Ontology Evolution without Tears
Haridimos Kondylakis, Dimitris Plexousakis
Information Systems Laboratory, FORTH-ICS
N. Plastira 100, Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece

ABSTRACT
The evolution of ontologies is an undisputed necessity in ontology-based data integration. Yet, few research efforts have focused on addressing the need to reflect the evolution of ontologies used as global schemata onto the underlying data integration systems. In most of these approaches, when ontologies change their relations with the data sources, i.e., the mappings, are recreated manually, a process which is known to be error-prone and time-consuming. In this paper, we provide a solution that allows query answering in data integration systems under evolving ontologies without mapping redefinition. This is achieved by rewriting queries among ontology versions and then forwarding them to the underlying data integration systems to be answered. To this purpose, initially, we automatically detect and describe the changes among ontology versions using a high level language of changes. Those changes are interpreted as sound global-as-view (GAV) mappings, and they are used in order to produce equivalent rewritings among ontology versions. Whenever equivalent rewritings cannot be produced we a) guide query redefinition or b) provide the best “over-approximations”, i.e., the minimally-containing and minimally-generalized rewritings. We prove that our approach imposes only a small overhead over traditional query rewriting algorithms and it is modular and scalable. Finally, we show that it can greatly reduce human effort spent since continuous mapping redefinition is no longer necessary.

Keywords
Ontology Evolution, Data Integration, Query Rewriting.

1. INTRODUCTION
The development of new scientific techniques and the emergence of new high throughput tools have led to a new information revolution. The nature and the amount of information now available open directions of research that were once in the realm of science fiction. During this information revolution the data gathering capabilities have greatly surpassed the data analysis techniques, making the task to fully analyze the data at the speed at which it is collected a challenge. The amount, diversity, and heterogeneity of that information have led to the adoption of data integration systems in order to manage it and further process it. However, the integration of these disparate data sources raises several semantic heterogeneity problems.

By accepting an ontology as a point of common reference, naming conflicts are eliminated and semantic conflicts are reduced. Ontologies are used to identify and resolve heterogeneity problems, usually at schema level, as a means for establishing an explicit formal vocabulary to share. During the last years, ontologies have been used as global schemata in database integration [1], obtaining promising results, for example in the fields of biomedicine and bioinformatics [2, 3]. When using ontologies to integrate data, one is required to produce mappings, to link similar concepts or relationships from the ontology/ies to the sources by way of an equivalence. This is the mapping definition process [4] and the output of this task is the mapping, i.e., a collection of mappings rules. In practice, this process is done manually with the help of graphical user interfaces and it is a time-consuming, labour-intensive and error-prone activity [5].

Despite the great amount of work done in ontology-based data integration, an important problem that most of the systems tend to ignore is that ontologies are living artifacts and subject to change [4]. Due to the rapid development of research, ontologies are frequently changed to depict the new knowledge that is acquired. The problem that occurs is the following: when ontologies change, the mappings may become invalid and should somehow be updated or adapted.

In this paper, we address the problem of data integration for evolving RDF/S ontologies that are used as global schemata. We address the problem for a core subset of SPARQL queries that correspond to union of conjunctive queries. We argue that ontology change should be considered when designing ontology-based data integration systems. A typical solution would be to regenerate the mappings and then regenerate the dependent artifacts each time the ontology evolves. However, as this evolution might happen too often, the overhead of redefining the mappings each time is significant. The approach, to recreate mappings from scratch each time the ontology evolves, is widely recognized to be problematic [5-7], and instead, previously captured information should be reused. However, all current approaches that try to do that suffer from several drawbacks and are inefficient [8, 9] in handling ontology evolution in a state of the art ontology-based data integration system. The lack of an ideal approach leads us to propose a new mechanism that builds on the latest theoretical advances on the areas of ontology change [10] and query rewriting [11, 12] and incorporates and handles ontology evolution efficiently and effectively. More specifically:

Corresponding Author: Haridimos Kondylakis
Tel:+30-2810-391499,
Fax: +30-2810-391428,
E-mail: kondylak@ics.forth.gr
We present the architecture of a data integration system, named Evolving Data Integration system, that allows the evolution of the ontology used as global schema. Query answering in our system proceeds in two phases: a) query rewriting from the latest to the earlier ontology versions and b) query rewriting from one ontology version to the local schemata. Since query rewriting to the local schemata has been extensively studied [11-13], we focus on a layer above and deal only with the query rewriting between ontology versions.

The query processing in the first step consists of: i) query expansion that considers constraints coming from the ontology, and ii) valid query rewriting that uses the changes between two ontology versions to produce rewritings among them.

In order to identify the changes between the ontology versions we adopt a high-level language of changes. We show that the proposed language possesses salient properties such as uniqueness, inversibility and composability. Uniqueness is a pre-requisite for the solution described in this paper, where the other two properties are nice to have, but they are not necessary for our solution. The sequence of changes between the latest and the other ontology versions is produced automatically at setup time and then those changes are translated into logical GAV mappings. This translation enables query rewriting by unfolding. Moreover, the inversibility is exploited to rewrite queries from past ontology versions to the current, and vice versa, and composability to avoid the reconstruction of all sequences of changes among the latest and all previous ontology versions.

Despite the fact that query rewriting always terminates, the rewritten queries issued, using past ontology versions, might fail. We show that this problem is not inhibiting in our algorithms but a consequence of information unavailability among ontology versions. To tackle this problem, we propose two solutions: a) either to provide best “over-approximations” by means of minimally-containing and minimally-generalized queries, or b) to provide insights for the failure by means of affecting change operations, thus driving query redefinition.

We show that our method is sound and complete and does not impose a significant overhead. Finally, we present our experimental analysis using two real-world ontologies. Experiments performed show the feasibility of our approach and the considerable advantages gained.

Such a mechanism, that provides rewritings among data integration systems that use different ontology versions as global schemata, is flexible, modular and scalable. It can be used on top of any data integration system – independently of the family of the mappings that each specific data integration system uses to define mappings between one ontology version and the local schemata (GAV, LAV, GLAV [13]). New mappings or ontology versions can be easily and independently introduced without affecting other mappings or other ontology versions. Our engine takes the responsibility of assembling a coherent view of the world out of each specific setting.

This paper is an extended and revised version of a previously published conference paper [14] whereas the implemented system was demonstrated in [15]. However, only the basic ideas were described in [1], without a detailed analysis of the theoretical foundation of the approach. This manuscript adds to the previously published results, the related work, the formal properties of the language of changes used to capture ontology evolution and the specific semantics of the implemented architecture. In addition, the new algorithms that were created are presented, their correctness is proved and their complexity is analyzed. Finally, an evaluation of the system is presented for the first time using real and synthetic set of queries, and a discussion is added to the conclusion of this paper.

The rest of the paper is organized as follows: Section 2 introduces the problem by an example and presents related work. Section 3 presents the architecture of our system and describes its components. Section 4 describes the semantics of such a system and Section 5 elaborates on the aforementioned query rewriting among ontology versions. Finally, Section 6 presents our experimental analysis and Section 7 provides a summary and an outlook for further research.

2. MOTIVATING EXAMPLE & RELATED WORK

Consider the example RDF/S ontology shown on the left of Figure 1. This ontology is used as a point of common reference, describing persons and their contact points (“Cont.Point’). We also have two relational databases DB1 and DB2 mapped to that version of the ontology. Assume now that the ontology designer decides to move the domain of the “has_cont_point” property from the class “Actor” to the class “Person”, and to delete the property “gender”. Moreover, the “street” and the “city” properties are merged to the “address” property. Merging is a concatenation with some special character like comma between the words. Furthermore, the “name” property is renamed to “fullname” as shown on the right of Figure 1. Then, one new database DB3 is mapped to the new version of the ontology leading to two data integration systems that work independently. In such a setting we would like to issue queries formulated using any ontology version available. Moreover, we would like to retrieve answers from all underlying databases.

![Figure 1. The motivating example of an evolving ontology](image-url)

Several approaches have been proposed so far to tackle similar problems. For example, for XML databases there have been several approaches that try to preserve mapping information under changes [16] or propose guidelines for XML schema evolution in order to maintain the mapping information [17]. Moreover, augmented schemata were introduced in [18] to enable query
answering over multiple schemata in a data warehouse, whereas other approaches change the underlying database systems to store versioning and temporal information such as [19-24]. However, our system differs from all the above in terms of both goals and techniques.

