Exploring RDF/S Evolution using Provenance Queries

Haridimos Kondylakis
Dimitris Plexousakis
Institute of Computer Science, FORTH
Vasilika Vouton, Heraklion
Greece
kondylak@ics.forth.gr
dp@ics.forth.gr

ABSTRACT

The evolution of ontologies is an undisputed necessity in current research community. The problem of understanding this evolution is a fundamental problem as, based on this understanding, maintainers of depending artifacts need to take a decision about possible changes. Moreover, as ontologies are often developed by several ontology engineers, it is also important for them to understand what changes have been made by each other. Recent research focuses on just identifying and presenting the changes from one ontology version to another. In this paper, we argue that this is not enough and that we need more fine-grained methods for understanding how the ontology evolved. To this direction, we present a module, named *ProvenanceTracker,* which gets as input the list of changes between two or more RDF/S ontology versions and can answer fine-grained provenance queries about ontology resources. Our module can identify *when* a resource was created and *how*. The sequence of changes that led to the creation of that specific resource can be identified and presented to the user. We evaluate the time complexity of our approach and show that it can possibly reduce the human effort spent on understanding ontology evolution.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms

Algorithms, Experimentation, Languages, Theory.

Keywords

Ontology Evolution, Provenance

1. INTRODUCTION

Ontologies are defined as formal, explicit specification of a shared conceptualization of a domain of interest [1]. However ontologies are not static but they are living artifacts and subject to change [2]. Due to the rapid development of research, ontologies are frequently changed to depict the new knowledge that is acquired. Since ontologies are usually managed independently from each other they can be used and extended without the explicit permission of the owner. In several cases, the owner of an ontology is completely unaware of who uses or extends his ontology.

It is therefore vital to be able to support the ontology engineers and the maintainers of the dependent artifacts in this complex process of ontology evolution [3]. Several approaches so far deal with problems such as consistency maintenance, backward compatibility, ontology manipulation, change propagation, etc. [2]. In the field of *a posteriori* understanding ontology evolution most of the approaches use different representation languages to model ontology evolution [4] that they just present to the users. However, although the languages of changes used have become more concise and compact - by employing high-level change operators (operators that can describe complex updates, e.g. the insertion of an entire sub-sumption hierarchy) - still ontology understanding relies on just presenting to the users a huge list of changes between ontology versions.

In this paper, we argue that only listing the changes between two versions is insufficient for the purpose of understanding ontology evolution. Moreover, we provide a solution to this problem by answering provenance queries concerning both the data and the schema information of an ontology. To that direction, we present a module, named *ProvenanceTracker* that gets as input two or more ontology versions and it is able to answer queries requesting fine-grained provenance information. In order to do that, a preprocessing step is required that automatically generates the "on-the-fly" the sequence of changes between those versions. This is accomplished by employing an external module, described extensively in [4], which gets as input subsequent ontology versions and produces automatically the sequence of changes between them.

In our approach we define the notions of *how* and *when provenance* and we present the corresponding algorithms. Using our module a user can identify with which change operation a resource was introduced (*how*) and in which ontology version (*when*). Moreover, the list of change operations that led to the creation of that specific resource can be computed and presented to the user (*extended-how*) allowing further exploration. This knowledge can be used to drive developer's understanding on ontology evolution for that specific resource. The simplicity of our approach makes it a valuable tool for ontology engineers and provides a unique vantage point on long and complex evolution histories.

Finally, we describe our implementation and we present our experimental analysis using two well-known ontologies CIDOC-CRM [5] and Gene Ontology [6]. Experiments performed show the feasibility of our approach and the considerable advantages gained.

The rest of the paper is organized as follows: Section 2 presents related work and Section 3 provides preliminaries and introduces the problem by an example. Then, Section 4 presents the

