$P.IsA(P_1, P_2)$	P_1 is a direct/indirect subproperty of P_2
$C.Inst(x, C)$	x is a direct/indirect class instance of C
$PI(x, y, P)$	Pair(x,y) is a property instantiation of P

to handle operations like the addition or removal of a class, property, or instance (see [2] for more details on this issue). Notice that the same representation pattern can be used for other ontological languages as well [2].

We equip our FOL with closed semantics, i.e., admit the *closed world assumption* (CWA). This means that, for two formulas p, q , if $p \not\vdash q$, then $p \vdash \neg q$. An ontology is represented as a set of positive ground facts only, so, given CWA, it holds that: (a) an ontology is always consistent, (b) a positive ground fact is implied by an ontology iff it is contained in it, and, (c) a negative ground fact is implied by an ontology iff its positive counterpart is not contained in it.

An *update* is any set of positive and/or negative ground facts. By the principle of success and the properties (b), (c) above, we conclude that after executing an update, all positive ground facts in U will be included in the ontology, while all the positive counterparts of the negative ground facts in U will not be included in the ontology (unless the update is infeasible). Thus, positive ground facts in an update correspond to additions, while negative ones correspond to removals. This way of viewing updates allows us to handle essentially any operation, because any operation can be expressed as a set of additions and/or removals of ground facts in our model.

Our framework needs also to define its consistency model in a formal way. Consistency can in general be formalized using a set of integrity constraints (rules) upon the ontology. Notice that these constraints should: (a) capture the notion of consistency in the standard sense (e.g., that every class is a subclass of the top class for the RDF case) and (b) encode the semantics of the various constructs of the underlying language (RDF in our case), which are not carried over during the transition to FOL (e.g., IsA transitivity) [2]. The

Table 3: Indicative Consistency Rules

Rule ID/Name	Integrity Constraint	Intuitive Meaning
R5 $C.IsA$ Applicability	$\forall x, y: C.IsA(x, y) \rightarrow CS(x) \wedge CS(y)$	Class IsA applies between classes
R6 $P.IsA$ Applicability	$\forall x, y: P.IsA(x, y) \rightarrow PS(x) \wedge PS(y)$	Property IsA applies between properties
R12 $C.IsA$ Transitivity	$\forall x, y, z: C.IsA(x, y) \wedge C.IsA(y, z) \rightarrow C.IsA(x, z)$	Class IsA is Transitive
R14 $P.IsA$ Transitivity	$\forall x, y, z: P.IsA(x, y) \wedge P.IsA(y, z) \rightarrow P.IsA(x, z)$	Property IsA is Transitive
R22 Class IsAs and Top	$\forall x: CS(x) \rightarrow C.IsA(x, \top)$	All classes are subclasses of the top node

latter type of constraints is very important, in the sense that it forces an ontology to contain all its implicit knowledge as well in order to be consistent.

Table 3 contains an indicative list of the rules we use for the case of RDF; these rules are based on the model described in [8] (see also [6] for a similar effort). Notice that the constraints presented are only a parameter of the model; our framework does not assume any particular set of constraints (in the same sense that it does not assume any particular ontology representation language).

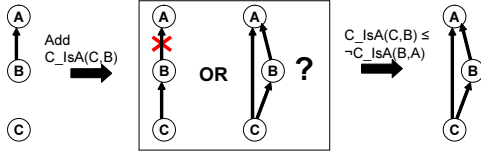
3.2 Inconsistency Resolution and Ordering

As already mentioned, the raw application of an update would guarantee success but could often violate consistency (i.e., it could violate an integrity constraint). For example, under the consistency context of Table 3, the class deletion in Fig. 1 would violate rule R5. In such cases, we need to determine the various options that we have in order to resolve the inconsistency.

The formalization of the consistency model using rules has the important property that, apart from detecting inconsistencies, it also provides a straightforward methodology to determine the various available options for resolving them. In effect, the rules themselves and the FOL semantics indicate the appropriate side-effects to be taken when an inconsistency is detected. In the above case, rule R5 implies that, in order to restore consistency after the removal of class B (per the update and the principle of success), we have to delete the IsAs that involve B .

Table 4: Ordering of predicates

$$PI < C_Inst < P_IsA < C_IsA < \neg PI < \neg C_Inst < \neg P_IsA < \neg C_IsA < \neg Domain < \neg Range < \neg CI < \neg PS < \neg CS < Domain < Range < CI < PS < CS$$


Figure 3: Adding an IsA

Notice that the detected side-effects are updates themselves, so they are enforced upon the ontology by being executed along with the original update; moreover, they could, just like any update, cause additional side-effects of their own. Another important remark is that, in the above case, the inconsistency resolution mechanism gave a straightforward result, in the sense that we only had one option to break the inconsistency (i.e., remove the IsAs); however, in certain cases, we may have more than one alternative options.