Other works focus on the problem of updating RDF/S [25-27] or OWL-DL [28] knowledge bases. These works mostly try to determine the effects and side-effects of elementary or complex change operations and to characterize the different class of updates with a well-defined semantics. In our work, however, we do not deal with the effects of the change operations on the ontology but we assume that the ontology versions are directly given. Moreover, we use a language with well-defined semantics in order to identify a-posteriori the changes that have happened to the ontology.

The most relevant approaches that could be employed for resolving the problem of data integration with evolving ontologies is mapping adaptation [5] and mapping composition and inversion [9].

In mapping adaptation [5] the main idea is that schemata often evolve in small, primitive steps; after each step the schema mappings can be incrementally adapted by applying local modifications. However, this approach is integration system-dependent, and is not specified in which way the list of changes might be discovered when two schema versions are directly provided. But even when such a list of changes can be obtained, applying the incremental algorithm for each change and for each mapping in this potentially long list will be highly inefficient. Another problem is that multiple lists of changes (by introducing redundant additions/deletions for example) may have the same effect of evolving the old schema into a new one [6]. Finally, there is no guarantee that after repeatedly applying the algorithm, the semantics of the resulting mappings will be the desired ones. This happens because complex evolution might happen, that cannot be modeled with simple additions and deletions, and dependencies might be lost. In order to tackle these problems we use a more expressive language of changes that leads to unique sequence of changes between two ontology versions with reduced size compared to the long list of low-level operators. Moreover, the initial semantics of the provided mappings are maintained since we do not change the mappings but instead we rewrite the queries.

A more general formalization of the mapping adaptation problem is through mapping composition and inversion [6, 9]. The approach would be to describe ontology evolution as mappings and to employ mapping composition/inversion to derive the adapted mappings. However, mapping composition proved to be a difficult problem and mapping inversions a more difficult one. In [29] it was shown that no first-order language is closed under composition and second-order mappings should be used instead, whereas the identification of a language closed under both inversion and composition is still an open problem [9]. An exact inverse may not exist and several notions of “approximations” of inverses have been lately developed such as quasi-inverses [30], maximum recoveries [31], chase-inverses [9] etc. A recent system that tries to build on composition and quasi-inverse schema mappings is PRISM [32]. However, the sequence of changes among two versions is not unique and disambiguation is needed in several places by domain experts. Moreover, the composed mappings might be too difficult for domain experts to grasp and understand (they are second-order mappings). Our approach avoids the constant involvement of domain experts since continuous mapping redefinition is no longer necessary. The changes among the ontologies are produced automatically. Moreover, instead of composing all mappings each time, in our case they are kept intact in order to be verified and updated by domain experts.

To the best of our knowledge no system today is capable of retrieving information mapped with different ontology versions.

3. EVOLVING DATA INTEGRATION

We conceive an Evolving Data Integration system as a collection of data integration systems, each one of them using a different ontology version as global schema. Therefore, we extend the traditional formalism from [13] and define an Evolving Data Integration system as:

Definition 3.1 (Evolving Data Integration System): An Evolving Data Integration system \( I \) is a tuple of the form \((O_1, S_i, M_i), ..., (O_m, S_m, M_m))\) where:

- \( O_i \) is a version of the ontology used as global schema,
- \( S_i \) is a set of local sources and
- \( M_i \) is the mapping between \( S_i \) and \( O_i, (1 \leq i \leq m) \).

Next we discuss how the specific components are specialized in the context of an Evolving Data Integration system.

3.1 Global and Local Schemata

Considering \( O \), we restrict ourselves to valid RDF/S knowledge bases, as most of the Semantic Web Schemas (85.45%) are expressed in RDF/S [33].

The representation of knowledge in RDF [34] is based on triples of the form (subject predicate object). Assuming two disjoint and infinite sets \( U, L \), denoting the URIs and literals respectively, \( T = U \times U \times (U \cup L) \) is the set of all triples. An RDF Graph \( V \) is defined as a set of triples, i.e., \( V \subseteq T \). In this paper, we ignore unnamed resources, also called blank nodes. RDF/S [35] introduces some built-in classes (class, property) which are used to determine the type of each resource. The typing mechanism allows us to concentrate on nodes of RDF graphs, rather than triples, which is closer to ontology curators’ perception and useful for defining intuitive high-level changes.

RDFS provides also inference semantics, which is of two types, namely structural inference (provided mainly by the transitivity of subsumption relations) and type inference (provided by the typing system, e.g., if \( p \) is a property, the triple \( (p, \text{type}, \text{property}) \) can be inferred). The RDF Graph containing all triples that are either explicit or can be inferred from explicit triples in an RDF Graph \( V \) (using both types of inference), is called the closure of \( V \) and is denoted by \( Cl(V) \). An RDF/S Knowledge Base (RDF/S KB) \( V \) is an RDF Graph which is closed with respect to type inference, i.e., it contains all the triples that can be inferred from \( V \) using type inference. Moreover, we assume that the RDF/S Knowledge Bases are valid. The notion of validity has been described in various fragments of the RDFS language. The validity constraints that we consider in this work concern 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
Several languages with such high-level change operations exist, addition and deletion of the same triple to happen at the same time, be required to reach this state. Moreover, we have to note that several change operations might be employed instead, which describes more complex updates, as it's non-ambiguous detection of the changes among the ontology versions.

In its simplest form, a language of changes that describes how an ontology version was derived from another ontology version. 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 [38, 39]. Such a language is called a low-level language of changes. However, a significant number of recent works [10, 39, 40] 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 (they group individual additions and deletions).

A high-level language is preferable than a low-level one [41], as it is more intuitive, concise, closer to the intentions of the ontology editors and captures more accurately the semantics of change. For example, the high-level change operation "Rename_Property(fullname, name)" is more informative the following set of low-level operations "Delete(fullname, type, property), Add(name, type, property)". As we shall see later on, a high-level language is beneficial for our problem for two reasons: First, because the produced change log has a smaller size and is more important because such a language yields logs that contain a smaller number of individual low-level deletions (which are non-information preserving) and this affects the effectiveness of our rewriting. In our work, a change operation is defined as follows:

Definition 3.2 (Change Operation): A change operation $u$ over $O$, is any tuple $(\delta_a, \delta_d)$ where $\delta_a \cap O = \emptyset$ and $\delta_d \subseteq O$. A change operation $u$ from $O_1$ to $O_2$ is a change operation over $O_1$ such that $\delta_a \subseteq O_1 \cap O_2$ and $\delta_d \subseteq O_1 \cap O_2$.

Obviously, $\delta_a$ and $\delta_d$ are sets of triples. For simplicity we will denote $\delta_a(u)$ the added and $\delta_d(u)$ the deleted triples of a change $u$. Moreover, we have to note that several change operations might be required to reach $O_2$ from $O_1$, and that’s why $\delta_a \subseteq O_2 \cap O_1$ and $\delta_d \subseteq O_2 \cap O_1$ and not $\delta_a = O_2 \cap O_1$ and $\delta_d = O_2 \cap O_1$. We are interested for the changes that $\delta_a(u) \cap \delta_a(u) = \emptyset$ and $\delta_d(u) \cup \delta_d(u)$ = $\emptyset$. The first condition guarantees that no change operation would require the addition and deletion of the same triple to happen at the same time. The second condition guarantees that at least something should be added or deleted.

Several languages with such high-level change operations exist [10, 39, 40]. However, in order to be able to use such a language for query rewriting, as we shall see in the sequel, it is necessary the sequence of changes among two ontology versions to be unique. Composition and inversion are desirable but not obligatory properties that enhance the quality of the solution proposed. Such a high-level language of changes and the corresponding detection algorithm is presented in [10]. It contains over 70 types of change operations and the complete list can be found on [42]. The definitions of the change operations used in this paper are shown in Figure 2. Hereafter, whenever we refer to a change operation, we mean a change operation from those proposed in [10]. Now we will shortly describe the main properties of that language (see [36] for more information and proofs).

Definition 3.3 (Application semantics of a high-level change): The application of a change $u$ over $O$, denoted by $u(O)$, is defined as: $u(O) = (O \cup \delta_a(u)) \setminus \delta_d(u)$.

One key observation here is that the application of our change operations is not conditioned by the current state of the ontology. Moreover, for any two changes $u_1$, $u_2$ in such a sequence it holds that $\delta_a(u_1) \cap \delta_a(u_2) = \emptyset$ and $\delta_d(u_1) \cap \delta_d(u_2) = \emptyset$ and as a result:

Proposition 1: The sequence of change operations from $O_1$ to $O_2$, denote by $E^{O_2, O_1}$, as detected by the corresponding algorithm in [10] is unique.