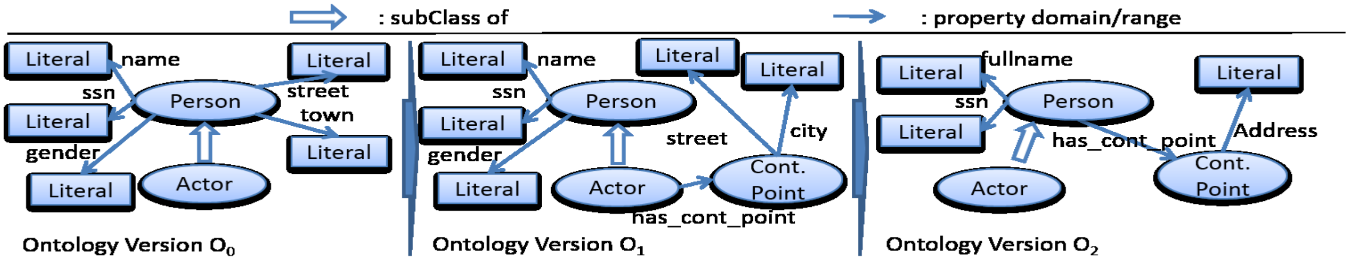


Figure 1. Example Ontology Evolution

algorithms for answering provenance queries about ontology evolution. Section 5 describes the implemented system and our experimental analysis. Finally, Section 6 provides a summary and an outlook for further research.

2. RELATED WORK

Management of provenance information has been extensively studied in the literature, using different methods and approaches. Different authors define different provenance management techniques (e.g. *Why*-provenance [7], *Trio*-Provenance [8], provenance *semi-rings* [9]) that either try to provide annotations at the tuple level or to extract provenance information by analysing queries. Other works such as [10] try to describe the relationship between source and target data in a data integration scenario. However our approach differs in both methods and goals. To the best of our knowledge, there is no other approach that tries to answer provenance queries on ontology evolution.

Other works that could be employed to understand ontology evolution focus on change detection. Those systems can be classified under two basic dimensions, namely the level of changes they support (low-level or high-level) and the underlying representation language assumed (Description Logic [3], RDF/S [4] etc.). In its simplest form, a language of changes consists of only two *low-level* operations, *Add(x)* and *Delete(x)*, which determine individual constructs (e.g., triples) that were added or deleted [11, 12]. However, a significant number of recent works [4, 12-15] imply that *high-level* change operations should be employed instead, which describe more complex updates, as for instance the insertion of an entire subsumption hierarchy. A high-level language is preferable than a low-level one [16], as it is more *intuitive*, *concise*, *closer to the intentions* of the ontology editors and *captures more accurately* the semantics of change. However, there is no agreed-upon list of changes that are necessary for any given context. In our case, we do not redefine such a language but we only use one of them. Moreover, our results are not limited to this specific language as we shall see later in this paper.

A similar approach to ours, is in [3] where a change is defined and detected using temporal queries over a version log that contains recordings of the applied changes. However, the version log must be updated whenever a change occurs. This overrules the use of this approach in non-curated or distributed environments. In our approach, on the other hand, the changes can be produced a posteriori and no temporal queries are used.

In [17] the authors provide a mechanism to document schema evolution of relational DBs, by presenting automatically the changes (called SMOs) between those versions. However, those changes are not detected fully automatically. Moreover, they offer a schema evolution history analysis tool, but this tool only provides coarse-grained results.

Finally, in [18] the authors present a tool to allow several developers to make changes concurrently and remotely to the same ontology, track changes, and manage ontology versions. However, this tools focuses on conflict resolution and cannot provide answers to fine-grained provenance queries.

3. PRELIMINARIES & MOTIVATING EXAMPLE

RDF is a language for describing web resources [19]. Information in RDF is represented using triples of the form (*subject, predicate, object)* which record that *subject* is related to *object* via *predicate*. RDF datasets have attached semantics through RDFS schemas [19]. RDFS is a vocabulary description language that includes a set of inference rules use to generate new, implicit triples from explicit ones. Most of the Semantic Web Schemas (85,45%) are expressed in RDF/S [20] and RDF/S offers, in our case, an optimum trade-off between expressive power and efficient reasoning support. In this paper we restrict ourselves to *valid RDF/S knowledge bases*. The validity constraints [4] that we consider in this work concern mostly the *type uniqueness,* i.e., that each resource has a unique type, the *acyclicity* of the *subClassOf* and *subPropertyOf* relations and that the subject and object of the instance of some property should be correctly classified under the domain and range of the property, respectively. Those constraints are enforced in order to enable *unique* and *non-ambiguous* detection of the changes among the ontology versions as we shall later discuss.