In the cases where we have different alternative sets of side-effects to select among, a mechanism to determine the “best” option, according to some metric, should be in place. In section 2.3, we showed that our “preference” among the side-effects can be encoded using an ordering; given such an ordering (say $<$), all we need to do is find the minimal set of side-effects (with respect to $<$) and implement it. As usual, our framework does not depend on any particular ordering; however, the ordering employed should be total and should always have a minimum. These requirements guarantee that, for any set of potential reactions (i.e., for any collection of sets of side-effects), we can always find a minimum element (i.e., a set of side-effects that is minimal with respect to $<$).

For our RDF case the ordering is based on the ordering shown in Table 4 among the 18 positive and negative predicates. This ordering is expanded to refer to updates (i.e., sets of ground facts) using the general idea that an update U_1 is “preferable” or “better” than U_2 (denoted by $U_1 < U_2$) iff the “worst” predicate used in update U_1 , is “better” than the “worst” predicate used in update U_2 where the predicates’ relative preference is determined by the order shown in Table 4. Ties are resolved using cardinality considerations and/or the relative importance of the predicate’s arguments in the original ontology. Further details are omitted due to space limitations.

3.3 Evolution Algorithm

We will now show how one can use the above framework in order to develop an evolution algorithm (cf. table 5). Let us consider the update example of Fig. 3. Our original update is $U = \{C_IsA(C,B)\}$, denoting that an IsA between C and B should be added. We first need to check whether this update will violate any rule (line 2.1); this can be easily done by checking against all rules in which C_IsA (or $\neg C_IsA$) is involved.

In general, more than one rules may be violated, in which case we process them in arbitrary order (line 2.2). In this

Table 5: General Algorithm

Input: Model, Rules, Ordering $<$, Update U , Ontology K
REPEAT
 (1) Select (arbitrarily) an unprocessed predicate in U , say P
 (2.1) IF there is no rule violated by P , THEN mark P as processed, add P to the side-effects of U and return
 (2.2) ELSE select (arbitrarily) one violated rule, say R
 (2.2.1) FOR each possible way to resolve the violation of R , add the respective predicates as side-effects in U and recursively call the algorithm using the new U
 (2.2.2) When recursion returns compare (using $<$) the returned side-effects and return the “best” to the caller
 UNTIL no unprocessed predicates exist
Output: Update U enriched with its side-effects

case, it can be verified that the addition of $C_IsA(C,B)$ will only violate rule R12 (IsA transitivity), for $x = C$, $y = B$, $z = A$. This is true because the addition of that IsA fires the transitivity rule so the implicit knowledge ($C_IsA(C,A)$) should be added as well. This option is the standard way of satisfying transitivity, but rule R12 also gives us the alternative to remove the old IsA between B and A (this alternative simply prevents the transitivity rule from firing).

In order to explore all alternatives regarding the possible side-effects, the comparison (using $<$) between the first and the second option is postponed until the full set of side-effects has been computed. Therefore, at this point, the algorithm suggests two different alternative updates, one per side-effect, namely $U_1 = \{C_IsA(C,B), C_IsA(C,A)\}$ and $U_2 = \{C_IsA(C,B), \neg C_IsA(B,A)\}$ (line 2.2.1). Then, the algorithm recursively calls itself twice (once for U_1 and once for U_2). Both calls will indicate no further side-effects, as there are no further rules violated; in the general case, the side-effects could have side-effects of their own, so the recursion should continue until no further side-effects exist. Once all recursions stop, the returned sets of side-effects are compared using $<$ and the minimal is selected for implementation (line 2.2.2). In this case, the first option (i.e., U_1) is the “best” option according to $<$ (see table 4), i.e., the IsA between C and A should be added; this indeed sounds like the most natural result, but it could be different if the ordering was different.

Notice that the general algorithm (table 5) is applicable for any language (i.e., ontology model), consistency model and ordering and that several details of the algorithm have been brushed out. One such detail, for example, is the recognition and rejection of an infeasible update. Our algorithm’s complexity depends on its parameters, namely the language, consistency model and ordering; for the particular parameters used for RDF, termination can be guaranteed.