It is shown that those change operations compose indeed. By using composition we will be able to use the intermediate evolution logs between ontology versions instead of constructing all change logs between the latest ontology version and all past ontology versions.

Definition 3.4 (Composition of Change Operations): A change operation $u_1 \cdot u_2$ is the composition of $u_1$ and $u_2$ (computed over $O_1$ and $O_2$), if the result of applying $u_1 \cdot u_2$ on $O_1$ is the same with the result of applying in sequence $u_1$ and $u_2$ on $O_1$ in any order.

$u_1 \cdot u_2(O_1) = u_1(u_2(O_1)) = u_2(u_1(O_1))$

Proposition 2: Let $u_1$, $u_2$ two change operations from $O_1$ to $O_2$. Then $u_1 \cdot u_2 = (\delta_a(u_1) \cup \delta_a(u_2), \delta_d(u_1) \cup \delta_d(u_2))$

Finally, the inverse of a change operation can always be found. By automatically constructing the inverse of a sequence of change operations (from $O_1$ to $O_2$), we will be able to rewrite queries expressed using $O_2$ to $O_1$ and vice versa.

Definition 3.5 (Inverse of a change operation): Let $u$ be a change operation from $O_1$ to $O_2$. A change operation $u_{inv}$ from $O_2$ to $O_1$ is the inverse of $u$ if: $u_{inv}(u(O_1)) = O_1$

Proposition 3: The inverse of a change operation $u$ (denoted by $inv(u)$) from $O_1$ to $O_2$ is: $inv(u) = (\delta_a(u), \delta_d(u))$

Corollary 1: The inverse of a sequence of change operations $E^{O_2, O_1} = [u_1, ..., u_n]$ constructed from $O_1$ to $O_2$ is $inv(E^{O_2, O_1}) = [inv(u_1), ..., inv(u_n)]$

In our example the change log $E^{O_2, O_1}$, consists of the following change operations:

$u_1$: Rename_Property(fullname, name)
$u_2$: Split_Property(address, {street, city})
$u_3$: Specialize_Domain(has_cont_point, Person, Actor)
$u_4$: Add_Property(gender, $\emptyset$, $\emptyset$, $\emptyset$, Person, xsd: String, $\emptyset$, $\emptyset$)
It is obvious, that applying those change operations on $O_2$, results $O_1$. Moreover, the inverse of the sequence of change operations for our running example is the following:

<table>
<thead>
<tr>
<th>Change</th>
<th>Intuition</th>
<th>$inv(u_4)$: $Delete_Property(gender, \phi, \phi, Person, xsd: String, \phi, \phi)$</th>
<th>$inv(u_3)$: $Generalize_Domain(has_cont_point, Actor, Person)$</th>
<th>$inv(u_2)$: $Merge_Properties(street, city, address)$</th>
<th>$inv(u_1)$: $Rename_Property(name, full_name)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Add_Property(a,P1,P2,P3,P4,p5,p6,P7,P8)$</td>
<td>Add property $a$ with its neighborhood links. $P1$ is the set of new parent properties of $a$, $P2$ is the set of properties that have as parent $a$, $P3$ is the set of new meta-properties of $a$, $P4$ is the set of new property instances of $a$, $p5$ is the new domain of $a$, $p6$ is the new range of $a$, $P7$ is the set of new comments of $a$, $P8$ is the set of new labels of $a$</td>
<td>$\forall p \in P1 : (a, sub_PropertyOf, p)$, $\forall p \in P2 : (p, sub_PropertyOf, a)$, $\forall p \in P3 : (a, type, p)$, $\forall p \in P4 : (p, a, p2)$, $\forall p \in P5 : (p1, a, p2)$, $\forall p \in P6 : (a, range, p6)$, $\forall p \in P7 : (a, comment, p)$, $\forall p \in P8 : (a, label, p)$, (a, type, property)</td>
<td>Delete $Property(a,P1,P2,P3,P4,p5,p6,P7,P8)$</td>
<td>Inv.</td>
<td>GAV</td>
</tr>
<tr>
<td>$Delete_Property(a,P1,P2,P3,P4,p5,p6,P7,P8)$</td>
<td>Delete property $a$ with its neighborhood links. $P1$ is the set of new parent properties of $a$, $P2$ is the set of properties that have as parent $a$, $P3$ is the set of new meta-properties of $a$, $P4$ is the set of new property instances of $a$, $p5$ is the new domain of $a$, $p6$ is the new range of $a$, $P7$ is the set of new comments of $a$, $P8$ is the set of new labels of $a$</td>
<td>$\forall p \in P1 : (a, sub_PropertyOf, p)$, $\forall p \in P2 : (p, sub_PropertyOf, a)$, $\forall p \in P3 : (a, type, p)$, $\forall p \in P4 : (p, a, p2)$, $\forall p \in P5 : (p1, a, p2)$, $\forall p \in P6 : (a, range, p6)$, $\forall p \in P7 : (a, comment, p)$, $\forall p \in P8 : (a, label, p)$, (a, type, property)</td>
<td>Add $Property(a,P1,P2,P3,P4,p5,p6,P7,P8)$</td>
<td>Inv.</td>
<td>GAV</td>
</tr>
<tr>
<td>$Rename_Property(a,b)$</td>
<td>Rename property $a$ to $b$</td>
<td>(b, type, property)</td>
<td>Rename $Property(b,a)$</td>
<td></td>
<td>GAV</td>
</tr>
<tr>
<td>$Split_Property(a,B)$</td>
<td>Split property $a$ into properties contained in $B$</td>
<td>$\forall b_i \in B : (b_i, type, property)$ (1 ≤ $i$ ≤ $n$)</td>
<td>$Merge_Properties(B, a)$</td>
<td>$\forall x, y, a(x, y) \rightarrow b(x, y)$</td>
<td>GAV</td>
</tr>
<tr>
<td>$Merge_Properties(A, b)$</td>
<td>Merge properties contained in $A$ into $b$</td>
<td>$\forall a_i \in A : (a_i, type, property)$ (1 ≤ $i$ ≤ $n$)</td>
<td>Split $Property(b,A)$</td>
<td></td>
<td>GAV</td>
</tr>
<tr>
<td>$Generalize_Domain(a,b,c)$</td>
<td>Change the domain of property $a$ from $b$ to a superclass $c$</td>
<td>(a, domain, c)</td>
<td>$Specialize_Domain(a, c, b)$</td>
<td></td>
<td>GAV</td>
</tr>
<tr>
<td>$Specialize_Domain(a,b,c)$</td>
<td>Change the domain of property $a$ to a subclass of it</td>
<td>(a, domain, b)</td>
<td>$Generalize_Range(a,c,b)$</td>
<td></td>
<td>GAV</td>
</tr>
</tbody>
</table>

Figure 2. The change operations used in this paper
We have to note that in the general case neither split nor merge change operations are irreversible without a matcher, that will be called when applying the “split” function (in the GAV mappings).

In our case we implemented a simple matcher that employs heuristic-based techniques for matching the correct parts of the instance of a class/property that needs to be split. Moreover, our design was modular to allow any custom-made off-the-self matcher to identify the required matchings. However, the focus of this paper is not on developing a sophisticated matcher, but on query rewriting so we will not elaborate more on this topic.

Another useful comment here is that instead of the specific query rewriting we will not elaborate more on this topic.

4. SEMANTICS OF AN EVOLVING DATA INTEGRATION SYSTEM

Now we will define semantics for an Evolving Data Integration system I. Our approach is similar to [13] and is sketched in Figure 3. We start by considering a local database for each (O_i, S_i, M_i), i.e., a database D_i that conforms to the local sources of S_i. For example, D_i is a local database for (O_i, S_i, M_i) that conforms to the local sources S_{1i}, S_{2i} and S_{3i}.

Now, based on D_i we shall specify the information content of the global schema O_i. We call a database for O_i a global database. However, since those global databases might be many, we are interested in the legal global databases.

![Figure 3. The semantics of an Evolving Data Integration system](image)

**Definition 4.1** (Legal Global Database): A global database G_i for (O_i, S_i, M_i) is said to be legal with respect to D_i if

- G_i satisfies all subClass/subProperty constraints of O_i
- G_i satisfies the mapping M_i with respect to D_i

The notion of G_i satisfying the mapping M_i, with respect to D_i, is defined as it is commonly done in traditional data integration systems (see [13] for more details). It depends on the different assumptions that can be adopted for interpreting the tuples that D_i assigns to relations in local sources with respect to tuples that actually satisfy (O_i, S_i, M_i). Since such systems have been extensively studied in the literature we abstract from the internal details and focus on the fact that for each (O_i, S_i, M_i) of our system we can obtain several legal global databases G_i.