Now as an example, consider an ontology shown on the left of Figure 1 (ontology version *O_0*). This ontology is used as a point of common reference, describing people and their contact points. Assume now that at some point in time, the ontology evolves and we get *O_1* by adding the class "*Cont.Point*" describing contact points and the property "*has_cont_point*" between the class "*Actor*" and the class "*Cont.Point*". Moreover, the domain of the literals "*street"* and "*city*" is changed to the class "*Cont.Point*". Then the ontology designer decides to evolve again the ontology and to produce *O_2*. So, the domain of the "*has_cont_point*" property is moved from the class "*Actor*" to the class "*Person*", and the property "*gender*" is deleted. Moreover, the "*street*" and the "*city*" properties are merged to the "*address*" property as shown on the right of Figure 1. For modeling this evolution we use the language of changes and the corresponding detection algorithm as proposed in [4]. The language contains over 70 types of change operations and three of them are described in Figure 2.

Change	***Generalize_Domain(a,b,c)***	***Rename_Property(a,b)***	***Merge_Properties(A,b)***
Intuition	Change the domain of prop. *a* from *b* to superclass *c*	Rename property *a* to *b*	Merge properties contained in *A* into *b*
δ_a	*[(a, domain, c)]*	*[(b, type, property)]*	*(b, type, property)*
δ_d	*[(a, domain, b)]*	*[(a, type, property)]*	$\forall a \in A$ *: (a, type, property)*
Inverse	*Specialize_Domain(a,c,b)*	*Rename_Property(b,a)*	*Split_Property(b, A)*

Figure 2. Example change operations

A change operation is defined as follows:

Definition 3.1 (Change Operation): *A change operation u over an RDF ontology O, is any tuple (δ_a, δ_d) where $\delta_a \cap O = ø$ and $\delta_d \subseteq O$. A change operation u from O_1 to O_2 is a change operation over O_1 such that $\delta_a \subseteq O_2 \backslash O_1$ and $\delta_d \subseteq O_1 \backslash O_2$.*

Obviously, δ_a and δ_d are sets of triples. For simplicity we will denote $\delta_a(u)$ the *added* and $\delta_d(u)$ the *deleted* triples of a change *u*. From the definition, it follows that $\delta_a(u) \cap \delta_d(u) = ø$ and $\delta_a(u) \cup \delta_d(u) \neq ø$ if $O_1 \neq O_2$. The application of a change *u* over an ontology version *O*, denoted by *u(O)*, is defined as $u(O) = (O \cup \delta_a(u)) \backslash \delta_d(u))$. Moreover the application of a sequence of change operations *us* to an ontology, i.e. *us(O)*, is defined as the sequential application of the change operation in *us* to *O*. An important note for those change operations is that for any two changes u_1, u_2 in such a sequence it holds that $\delta_a(u_1) \cap \delta_a(u_2) = ø$ and $\delta_d(u_1) \cap \delta_d(u_2) = ø$. As a consequence the sequence of changes between two ontology versions is unique. The interested reader is forwarded to [4] for more information on the aforementioned language of changes.

In our example the change log between O_0 and O_1, i.e. the E^{O_0,O_1}, consists of the following change operations:

u1: Add_Class(Cont.Point, ø, ø, ø, ø, ø,)
u2:Add_Property(has_cont_point, ø, ø ,ø ,ø, Actor, Cont.Point, ø, ø)
u3: Move_Property(town, Person, Cont.Point))
u4: Move_Property(street, Person, Cont.Point))
u5: Rename_Property(town, city)

Moreover, the change log E^{O_1,O_2} consists of the following change operations:

u6: Delete_Property(gender, ø, ø ,ø ,ø, Person, xsd:String, ø, ø)
u7: Generalize_Domain(has_cont_point, Actor, Person)
u8: Merge_Properties({street, city},address)
u9: Rename_Property(name, fullname)

Obviously, $E^{O_0,O_2} = E^{O_0,O_1} \cup E^{O_1,O_2}$. In this paper we argue that only presenting the above sequence of changes is not enough for understanding how ontology evolved. Especially in real world scenarios, the large number of change operations makes it impossible for ontology developers to understand ontology evolution based solely on those. In our experiments for example, we had 4175 changes for Gene Ontology and 726 changes for CIDOC-CRM.

To overcome this problem we designed and implemented the *ProvenanceTracker* module. This module augments the understanding of ontology evolution by answering a range of provenance queries, including the following ones: How was a resource added to the ontology? By which change operation was the "Address" literal added? What are the change operations that had some influence on the creation of the "Address" literal? When was the "*Address*" literal added to the ontology?