A downside of the generality enjoyed by this algorithm is that it is not efficient. To remedy this problem, we can develop simpler, special-purpose algorithms, for the particular application that we are interested at (RDF in our case). These “instantiations” are much faster than the general algorithm, but can still be proven equivalent to it, i.e., formally sustained. Thus, we can guarantee that they exhibit the expected/desired behavior, by verifying them against the general-purpose algorithm above. Notice that these special-purpose algorithms are similar to ad-hoc methodologies employed by other systems but without resorting to the tedious and error-prone case-based reasoning usually employed for this purpose. Moreover, the general algorithm could still be used to implement any possible operation, beyond these

Table 6: Special Purpose Algorithm: Remove Class

Remove class A:

- (1) If class A is in K THEN
 - (1.1) Remove all class IsA relationships deriving from A.
 - (1.2) Remove all class IsA relationships arriving in A.
 - (1.3) Remove all instantiation links between a resource and A.
 - (1.4) FOR every property P whose range/domain is A
 - (1.4.1) Remove all property IsA relationships deriving from P.
 - (1.4.2) Remove all property IsA relationships arriving in P.
 - (1.4.3) Remove all instantiation links of P.
 - (1.4.4) Remove P and the information on its range/domain.
- (1.5) Remove A.

specific solutions.

Table 6 shows, as an example, one such special purpose algorithm used for removing a class *A* from an ontology *K*. Notice that some lines in this algorithm ((1.4.1)-(1.4.4)) would spawn another special purpose algorithm for executing each removal (thus, possibly, incurring further side-effects). Similar algorithms have been developed for other operations, but are omitted due to space limitations.

4. SUMMARY

In this paper, we identified several difficulties associated with the development of ad-hoc ontology evolution algorithms. We decomposed the process of coping with ontology evolution into 5 discrete steps. This way, devising an ontology evolution algorithm is reduced to the process of instantiating each step in a modular way. To this end, we presented a formal framework with the aid of which an evolution algorithm can be materialized as a set of adequate parameterizations which are the following:

1. The ontology representation model and its mapping to FOL.
2. The definition of the allowed change operations in the model. Notice that this is not necessary, as the framework is general enough to support any update, but we may, for some reason, want to disallow certain operations for some application.
3. The consistency rules that allow us to detect inconsistencies as well as to determine how the inconsistencies can be resolved.
4. The preference ordering among side-effects that encodes the selection mechanism.

Parameters 1,2 and 4 of our framework correspond to steps 1,2 and 5 respectively. The third parameter corresponds to the consistency context, based on which our framework instantiates steps 3 and 4. Once these parameters are set, we can apply the general algorithm presented in Table 5 to perform any change. For efficiency reasons, it may be useful to generate simpler and more efficient special purpose algorithms based on the general one. This can be done only for specific instantiations of the above parameters, as in the case study of RDF updating presented here. This case study set the proper parameter values for the RDF model, allowing us to propose a particular special purpose algorithm for RDF updating which evidently features the desired properties.

Our method exhibits a consistent behavior with respect to the various choices involved, regardless of the particular ontology or update operation at hand. It has a formal foundation, issuing a solid, consistent, transparent and customizable method to handle any type of change operation, including updates that have not been considered at design time. Our framework is modular in the sense that it could work with any language, rules and/or ordering given.

We are currently implementing our RDF change algorithm in the context of the FORTH-ICS Semantic Web Knowledge Middleware (SWKM), which provides generic services for acquiring, refining, developing, accessing and distributing community knowledge. In the future, we plan to incorporate optimization techniques that could speed up the most common change operations applied upon an RDF KB, as well as to verify the effectiveness of our proposed ordering using experiments with real users.

5. ACKNOWLEDGEMENTS

This work was partially supported by the EU projects CASPAR (FP6-2005-IST-033572) and KP-Lab (FP6-2004-IST-4).

6. REFERENCES

- [1] M. Dalal. Updates in propositional databases. Technical Report DCS-TR-222, Department of Computer Science, Rutgers University, 1988.
- [2] G. Flouris. On the Evolution of Ontological Signatures. *Proceedings of the Workshop on Ontology Evolution*, 2007.
- [3] T. Gabel, Y. Sure, and J. Voelker. KAON-ontology management infrastructure. *SEKT informal deliverable*, 3(1).
- [4] C. Goble, D. McGuinness, R. Moller, and P. Patel-Schneider. OilEd a reasonable ontology editor for the semantic web. *Description Logics Workshop*.
- [5] M. Klein and N. Noy. A component-based framework for ontology evolution. *Workshop on Ontologies and Distributed Systems at IJCAI*, 2003.
- [6] S. Munoz, J. Perez, and C. Gutierrez. Minimal deductive systems for rdf. In *Proceedings of the 4th European Semantic Web Conference*, 2007.
- [7] N. Noy, R. Ferguson, and M. Musen. The knowledge model of Protégé-2000: Combining interoperability and flexibility. *Lecture Notes in Artificial Intelligence (LNAI)*, 1937:17-32.
- [8] 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.
- [9] L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-driven ontology evolution management. *Proceedings of the 13th European Conference on Knowledge Engineering and Knowledge Management EKAW*, 2002.
- [10] Y. Sure, J. Angele, and S. Staab. OntoEdit: Multifaceted Inferencing for Ontology Engineering. *Journal on Data Semantics*, 1(1):128-152, 2003.