Now, we can repeat the same process, i.e., to consider the legal global databases as sources and a database D which we will simply call the global database, the database that conforms to them. Now we can similarly define the total databases (databases for O_w) and the legal total databases. We use the term “total” only to differentiate it from a global database, since we will extensively use it from now on.

**Definition 4.2** (Legal Total Database): A total database T for I is said to be legal with respect to D if

- T satisfies all subClass/subProperty constraints of O_w
- T satisfies the evolution log E = \( \bigcup_{m \in \text{om}} E^m \) wrt. D

Now we specify the notion of T satisfying the evolution log E with respect to D. In order to exploit the strength of the logical languages towards query reformulation, we convert our change operations into logical GAV mappings. So, when we refer to the notion of T satisfying E, we mean T satisfying the GAV mappings produced from E. The GAV mappings for some of the change operations used in this paper can be found in Figure 2. In relational databases a GAV mapping associates a table from the target schema to a query over the source schemata. So, in our case a GAV mapping associates to a class/property g in T a query q_{g} over the other ontology versions G1, ..., G_m i.e., \( g \rightarrow q_{g} \).

**Definition 4.3**: A database T satisfies the mappings g \( \rightarrow \) q_{g} with respect to D if \( g \rightarrow q_{g} \) where q_{g} is the result of evaluating the query q_{g} over D.

For example, the sequence of the GAV mappings that corresponds to our sequence of changes is:

- \( m_1: \forall x, y, \text{fullname}(x, y) \rightarrow \text{name}(x, y) \)
- \( m_2: \forall x, y, \text{address}(x, y) \rightarrow \exists y_1, y_2, \text{street}(x, y_1) \land \text{city}(x, y_2) \land \text{concat}(y_1, y_2) \)
- \( m_3: \forall x, \text{has_cont_point}(\text{Person}, x) \rightarrow \text{has_cont_point}(\text{Actor}, x) \)

For u_4 there is no GAV mapping constructed since we do not know where to map the deleted element. Now it becomes obvious that the more individual additions and deletions in our language of changes, the more change operations won’t have corresponding GAV mappings. This is why languages with “high-level” changes, i.e. changes that group together several individual additions and deletions are preferable.

By the careful separation between the legal total database T and the legal global databases G_i we have achieved the modular design of our Evolving Data Integration system and the separation between the traditional data integration semantics and the additions we have imposed in order to enable ontology evolution. Thus, our approach can be applied on top of any existing data integration system to enable ontology evolution. Moreover, we’ve managed to model ontology evolution in data integration as query answering over materialized views.

5. QUERY PROCESSING

Queries to I are posed in terms of the global schema O_w. For querying, we adopt a core subset of the SPARQL language corresponding to union of conjunctive queries [43]. We choose SPARQL since it is currently the standard query language for the semantic web and has become an official W3C recommendation. Essentially, SPARQL is a graph-matching language. Given a data source, a query consists of a pattern which is matched against, and
the values obtained from this matching are processed to give the answer. A SPARQL query consists of three parts. The pattern matching part, which includes several features of pattern matching of graphs, like optional parts, union of patterns, nesting, filtering (or restricting) values of possible matchings. The solution modifiers, which once the output of the pattern has been computed (in the form of a table of values of variables), allows to modify these values applying classical operators like projection, distinct, order, limit, and offset. Finally, the output of a SPARQL query can be of different types: yes/no answers selections of values of the variables which match the patterns, construction of new triples from these values, and descriptions of resources.

In order to avoid ambiguities in parsing, we present the syntax of SPARQL graph patterns in a more traditional algebraic way, using the binary operators UNION (U), AND (Λ), OPT, and FILTER according to [44]. Assuming the existence of an infinite set of variables Var disjoint from U, L, a SPARQL graph pattern expression is defined recursively as follows:

- A tuple from (U ∪ L ∪ Var) x (L ∪ Var) x (U ∪ L ∪ Var) is a graph pattern (a triple pattern)
- If P1 and P2 are graph patterns, then expressions (P1 ∧ P2), (P1 OPT P2) and (P1 U P2) are graph patterns.
- If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern.

A SPARQL built-in condition is constructed using elements of the set (U ∪ L ∪ Var) and constants, logical connectives, inequality symbols, the equality symbol etc. (see [43] for a complete list).

In this paper we adopt a streamlined version of the core fragment of SPARQL as presented in [44] with precise syntax and semantics. Moreover, we restrict even more the specific fragment of SPARQL since we do not consider OPT and FILTER operators which we leave for future work. The remaining SPARQL fragment corresponds to union of conjunctive queries [44] (this will not hold if we allow OPT and FILTER operations). Moreover, the application of the solution modifiers and the output is performed after the evaluation of the query. So, without loss of generality we will not present them in this paper. Continuing our example, assume that we would like to know the “ssn” and “firstname” of all persons stored on our DBs and their corresponding “address”. The SPARQL query, formulated using the latter version of our ontology is:

\[
q_1: \text{select ?SSN ?NAME ?ADDRESS where } \left\{ \begin{array}{l}
\text{?X type Person.} \\
\text{?X ssn ?SSN.} \\
\text{?X firstname ?NAME.} \\
\text{?X has_cont_point ?Y.} \\
\text{?Y type Cont.Point,} \\
\text{?Y address ?ADDRESS}\end{array}\right\}
\]

Using the semantics from [44] the algebraic representation of \(q_1\) is equivalent to:

\[
q_1: \forall \pi \exists?SSN,?NAME,?ADDRESS ( \left\{ \begin{array}{l}
\text{(?X, type, Person)} \land \\
\text{(?X, ssn, ?SSN)} \land \\
\text{(?X, firstname, ?NAME)} \land \\
\text{(?X, has_cont_point, ?Y)} \land \end{array}\right)
\]

Now we define what constitutes an answer to a query over \(O_m\). We will adopt the notion of certain answers [11, 13].

**Definition 5.1 (Certain Answers):** Given a global database \(\mathcal{D}\) for \(I\), the answer \(q^{1\mathcal{D}}\) to a query \(q\) with respect to \(I\) and \(\mathcal{D}\), is the set of tuples \(t\) such that \(t \in q^\mathcal{D}\) for every total database \(T\) that is legal for \(I\) with respect to \(\mathcal{D}\), i.e., such that \(t\) is an answer to \(q\) over every database \(T\) that is legal for \(I\) with respect to \(\mathcal{D}\). The set \(q^{1\mathcal{D}}\) is called the set of certain answers to \(q\) with respect to \(I\) and \(\mathcal{D}\).

Although certain answers are mostly used in local-as-view data integration systems, still in our case the ontological constraints may introduce incompleteness, so certain answers have to be adopted [12]. In fact, it has been shown [45], that computing certain answers to union of conjunctive queries over a set of legal total databases, corresponds to evaluating the query over a special database called canonical which represents all possible total databases legal for the data integration system and which may be infinite in general. In order to define the canonical database we first define the retrieved database.

**Definition 5.2 (Retrieved database):** If \(\mathcal{D}\) is a global database for the Evolving Data Integration system \(I\), then the retrieved total database \(\text{ret}(I, \mathcal{D})\) is the total database obtained by computing and evaluating, for every element of \(O_m\), the query associated to it by \(E\) over the global database \(\mathcal{D}\).

The query associated to an element of \(O_m\) is actually the GAV mapping produced by the GAV interpretation of the sequence of changes among ontology versions. (If no such mapping exists, the element will appear in the other ontology versions as well and we can query it).

**Definition 5.2 (Canonical database):** If \(\mathcal{D}\) is a global database for the Evolving Data Integration system \(I\), then the canonical total database \(\text{can}(I, \mathcal{D})\) is the set of retrieved total databases \(\text{ret}(I, \mathcal{D})\) that do not violate any constraint in \(O_m\).

Now, instead of trying to construct the canonical database and then evaluate the query, another approach is to transform the original query \(q\) into a new query \(\text{exp}(q)\) over the \(O_m\) (which is called the expansion of \(q\) w.r.t. \(O_m\)) such that the answer to...
exp_{\text{out}}(q)$ over the retrieved total database is equal to the answer to $q$ over the canonical database [45]. This step is performed by the “Parser/Expander” component shown in Figure 4. Now, in order to avoid building the retrieved total database we do not evaluate $exp_{\text{out}}(q)$ on the retrieved total database. Instead, we transform $exp_{\text{out}}(q)$ to a new query $\text{valid}_{\text{q}}(exp_{\text{out}}(q))$ over the global relations on the basis of $E$ and we use that query to access the underlying data integration systems. This is performed by the “Valid Rewriter” component which is also shown in Figure 4. This is a common approach in data integration under constraints that we also adopt. Below we describe the details of each aforementioned step.