Similar terminology [21] is widely used in relational environments. However, to the best of our knowledge it is the first time that we use this terminology to capture provenance information on ontology evolution. Moreover, although our ontology and change operations can be used on instance level as well, in this paper we will focus only on schema level without loss of generality.

4. QUERIES ON SCHEMA PROVENANCE

As already mentioned, presenting only the list of changes between ontology versions in not adequate for understanding ontology evolution. To answer queries about how a specific resource was added we define the notion of an affecting change operation.

Definition 4.1 (Affecting Change Operation). *Let r be a resource of an ontology version O_m and E^{O_k,O_m} (k<m) the sequence of changes between O_k and O_m. A change operation $u \in E^{O_k,O_m}$ affects the resource r, denoted by aff(r), if $r \in O_m$ and $r \in \delta_a(u)$.*

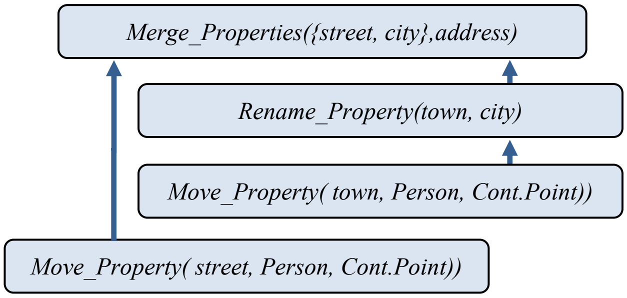
An affecting change operation captures the way a resource was introduced in the ontology. Assuming that we have E^{O_k,O_m} already constructed it is quite easy to identify the affecting change operation by just scanning the change log once. We have to note that the affecting change operation if it exists is *unique*. This is due to the fact that for our languages of changes it holds the following: for any two changes u_1, u_2, $\delta_a(u_1) \cap \delta_a(u_2) = ø$ and $\delta_d(u_1) \cap \delta_d(u_2) = ø$ as described in Section 2. In our example the query *how("Address")* when applied in E^{O_0,O_2} it will return the change operation *u8:Merge_Properties({street, city},address)*. In the case that no affecting change operation is found, no answer will be returned. This means that the aforementioned resource was added before O_k and we have no information about it.

Now we would like to know in which ontology version the "*Address*" resource was introduced. The idea is similar to answering *how* provenance queries. We only have to scan once the change log $E^{O_k,O_m} = E^{O_k,O_{k+1}} \cup \ldots \cup E^{O_{m-1},O_m}$. If $aff(r) \in E^{O_k,O_m}$ then obviously $r \in O_m$ so we can conclude that *r* was introduced in O_m. In our example the query *when("Address")* will return O_2 as an answer.

Presenting only the affecting change operations and the ontology version that a resource has been introduced does not necessarily provide insights on the corresponding ontology evolution. When drastic evolution occurs, those are not enough and we would like to get more information about which part of the ontology evolved to produce the specific resource. So, instead of providing just the affecting change operation and the ontology version, our idea is to present the history of the evolution of the specific parts of the ontology as an answer to *extended-how* provenance queries.

For example, by checking the change log E^{O_0,O_2} presented on Section 3, we can easily identify that the operations shown in Figure 3, describe the evolution of the "*address*" resource.

Presenting such a graph to the ontology engineers, their understanding on the ontology evolution is focused on the specific parts that evolved to produce the aforementioned resource. Such a sequence of change operations that depict the history of the ontology with respect to a specific resource *r* is called a *change path* for that resource. However, before defining the change path for a given resource we will define the change path for a *change operation* first.



Merge_Properties({street, city},address)
Rename_Property(town, city)
Move_Property(town, Person, Cont.Point))
Move_Property(street, Person, Cont.Point))

Figure 3. The change path for the "Address" resource visualized as a tree.

Definition 4.2 (Change path for a change operation). *A change path for the change operation $u \in E^{Ok,Om}$, (k<m) denoted by us_{path}^{u}, is the minimal sequence of change operations in $E^{Ok, Om}$ such that $u \in us_{path}^{u}$ and that us_{path}^{u} (O_k) $\subseteq O_m$.*

A change path is *minimal* in the sense that one cannot remove any of the change operations in it and still us_{path}^{u} (O_k) $\subseteq O_m$. The change path presents the history of the evolution of the specific part of the ontology for a specific change operation. For example, the *change path* for the change *u8: Merge_Properties({street, city},address)* is $us_{path}^{u8} =[u_3, u_4, u_5, u_8]$ as shown in Figure 3 and us_{path}^{u8} (O_0) $\subseteq O_2$.

Proposition 2 (Uniqueness**):** *The change path us_{path}^{u} over $E^{Ok,Om}$ is unique.*

Proof: Assume us_{path}^{u} is not unique. This would mean that we can have two change paths us_{path1}^{u} and us_{path2}^{u}. Since they are both change paths it should hold that $size(us_{path1}^{u})=size(us_{path2}^{u})$ since they both have to be minimal. Now let $us_{path1}^{u} = [u_{k1}, ..., u_{kn}]$ and $us_{path1}^{u} = [u_{m1}, ..., u_{mn}]$. Since they are both change paths $u= u_{kn}= u_{mn.}$ For *i*<n, each one of the u_{ki}, u_{mi} deletes a part of the ontology and adds another part. Since the two change paths have the same minimal size and $u= u_{kn}= u_{mn}$ in order to be different there must exist two change operations u_{ki} , u_{mj} such that $u_{ki} \neq u_{mj}$ and $\delta_d(u_{ki}) \cap \delta_d(u_{mi}) \neq ø$ since they should delete a common part of the ontology. However, this is impossible since $\delta_d(u_1) \cap \delta_d(u_2)= ø$ for our change operations▪

```
Algorithm 4.1: ComputeChangePath($E^{Ok,Om}$, $u$)
Input: A sequence $E^{Ok,Om}$= [$u_1$, ...,$u_n$ ] and one change operation $u$
Output: a sequence of change operations $us'$
1. $us' := u$
2. For $i=n$ to $1$
3.     if there exists $t \in \delta_d(u_i)$ such that $t \in \delta_a(us')$
4.         $us' := us' \cup u_i$
5. Return $us'$
```

Figure 4. Computing the change path for a given change operation.

Now, we will present an algorithm that given a change log produces the change path for a change operation *u*. The algorithm is shown in Figure 4. The idea is the following: The algorithm starts from the input change operation and identifies the triples that are added to the ontology, possibly by deleting other triples. Then it searches for the change operations that delete that added information in order to add new information and so on. After the execution the change path for *u* will be stored in *us'*.

Theorem 1: *The algorithm ComputeChangePath computes us_{path}^{u} over $E^{Ok,Om}$.*

Proof: In order to prove that algorithm *ComputeChangePath* computes the change path for a given change operation *u* over a change log $E^{Ok,Om}$ we have to prove that (a) $u \in us'$, (b) us' (O_1) $\subseteq O_2$ and that (c) *us'* is minimal.

(a) From line 1 of the algorithm indeed $u \in us'$.

(b) Let's assume that us' (O_k) is not a subset of O_m. This would mean that there exists at least one triple, assume t' in us' (O_k) such that it does not exist in O_m. So, to reach O_k, there should be a change operation u' such that $t' \in \delta_d(u')$ such that $t' \in \delta_a(us')$ not identified by our algorithm. However this is impossible from line 3 of our algorithm.

(c) Now we prove minimality. Let's assume that *us'* is not minimal. This would mean that there is us_{path} with $size(us_{path})< size(us')$. This would mean that there exist $u' \in us'$ such that $u' \notin us_{path}$. Of course this would mean from lines 3 and 4 that there exist t' such that $t' \in \delta_d(u')$ and $t' \in \delta_a(us')$. However, this would mean that t' does not belong to $O_{m,}$ and should be deleted by another change operation. However for our change operations it holds that $\delta_d(u_1) \cap \delta_d(u_2)= ø$ which makes the previous statement impossible. So *us'* is minimal as well▪

The time complexity of the algorithm is $O(N*M*S)$ where *N* is the number of change operations, *M* the maximum size of triples in a change operation *u* and *S* the number of triples in $\delta_a(us')$. Moreover, it is easy to change Algorithm 4.1 in order to retrieve the change path for a given *resource*. This will allow the developers to examine the evolution of the ontology concerning a specific resource:

Definition 4.3 (Change path for a resource). *The change path us_{path} over $E^{Ok,Om}$ for the resource $r \in O_m$ is $us_{path}^{r}= \cup us_{path}^{ui}$, $r \in u_i$.*

The algorithm is shown in Figure 5. The idea is that we would like to retrieve the history of the evolution of resource *r*. However, *r* might appear in several triples so we need to identify all change paths that have to do with it.