5.1 Query expansion.

In this step, the query is expanded to take into account the constraints coming from the ontology. Query expansion amounts to rewriting the query $q$ posed to the ontology version $O_m$ into a new query $q_{vp}$ so that all the knowledge about the constraints in ontology has been “compiled” into $q_{vp}$.

**Definition 5.3 (Query Expansion):** Let $I$ an Evolving Data Integration system and let $q$ be a query over $O_m$. Then $q_{vp}$ is called an expanded query of $q$, i.e. $exp_{\text{out}}(q)$, w.r.t. $I$ if, for every global database $D$, $q_{vp} \equiv q_{\text{valid}}(1,2)$. Query expansion is also known as perfect rewriting. Algorithms for computing the query expansion/perfect rewriting of a query $q$ w.r.t. to a schema, have been presented in [37], [11], [46], [12] and mainly use chase/backchase algorithms [47]. In our work, we use the QuOnto system [12] in order to produce the query expansion of our initial query. Query expansion is in our case PTIME in the size of ontology and NP in the size of query. For more general classes of logic it is complete for PSPACE and 2EXPTIME as proved in [11]. Continuing our example if we expand $q_1$ we get $q_2$:

\[
q_2 := \pi_{\text{SSN}, \text{NAME}, \text{ADDRESS}} (\text{?X, type, Person}) \wedge (\text{?X, ssn, ?SSN}) \wedge (\text{?X, fullname, ?NAME}) \wedge (\text{?X, has_cont_point, ?Y}) \wedge (\text{?Y, type, Cont.Point}) \wedge (\text{?Y, address, ?ADDRESS})
\]

This is produced by considering the transitive constraint of the subclass relation among the classes “Person” and “Actor”.

5.2 Computing Valid Rewritings

Now instead of evaluating $exp_{\text{out}}(q)$ over the retrieved total database, we transform it to a new query called valid rewriting, i.e., $\text{valid}_{\text{q}}(exp_{\text{out}}(q))$. This is done, as already discussed, in order to avoid the construction of the retrieved total database.

**Definition 5.4 (Valid Rewriting):** Let $I$ an Evolving Data Integration system and let $q$ be a query over $\text{ret}(I, D)$. Then $q_{\text{valid}}$ is called a valid rewriting of $q$ w.r.t. $\text{ret}(I, D)$ if, for every global database $D$, $q_{\text{valid}}(1,2) \equiv q_{\text{valid}}(1,2)$.

When the retrieved total database is produced by GAV mappings, as in our case, query rewriting is simply performed using unfolding [12]. This is a standard step in data integration [13] which trivially terminates and it is proved that it preserves soundness and completeness [45]. Moreover, due to the disjointness of the input and the output alphabet in a change operation, each GAV mapping acts in isolation on its input to produce its output. So we only need to scan the GAV mappings once in order to unfold the query and the time complexity of this step $O(N*M)$ where $N$ is the number of change operations in the evolution log and $M$ is the number of triple patterns in the query. Now, we can state the main result of this section.

**Theorem 1 (Soundness and Completeness):** Let $I$ an Evolving Data Integration system, $q$ a query posed to $I$, $D$ a global database for such that $I$ is consistent w.r.t. $D$, and $t$ a tuple of constants of the same arity as $q$. Then $t \in q_{\text{vp}}$ if and only if $t \in \text{valid}_{\text{q}}(exp_{\text{out}}(q))$.

**Proof:** By soundness and completeness of unfolding $t \in \text{valid}_{\text{q}}(exp_{\text{out}}(q))$ if and only if $t \in q_{\text{vp}}$. Now by the soundness of the query expansion step we have that $t \in \text{valid}_{\text{q}}(exp_{\text{out}}(q))$ if and only if $t \in q_{\text{vp}}$. This proves the claim.

Continuing our example we will show how the valid rewriting of $q_1$ is constructed using unfolding steps. Each one of those steps uses one GAV mapping to replace a subgoal in the query with its definition in the mapping. So, initially the mapping $\mu_1$ is used. Recall that $\mu_1$ is produced from the $u_1$ change operation (Rename_Property(fullname, name)) that replaces the property “fullname” with the property “name”. So, the following query is produced by renaming also the “fullname” property on the query with the “name” property:

\[
q_3 := \pi_{\text{SSN}, \text{NAME}, \text{ADDRESS}} (\text{?X, type, Person}) \wedge (\text{?X, ssn, ?SSN}) \wedge (\text{?X, fullname, ?NAME}) \wedge (\text{?X, has_cont_point, ?Y}) \wedge (\text{?Y, type, Cont.Point}) \wedge (\text{?Y, address, ?ADDRESS})
\]

Then the mappings $\mu_2$ is used for replacing the “address” property with the “city” and the “street” literals. So, the following query is produced.

\[
q_4 := \pi_{\text{SSN}, \text{NAME}, \text{ADDRESS}} (\text{?X, type, Person}) \wedge (\text{?X, ssn, ?SSN}) \wedge (\text{?X, fullname, ?NAME}) \wedge (\text{?X, has_cont_point, ?Y}) \wedge (\text{?Y, type, Cont.Point}) \wedge (\text{?Y, street, ?ADR1}) \wedge (\text{?Y, city, ?ADR2}) \wedge (\text{concat(?ADDRESS, ?ADR1, ?ADR2)})
\]

Then $\mu_3$ is used. Recall that this is produced from the $u_3$ change operation (Specialize_Domain(has_cont_point, Person, Actor)) that specializes the domain of the “has_cont_point” property to the class “Actor”. So, the query $q_5$ is generated.

\[
q_5 := \pi_{\text{SSN}, \text{NAME}, \text{ADDRESS}} (\text{?X, type, Actor}) \wedge (\text{?X, ssn, ?SSN}) \wedge (\text{?X, name, ?NAME}) \wedge (\text{?X, has_cont_point, ?Y}) \wedge (\text{?Y, type, Cont.Point}) \wedge (\text{?Y, street, ?ADR1}) \wedge (\text{?Y, city, ?ADR2}) \wedge (\text{concat(?ADDRESS, ?ADR1, ?ADR2)})
\]
concat(?ADDRESS, {?ADR1, ?ADR2})

Since this is actually the union of a query with itself, the query that will be generated for $O_j$ is $q_6$.

$q_6$: $\pi_{?SSN, ?NAME, ?ADDRESS}($

\[(?X, \text{type}, \text{Actor}) \land
(?X, \text{ssn}, \text{?SSN}) \land
(?X, \text{name}, \text{?NAME}) \land
(?X, \text{has_cont_point}, \text{?Y}) \land
(?Y, \text{type}, \text{Cont.Point}) \land
(?Y, \text{street}, \text{?ADR1}) \land
(?Y, \text{city}, \text{?ADR2}) \land
\text{concat}(\text{?ADDRESS, {?ADR1, ?ADR2}})\])$

Finally our initial query will be rewritten to the union of $q_6$ (issued to the data integration system that uses $O_1$) and $q_2$ (issued to the data integration system that uses $O_2$).

Note that, $q_6$ sent to the data integration system that uses $O_1$ has encoded a function (concat) to concatenate the two literals “streets” and “city” to the literal “address”. This function should be executed in order to be able to unify the returned results with the results from $q_2$. However, this query cannot be sent as is to the data integration system that uses $O_1$ since a) SPARQL specification does not include the “concat” function and b) we don’t have access to the underlying data integration systems. That is why $q_6$ is rewritten to $q_7$ before it is sent to the data integration system that uses $O_1$.

$q_7$: $\pi_{?SSN, ?NAME, ?ADDRESS, ?ADR}$($

\[(?X, \text{type}, \text{Actor}) \land
(?X, \text{ssn}, \text{?SSN}) \land
(?X, \text{name}, \text{?NAME}) \land
(?X, \text{has_cont_point}, \text{?Y}) \land
(?Y, \text{type}, \text{Cont.Point}) \land
(?Y, \text{street}, \text{?ADR}) \land
(?Y, \text{city}, \text{?ADR})\])$

When the results are returned, the concatenation function is executed by our engine using our custom made implementation of that function on top of the data integration systems and the final results are unified. Similar strategy is followed for all GAV mappings that encode a function (split for example) and is the result of encoding heuristics when detecting the change operations among ontology versions.