Theorem 2: *The algorithm ComputeChangePathTriple computes the change path for a given resource r over $E^{Ok,Om}$.*

```
Algorithm 4.2: ComputeChangePathResource ( $E^{O1,O2}$, r)
Input: A sequence $E^{O1,O2}$= [$u_1$,…,$u_n$ ] and one resource $r$
Output: a sequence of change operations $us'$
1. $us' := ø$
2. For $i=n$ to 1
3.     If $t \in \delta_a(u_i)$ such that $r \in t$
4.         $us' := us' \cup ComputeChangePath(E^{O1,O2}, u_i)$
5. Return $us'$
```

Figure 5. Computing the change path for a given resource.

The algorithm is immediately proved by construction. Algorithm 4.2 needs to scan the change log one time per triple containing the resource *r* in order to identify the change operation that inserts the given resource. So the complexity of the algorithm becomes O(*T***N***M***S*) where *T* is the number of triples containing *r, N* is the number of change operations, *M* the maximum size of triples in a change operation *u* and *S* the number of triples in $\delta_a(us')$.

5. IMPLEMENTATION & EVALUATION

The *ProvenanceTracker* module described in this paper is implemented as a module of our *Exelixis* plarform (http://139.91.183.29:8080/exelixis/). The platform uses JAVA for the algorithms and HTML/JQuery for the presentation layer. Using the *Exelixis* platform, a user is able to load an RDF/S ontology, to visualize and explore it. Furthermore, as more ontology versions become available, the change logs between them are automatically constructed and stored to the system. Then, a user can issue queries - denoting the ontology version that those queries are using- which are being forwarded to the underlying data integration engines to be answered. The system automatically identifies registered data integration systems that might use different ontology versions and tries to produce equivalent query rewritings for them. If this is not possible, the reasons for this are reported and approximate query rewritings are used. The theory behind query answering can be found in [22] whereas a demo of the core components was presented at [23]. The module for automatically generating the sequence of changes among two ontology versions was presented at [4] whereas [24] and [25] report on other modules that try to respond to massive number of queries that might need to be changed by producing possible rewritings as well.

In order to evaluate the algorithms reported in this paper, we used a workstation with an Intel Core i7 processor running at 3.4 Ghz, and 4GB memory, using Windows 7x64. Moreover, we used two well-known ontologies: One medium-size ontology (CIDOC-CRM [6]) from the cultural domain which is rarely changed and one large-size ontology (Gene Ontology [6]) from the bioinformatics domain which is heavily updated daily.

CIDOC-CRM is an ISO standard which consists of nearly 80 classes and 250 properties. For our experiments we used versions dated from 02.2002 (v3.2.1) to 06.2005 (v4.2). The detected change log that was automatically produced identified 726 total changes from v3.2.1 to v.4.2. Gene Ontology (GO) on the other hand, is composed of about 28000 classes and 1350 property instances. GO is updated on a daily basis and for our experiments we used the change log from 25.11.08 to 26.05.09. The change log that was automatically produced contained 4175 changes.

5.1 Answering provenance queries

Next, we present experiments concerning the scalability of the algorithms for answering provenance queries. We measured the average execution time for computing answers to *how*/*when* and *extended-how* provenance queries. To do that we exhaustively queried for *how/when* and *extended-how* all resources in the latest ontology versions and the results are presented in Figure 6 and Fig. 7.

As shown in the figures, for both ontologies the average time to produce a change path increased linear to the number of changes we had to search. This is in line with the complexity of our algorithms as we presented previously. Moreover, the time to compute answers to *extended-how* provenance queries is greater than computing answers to *how*/*when* queries. This is reasonable since in the first case more triples are being added to the list of triples that we are looking for in the sequence of changes.

However, we can see that the overhead for searching the added triples in the change path has little impact in the total execution time since the dominant factor is the number of change operations. So, for CIDOC-CRM after 726 change operations we only need 275 msec in average to compute *how/when* provenance whereas for *why* provenance we need 280 msec. On the other hand, for Gene Ontology after 4175 changes we need 4611 msec for *how/when* queries and 4967 msec for *extended-how* queries.