### 5.3 Exploiting Composition

So far we have described a scenario where we construct the change logs $E_{O_m, O_k}$ between $O_m$ and all $O_i$ ($1 \leq i \leq m$). Then we formulate a query $q$ using the ontology version $O_m$, and we use the corresponding GAV mappings to produce and evaluate the $\text{valid}_q(\exp_{O_m}(q))$.

However, based on the composition property (Proposition 1), we could avoid the computation of all those change logs from scratch each time. Instead, of constructing $E_{O_m, O_k}$ for all $i$ ($1 \leq i \leq m$) we could only construct all $E_{O_m, O_k}^{i,j}$ ($2 \leq j \leq m$) between the subsequent ontology versions as shown in Figure 5.

**Corollary 2**: $E_{O_m, O_k} = \bigcup_{i=1}^{m-1} E_{O_k}^{i+1,0}$

*Proof:* The proof directly follows from the fact that the change operations we consider compose (Proposition 2) •

### 5.4 Exploiting Inversion

Ideally, we would also like to accept queries formulated using ontology version $O_1$ and to rewrite them to the newer ontology versions. This would be really useful since in many systems queries might be stored and we wouldn’t like to change them every time the ontology evolves. However, in order to achieve this we would have to use the inverse GAV mappings for query rewritings which are not always possible to produce. Our approach deals with the inversibility on the level of change operations and not at the logical level of the produced GAV mappings. So, instead of trying to produce the inverse of the initial GAV mappings, we invert the sequence of changes (which is always possible according to Corollary 1) and then use the inverted sequence of changes to produce the GAV mappings that will be used for query rewriting to the current ontology version. This is also shown in Figure 5 and enhances the impact of our approach.

Actually, it now becomes obvious that it is straight forward to accept a query formulated in any ontology version $O_1 (1 \leq i \leq m)$ and to get the rewritings for all ontology versions using the inverted list of changes for the $O_j$ that $j > i$.

### 5.5 Non-Information preserving changes

Despite the fact that both query expansion and unfolding always terminate in our setting, problems may occur. Consider as an example the query $q_8$ that asks for the “gender” and the “name” of an “Actor”, formulated using the ontology version $O_1$.

$q_8$: $\pi_{?NAME, ?GENDER}$($

\[(?X, \text{type}, \text{Actor}) \land
(?X, \text{name}, \text{?NAME}) \land
(?X, \text{gender}, \text{?GENDER})\])$

1 The complexity of the algorithm for constructing $E_{O_1, O_2}$ using as input $O_1$, $O_2$ is $O(\max(N_1, N_2, N_1^2))$ where $N_1$ is the size in triples of $O_1$, and $N$ is the size of their set difference between $O_1$ and $O_2$. 

---

Figure 5. Exploiting Composition & inversion

This would minimize the total construction cost\(^1\) - since the compared ontologies now have more common elements. However, we have to keep in mind that the time of constructing a sequence of changes is spent only once during system setup.

Moreover, whenever a new ontology version occurs, we can construct the change log between the new ontology version and the previous ontology version - and not all change logs from scratch. Of course, this will lead to larger sequences of change logs, but will allow the uninterrupted introduction of new ontology versions to the system.
Trying to rewrite the query \( q_b \) to the ontology version \( O_2 \) our system will first expand it. The expansion phase however, won’t change the query since there are no transitive constraints coming from the ontology for the used terms.

Then it will consider the GAV mappings produced from the inverted sequence of changes (as they have been presented at the end of the Section 3.2). So, the query \( q_b \) will be produced by unfolding using the mapping: \( \forall x, y, \text{name}(x, y) \rightarrow \text{fullname}(x, y) \),

\[
\begin{align*}
\{ & (?X, \text{type}, \text{Actor}) \land \\
& (?X, \text{fullname}, ?\text{NAME}) \land \\
& (?X, \text{gender}, ?\text{GENDER}) \}
\end{align*}
\]

However, it is obvious that the query produced will not provide any answers when issued to the data integration system that uses \( O_2 \) since the “gender” literal no longer exists in \( O_2 \). This happens because the \( \text{inv}(u) \) change operation is not \text{information preserving} change among the ontology versions. It deletes information from the ontology version \( O_1 \) without providing the knowledge that this information is transferred on another part of the ontology. This is also the reason that low-level change operations (simple triple addition or deletion) are not enough to dictate query rewriting.

Although, this might be considered as a problem, actually it is not, since if we miss the literal “gender” in version \( O_2 \) this would mean that we have no data in the underlying local databases for that literal. However the query still will fail and we need a mechanism to a) notify the user for the failure and b) provide best approximations.

5.5.1 Reasoning on queries

The first option is to notify the user that some underlying data integration systems were not possible to answer their queries and present the reasons for that. For our example query \( q_b \) our system will report that the data integration system that uses \( O_2 \) will fail to produce an answer because the literal “gender” does not exist in that ontology version. To identify the change operations that lead to such a result we define the notion of \text{affecting change operations}.

\textbf{Definition 5.5} (Affecting change operation): A change operation \( u \in E^{O_1, O_2} \) affects the query \( q \), denoted by \( u \triangleright q \), if:

\begin{enumerate}
  \item \( \delta_d(u) = \emptyset \)
  \item there exists triple pattern \( t \in q \) that can be unified with a triple of \( \delta_d(u) \).
\end{enumerate}

The first condition ensures that the operation deletes information from the ontology without replacing it with other information, thus the specific change operation is not information preserving. However, we are not interested in general for the change operations that are not information preserving. We specifically target those change operations that change the ontology part which corresponds to our query (condition 2). The algorithm for identifying the affecting change operations is shown on Figure 6.

Unification is a standard operation in logic programming. [48]. The algorithm for identifying affecting change operations is checks directly the change operations for the conditions described above. The time complexity of the algorithm is \( O(N^2M^2T) \), where \( N \) is the number of change operations in \( E^{O_1, O_2} \), \( M \) is the number of triple patterns in \( q \) and \( T \) is the maximum number of triples in the \( \delta_d(u) \) that \( u \in E^{O_1, O_2} \).

\textbf{Theorem 2:} The algorithm \text{IdentifyAffectingOperations} identifies the affecting change operations for a given query \( q \) over \( E^{O_1, O_2} \).

\textbf{Algorithm 5.1:} \text{IdentifyAffectingOperations}(q, E^{O_1, O_2})

\textbf{Input:} The query \( q \) formulated using ontology version \( O_1 \) and the the evolution log \( E^{O_1, O_2} \).

\textbf{Output:} The set of affecting change operations

1. \( S := \emptyset \)
2. For each \( u \in E^{O_1, O_2} \)
3. \( \quad \text{if } \delta_d(u) = \emptyset \text{ and } \exists t \in q, t' \in \delta_d(u) \text{ such that } t \text{ unifies } t' \)
4. \( \quad S := S \cup u \)
5. Return \( S \)

Figure 6. The algorithm for identifying affecting change operations for a query \( q \)

\textbf{Proof:} In line 2 the algorithm searches all change operations. For each one of those change operations, the algorithm checks the conditions in line 3. This immediately proves the claim.

Having defined the notion of affecting change operation we will prove the following:

\textbf{Proposition 4:} Let \( q = \cup_i q_i \). If for all \( q_i \), there exists \( u \in E^{O_1, O_2} \) such that \( u \triangleright q_i \), then valid\( (q) \) returns no answers.

\textbf{Proof:} The proof follows from the fact that if for a conjunctive query \( q \), there exists \( u \in E^{O_1, O_2} \) such that \( u \triangleright q \) then according to the Definition 5.5 the change operation will delete a part from the next version of the ontology that \( q \) still queries. Since the part of the schema that \( q \) will query would not be available in \( O_2 \), this means that the query will not return any answers. And since for all \( q_i \), there exists \( u \in E^{O_1, O_2} \) such that \( u \triangleright q_i \) this means that no subquery will return any answer.

Users can use that information in order to re-specify the input query if desired.

A question that arises is whether we could identify failures on the issued queries before the expansion phase. This would allow us to identify really fast the impact that the evolution has on the aforementioned queries. Although, we would identify the direct failures, the indirect ones (coming from the expansion of the queries) would not be identified. The case that such a mechanism would be useful would be when mappings are considered to be exact between the ontology versions, or when ontologies are interpreted as global schemata without constraints.