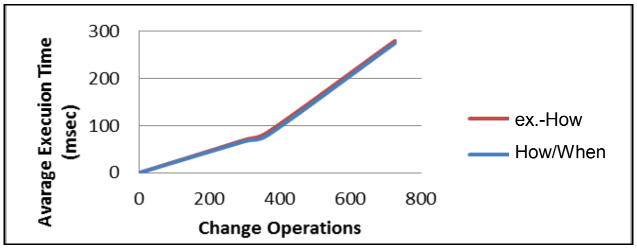


Figure 6. The average execution time compared to the number of changes for CIDOC-CRM

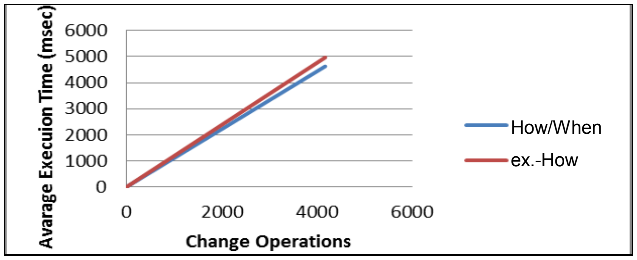


Fig. 7. The average execution time compared to the number of changes for Gene Ontology

Obviously, the time to compute a change path is greater for the Gene Ontology than for CIDOC-CRM. This is reasonable since for the Gene Ontology we have to search 4175 changes, whereas for CIDOC-CRM we only have to search 726 changes.

Moreover, we‘ve identified that the biggest number of changes in a change path in the case of Gene Ontology was 8 whereas for CIDOC-CRM it was 5. So, independent of the number of changes between ontology versions the interested user needs to check at most 8 change operations (including change operations in comments) to understand how the specific part of the ontology has been evolved. We have to note here that the average number of change operations that a user had to examine was 2 for CIDOC-CRM and 4 for GO which shows the added value of our approach even for ontologies that change often.

Finally, trying to understand the provenance queries, we made several interesting observations. One of them for example, was the following: We identified that in the evolution of the CIDOC-CRM ontology from version v3.2.1 to version v3.3.2, one ontology engineer renamed the class "*E11 Modification*" to "*E11 Modification Event*". A few years later another ontology engineer was employed to evolve the ontology. So in v4.2 we can see that the class "*E11 Modification Event*" was again renamed to "*E11 Modification*". If the second ontology engineer had an indication of the previous renaming he would avoid cycles, he would be able to identify possibly the reasons behind each renaming since we are also able to show comments from the ontology evolution. So, using provenance queries to explore ontology evolution can be a valuable tool reducing greatly the time spent on understanding evolution.

6. CONCLUSION & DISCUSSION

In this paper, we argue that ontology evolution is a reality and that the problem of understanding ontology evolution is a fundamental problem in the area. Ontology engineers should have proper tools to help them understand the choices of the past. To that direction, we presented a novel module that assists ontology evolution as the reality that ontology model changes.

Instead of just identifying and presenting the changes from one ontology version to another, our module can answer fine-fine grained provenance queries for a specific resource. It can identify *when* a resource was created, *how* it was introduced and it can present the change operations that lead to the creation (or deletion) of that resource and its evolution history. This greatly minimizes the total time for understanding ontology evolution. Experiments performed, show the potential impact of our approach. For example, for CIDOC-CRM provenance answers can be retrieved at most within 280 msec and for GO at most within 5sec even if there are more than 4000 changes that have to be examined. Moreover, ontology engineers have to examine at most 5 change operations for CIDOC-CRM and 8 change operations for GO to understand how the ontology evolved.

We need to note that we selected the specific language of changes for several reasons. One of them is because it is a high-level language of changes as already described in Section 2. Moreover, the language possesses nice properties such as *uniqueness*, *composition* and *inversion*. *Uniqueness* is a pre-requisite for our system whereas *composition* and *inversion* are desirable but not obligatory properties. So, instead of the specific language of changes other languages (and the corresponding detection algorithm) could be also used as long as they preserve *uniqueness.*

As future work, several challenging issues need to be further investigated. An interesting topic would be to extend our approach for OWL ontologies. Another interesting topic would be to present summaries of the evolved change path if they become too big. Ontology evolution is becoming more and more important topic and several challenging issues remain to be investigated in near future.

7. ACKNOWLEDGMENTS

This work was partially supported by the PlanetData NoE (FP7:ICT-2009.3.4, #257641), the eHealthMonitor (FP7-287509) and the MyHealthAvatar (FP7-600929) EU projects.