5.5.2 \text{Minimally-Containing Rewritings}

Besides providing an explanation for the failure, we can also provide the best “over-approximations”. The first solution here is the \text{minimally-containing rewriting}.

\textbf{Definition 5.6} (Minimally-Containing Rewriting [49]): A query \( q_{\text{mc}} \) is a \text{minimally-containing rewriting} of a conjunctive query \( q \) using a set of mappings \( E \) if and only if

\begin{enumerate}
  \item \( q_{\text{mc}} \) is a containing rewriting of \( q \) (\( q \sqsubseteq q_{\text{mc}} \)) and
  \item there exists no containing rewriting \( q' \) of \( q \) using \( E \), such that the expansion of \( q' \) contains the expansion of \( q_{\text{mc}} \).
\end{enumerate}
It is thus the “best “over-approximation” of q and it is dual to the “maximally-contained rewriting” \cite{47} which is the best “under-approximation” of q as shown on Figure 7.

Minimally-containing

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure7.png}
\caption{Maximally-contained rewriting vs. Minimally-containing rewriting}
\end{figure}

Now we will present an algorithm shown on Figure 8 and we will prove that the query that is computed by Algorithm 5.2 is indeed a minimally-containing rewriting of q, and thus it can be used in order to compute the minimally-containing rewriting of exp(q).

\begin{algorithm}
\caption{MinimallyContainingRewriting(q, E)}
\label{alg:mc}
\begin{algorithmic}
\State \textbf{Input}: q the conjunctive query formulated using ontology version \(O_1\) and \(E^{O_1,0_2}\) the sequence of change operation from \(O_1\) to \(O_2\).
\State \textbf{Output}: The minimally-containing rewriting of q or false
\State 1. \(q_{mc} := \text{valid}(q)\)
\State 2. \(A := \text{IdentifyAffectingOperations}(q_{mc}, E^{O_1,0_2})\)
\State 3. For each \(a \in A\)
\State 4. \(\text{Let } t \in q_{mc}, t' \in \delta_d(a) \text{ such that } t \text{ unifies } t'\)
\State 5. \(q_{mc} := q_{mc} \setminus \{t\}\)
\State 6. \(A := A \setminus \{a\}\)
\State 7. If \(q_{mc}\) is safe
\State 8. Return \(q_{mc}\)
\State 9. \textbf{else}
\State 10. Return false
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure8.png}
\caption{An alternative algorithm for computing minimally-containing rewritings}
\end{figure}

\textbf{Theorem 2}: MinimallyContainingRewriting(q, E) is a minimally-containing rewriting of a conjunctive query q using E.

\textbf{Proof}: In order to prove that MinimallyContainingRewriting(q, E) is a minimally-containing rewriting of q we will show that the Algorithm 5.2 is equivalent to the simplified version of the Chase/BackChase algorithm that has been proved \cite{47} to output such a rewriting. Recall that the simplified version of Chase/BackChase for a query q is the following:
\begin{enumerate}
\item Chase q and obtain the universal plan U.
\item Restrict the body of U only to the vocabulary of views obtaining a new query \(q_2\).
\item If \(q_2\) is safe, i.e., all head variables appear in the body, output \(q_2\), otherwise no containing-rewriting exists.
\end{enumerate}

The first step of the algorithm consists of a number of chase steps. In each chase step a constraint is applied to the query. Each chase step is actually one unfolding step with the difference that the head of one constraint is not replaced by the body, but it is added to the query as well. Then in the second step, according to the simplified version of Chase/BackChase, the body of U is restricted to the vocabulary of views obtaining a query \(q_2\). This step is actually the same as replacing the head of the mappings with their body. So the first two steps of the simplified chase algorithm behave exactly like the unfolding steps in our algorithm. The only difference is that in our case, several conjuncts might not be deleted in the unfolding step. However, those conjuncts are discovered using the algorithm for identifying the affected change operations (Theorem 2) and the deletion of these conjuncts is actually performed on line 4 of our algorithm. Finally, the third step of the algorithm is the same in our case as well. So our algorithm is equivalent to the simplified Chase/BackChase and returns the minimally-containing rewriting of the initial query with respect to \(E^*\).

A query is safe if all variables in the head of the query appear in the body as well. Concerning the time complexity, the algorithm first needs to unfold the query \(O(N*M)\) where N is the number of change operations in the evolution log and M is the number of sub-goals in the query) according to line 1 and then to detect the affecting change operations for the unfolded query \(O(N*UM+T)\) where \(UM\) is the number of sub-goals for the unfolded query and \(T\) is the maximum number of triples in the \(\delta_d(a)\) that \(a \in E^{O_1,0_2}\). Finally, the algorithms should search all subgoals of the unfolded query to identify the triples that unify with the affected changes and to delete them \((O(UM*T*A)\) where \(A\) is the number of the affected change operations). So the total time complexity is \(O(N*M)+(O(N*UM+T)*O(UM*T*A)\leq O(N*M)+O(N*M+T)+O(UM*T*N)\leq O(N*M)+2O(N*M*T)\leq O(N*M+T)\).

In our example the algorithm for producing the minimally-containing query would produce query \(q_{10}\) by deleting the triple pattern \((X, gender, ?GENDER)\) which is in not included in the vocabulary of \(O_2\).

However, the resulted query is not safe, since the variable ?GENDER does not exist in the query body, thus a minimally-containing rewriting cannot be produced in this case.

\subsection{5.5.3 Minimally-Generalized Rewritings}

Cases like the previous one, led us to search for another aspect of over-approximation. Our solution here is that when a change operation affects a query rewriting, we can check if there is a parent triple \(t'\) in the current ontology version which is not deleted in the next ontology version. If such a triple exists, we can ask for that triple instead, thus providing a generalized query.

\begin{definition}[Generalized query]
Let \(q\) a conjunctive query expressed using \(O_1\). We call \(q_{GEN}\) a generalized query of \(q\) over \(E^{O_1,0_2}\) iff:
\begin{itemize}
\item \(t\) is contained in \(q_{GEN}(q \subseteq q_{GEN})\) and \(u\) does not exist \(u \in E^{O_1,0_2}\) such that \(u \cap q_{GEN}\).
\end{itemize}
\end{definition}

Now we will define the notion of minimally-generalized query.

\begin{definition}[Minimally-Generalized query]
A generalized query \(q_{GEN}\) of \(q\) over \(E^{O_1,0_2}\) is called minimal if there is not \(q_{GEN}'\) such that \(q \subseteq q_{GEN}'\) and \(q_{GEN}' \subseteq q_{GEN}\).
\end{definition}
The idea of minimally-generalized query is that it is a query that can be answered on the evolved ontology version after applying the minimum number of "repairs" on the query in order to achieve that. However, in this case the repairs are applied using only the knowledge of the current ontology version and the change log. The algorithm for producing a minimally-generalized query over \( E^{O1, O2} \) for a given query \( q \) is shown in Figure 9.

The algorithm getParent is implemented by just querying a reasoner (Peller for example) and returns the first direct "parent" triple of \( t' \) if many exists (in lexicographic order). Moreover, it always terminates since the affecting change operations are finite. Our algorithm runs in \( O(A*N*M*T) \), where \( A \) is the maximum number of affecting change operations, \( M \) is the number of triple patterns in \( q \) and \( T \) is the maximum number of triples in the \( \delta_d(u) \) that \( u \in E^{O1, O2} \). Now we will prove the correctness of our algorithm.

**Algorithm 5.3: MinimallyGeneralizedQuery(q, O1, E^{O1, O2})**

**Input:** \( q \) the query formulated using ontology version \( O1 \) and \( E^{O1, O2} \) the sequence of change operations from \( O1 \) to \( O2 \).

**Output:** A minimally-generalized query of \( q \) if false

1. \( q_{mg} := q \)
2. \( A := IdentifyAffectingOperations(q_{mg}, E^{O1, O2}) \)
3. While(\( A \neq \emptyset \))
   - Let \( a \in A \)
   - Let \( t \in \delta_d(a) \cup \delta_d(a) \) such that \( t \) unifies with \( t' \in q_{mg} \)
   - parent := getParent(t)
   - If parent \( \neq \emptyset \) then
     - Replace \( t' \) with parent in \( q_{mg} \)
   - else
     - \( q_{mg} := false \)
     - break
4. \( A := IdentifyAffectingOperations(q_{mg}, E^{O1, O2}) \)
5. Return \( q_{mg} \)

**Figure 9.** The algorithm for identifying a minimally-generalized query

We have to note, that minimally-generalized queries may not be unique since a deleted property might have several superProperties. However, assuming a ordering between them (lexicographical for example) we can state the following theorem.