8. REFERENCES

[1] Gruber, T.R. 1993. A translation approach to portable ontology specifications. *Knowl. Acquis*. 5, pp. 199-220.

[2] Flouris, G., Manakanatas, D., Kondylakis, H., Plexousakis, D., Antoniou, G.: Ontology change, 2008. Classification and survey. *Knowl. Eng. Rev*. 23, pp. 117-152.

[3] Plessers, P., Troyer, O.D., Casteleyn, S. 2007. Understanding ontology evolution: A change detection approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5, pp. 39-49.

[4] Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V. 2013. High-Level Change Detection in RDF(S) KBs. *Transactions on Database Systems*, 38.

[5] Doerr, M., Ore, C.-E., Stead, S. 2007. The CIDOC conceptual reference model: a new standard for knowledge sharing. *Tutorials, posters, panels and industrial contributions at the ER*, vol. 83, pp. 51-56.

[6] Gene Ontology Consortium, 2004. The Gene Ontology (GO) database and informatics resource. *Nucl. Acids Res.* 32, D258-261.

[7] Buneman, P., Khanna, S., Tan, W.C. 2001. Why and Where: A Characterization of Data Provenance, *ICDT*, pp. 316-330.

[8] Benjelloun, O., Sarma, A.D., Halevy, A., Theobald, M., Widom, J. 2008. Databases with uncertainty and lineage. *The VLDB Journal*, 17, pp. 243-264.

[9] Green, T.J., Karvounarakis, G., Tannen, V. 2007. Provenance semirings. *ACM SIGMOD-SIGACT-SIGART PODS*. pp. 31 - 40 ACM, Beijing, China

[10] Chiticariu, L., Tan, W.-C. 2006. Debugging schema mappings with routes. *VLDB* , pp. 79-90.

[11] Volkel, M., Winkler, W., Sure, Y., Kruk, S.R., Synak, M. 2005. Semversion: A versioning system for rdf and ontologies. *ESWC*.

[12] Zeginis, D., Tzitzikas, Y., Christophides, V. 2007. On the Foundations of Computing Deltas Between RDF Models. *ISWC/ASWC*, pp. 637-651.

[13] Noy, N.F., Chugh, A., Liu, W., Musen, M.A. 2006. A Framework for Ontology Evolution in Collaborative Environments *ISWC*, pp. 544-558.

[14] Plessers, P., Troyer, O.D. 2005. Ontology Change Detection Using a Version Log. *ISWC*, pp. 578-592.

[15] Rogozan, D., Paquette, G. 2005. Managing Ontology Changes on the Semantic Web. *IEEE/WIC/ACM International Conference on Web Intelligence*, pp. 430-433.

[16] Stojanovic, L. 2004. Methods and Tools for Ontology Evolution. *Phd*. Univ. of Karlsruhe.

[17] Curino, C., Moon, H., Deutsch, A. and Zaniolo, C. 2013. Automating the database schema evolution process. *The VLDB Journal*, 22, 1, pp. 73-98.

[18] Jim, E., Ruiz, N., Grau, B. C., Horrocks, I. and Berlanga, R. 2011. Supporting concurrent ontology development: Framework, algorithms and tool. *Data Knowl. Eng.*, 70, 1, pp. 146-164.

[19] RDF Primer, W3C Recommendation: http://www.w3.org/TR/rdf-primer/

[20] Theoharis, Y., 2007. On Graph Features of Semantic Web Schemas. *IEEE Transactions on Knowledge and Data Engineering*, 20, pp. 692-702.

[21] Cali, A., Gottlob, G., Lukasiewicz, T. 2009. Datalog^{+-}: a unified approach to ontologies and integrity constraints. *ICDT*, pp. 14-30. ACM, St. Petersburg, Russia.

[22] Kondylakis, H., Plexousakis, D. 2013. Ontology evolution without tears. *Journal of Web Semantics: Science, Services and Agents on the World Wide Web*, 19, pp. 42-58.

[23] Kondylakis, H., Plexousakis, D. 2011. Exelixis: Evolving Ontology-Based Data Integration System. *SIGMOD*, pp. 1283-1286.

[24] Kondylakis, H., Plexousakis, D. 2011. Ontology Evolution in Data Integration: Query Rewriting to the Rescue. *ER*, pp. 393-401.

[25] Kondylakis, H., Plexousakis, D. 2012. Ontology Evolution: Assisting Query Migration. *ER*, vol. 7532, pp. 331-344