**Theorem 3:** The algorithm MinimallyGeneralizedQuery produces a minimally-generalized query of \( q \) over \( E^{O1, O2} \).

**Proof:** First we have to show that (a) the query produced is actually a generalized query and then that (b) the generalized query produced is minimal. Before proceeding in the proof recall that if \( q_1 \) and \( q_2 \) are two queries (of the same arity) for a schema \( S \), we say that \( q_1 \) is contained in \( q_2 \) with respect to \( S \), denoted by \( q_1 \subseteq q_2 \), if \( q_1^{S^0} \subseteq q_2^{S^0} \), i.e. the result of evaluating \( q_1 \) is a subset of the results of \( q_2 \) evaluation for every model \( MO \) of \( S \).

(a) Now in order to show that \( q_{mg} \) produced from Algorithm 5.3 is a generalized query we have to show that i) it does not exist \( u \in E^{O1, O2} \) such that \( u \not\vDash q_{mg} \) and ii) that \( q \subseteq q_{mg} \). Indeed from line (line 3) we remove each time one affecting change operation until \( A = \emptyset \). So if the algorithm finishes (and \( q_{mg} \neq false \) there would not be any change operations affecting \( q_{mg} \). Moreover, since in one iteration one triple pattern \( t' \in q_1 \) is replaced with its parent to produce \( q_2 \) the answers to \( q_1 \) would be contained in the answers to \( q_2 \), thus \( q_1 \subseteq q_2 \). By repeating the same operation \( q_1 \subseteq q_2 \ldots, q_n \subseteq q_{mg} \) and thus \( q_1 \subseteq q_{mg} \) by transitivity.

(b) Now we have to show that the generalized query produced \( q_{mg} \) is minimal. Let’s suppose that it is not minimal. This would allow the existence of a minimal generalized \( q_{mg} \) such that \( q \subsetneq q_{mg} \) and \( q_{mg} \subseteq q_{mg} \). By \( q_{mg} \subseteq q_{mg} \) this would mean that \( \exists t' \in q_{mg} \) such that \( t' \) is a parent of \( t \in q_{mg} \). But in order to construct \( q_{mg} \) we only use a parent triple pattern if a change operation affects that triple. This means that \( t \) is affected by a triple pattern. Thus, \( q_{mg} \) is not a generalized query which is not true.

![An alternative Ontology Version 1](image1.png)

**Figure 10.** An alternative Ontology Version 1

Assume for example, an alternative ontology version \( O1 \), shown on the left of Figure 10, where the "personal_info" property is a superProperty of the "gender" property. Assume also the same sequence of changes from \( O1 \) to \( O2 \) (the list of inverted changes presented in Section 3.2). Then, if query \( q_5 \) previously described is issued, we are able to identify that the triple "Actor, gender, xsd:string" has been deleted and look for a minimally-generalized rewriting.

The query that our system produces, and that provides more general answer to user query is:

\[
q11: \exists NAME, \exists GENDER
( (?X, type, Actor) \land
 ( ?X, fullname, ?NAME) \land
 ( ?X, personal_info, ?GENDER))
\]

### 5.6 Real Example Queries

This section presents two real example queries from the used ontologies. Its purpose is only to show how our approach is applied on the evaluation scenarios.

#### 5.6.1 CIDOC-CRM example query

Now we will present a real example from the CIDOC-CRM ontology, used and further analyzed in Section 6, using a simple template query from [50]. Assume for example that the user would like to get all objects used to capture an image. The corresponding SPARQL query formulated using the CIDOC-CRM version 4.2 is:

```
SELECT ?x WHERE {
  ?a rdf:type "E83.Image";
```
The query is issued to the system and initially it is expanded using the QuOnto engine. The engine will identify the subclasses and the sub-properties of the used classes/properties and it will produce the following query:

\[
\pi_5((?a, \text{type, E38.Image}) \land
   (?a, \text{P108B.was_produced_by, ?y}) \land
   (?y, \text{type, E65.Creation}) \land
   (?y, \text{P8F.took_place_on_or_within, ?x}) \land
   (?x, \text{type, E84.Information_Carrier}))
UNION
\pi_5((?a, \text{type, E38.Image}) \land
   (?a, \text{P108B.was_produced_by, ?y}) \land
   (?y, \text{type, E65.Creation}) \land
   (?y, \text{P8F.took_place_on_or_within, ?x}) \land
   (?x, \text{type, E22.Man-Made_Object}))
\]

The expanded query is the union of 47 subqueries. Assuming now that we have a database mapped to the ontology version 26.05.2009 the “Valid Rewriter” will check the constructed evolution log and will identify the corresponding GAV mappings that will be used for query rewriting to the target ontology version. One of these is produced from the following change operation

\[
\text{Split Class}(\text{GO_0050661}, \{ \text{GO_0050661}, \text{GO_0032562} \})
\]

So the query that it is issued on the data integration system that uses the version 26.05.2009 is the following:

\[
\pi_5((?x, \text{type, GO_0000166}) \land
   (?x, \text{P108B.was_produced_by, ?y}) \land
   (?y, \text{type, GO_0032562}) \land
   (?y, \text{P8F.took_place_on_or_within, ?x}) \land
   (?x, \text{type, E22.Man-Made_Object}))
UNION
\pi_5((?x, \text{type, GO_0000166}) \land
   (?x, \text{P108B.was_produced_by, ?y}) \land
   (?y, \text{type, GO_0032562}) \land
   (?y, \text{P8F.took_place_on_or_within, ?x}) \land
   (?x, \text{type, E22.Man-Made_Object}))
\]

6. IMPLEMENTATION & EVALUATION

The approach described in this paper was implemented on our exelixis\(^6\) platform [15]. We developed the exelixis platform as a web page using PHP/JQuery/HTML for the presentation and Java/PHP for implementing the algorithms. The interface is shown on Figure 11. Using our platform the user is able to load and visualize one version of an RDF ontology. The visualization is provided either through the jOWL\(^5\) API or the OWLSight\(^5\) plugin, or the Starlion\(^6\) tool. Then, the user is able to search for a class or property, to visualize the corresponding description and to explore the hierarchy of the ontology. Moreover, the user can construct a SPARQL query which is issued to the system. The system expands the query using the QuOnto engine and then it

---

\(^5\) http://139.91.183.29:8080/exelixis/
\(^6\) http://jowl.ontologyonline.org/
\(^6\) http://pellet.owldl.com/ontology-browser/
\(^6\) http://www.ics.forth.gr/~tzitzik/starlion/
computes the valid rewriting over the expanded query. Then, the query is forwarded to the underlying data integration systems, where it is answered. The results are unioned and presented to the user.

**Figure 11. The exelixis platform**

### 6.1 Evaluation Setup

In order to evaluate our system we used a workstation running Windows 7 with an Intel Core 2 Duo processor at 3.0 GHz, and 4GB memory.

Moreover, to test our system we used two ontologies: One medium-sized ontology (CIDOC-CRM), from the cultural domain which is rarely changed and one large-size ontology (Gene Ontology) from the bioinformatics domain which is heavily updated daily. Each one of those ontologies was used as a global schema in order to query the data mapped to them (to one of their versions).

CIDOC-CRM[^7] is an ISO standard which consists of nearly 80 classes and 250 properties. For our experiments we used 4 versions dated from 02.2002 (v3.2.1) to 06.2005 (v4.2) encoded in RDF/S. The detected change log that was produced identified 711 total changes.

Gene Ontology[^8] (GO) on the other hand, is composed of about 28000 classes. We have to note that the file containing the Gene ontology is over 100MB and most of the ontology editors fail to load the entire file. Moreover, we used the materialized version of GO without blank nodes. This restriction is enforced by the change detection algorithm we used, which does not deal with blank nodes. GO is updated on a daily basis and for our experiments we used 4 versions dated from 16.12.2008 to 26.05.2009. The change log that was produced contained 3482 changes.

The target of our evaluation was to demonstrate the impact of our system and to show that query rewriting between ontology versions can be achieved timely manner. To do that, we evaluated initially query rewriting between ontology versions. However, after rewriting queries between ontology versions, queries are forwarded to the underlying data integration systems in order to be answered. Since query evaluation affects user experience as well, we measured the time of the underlying data integration system (MASTRO) to evaluate the forwarded query.

We have to note that our purpose here was to evaluate the feasibility of our solution and not to do a thorough evaluation[^9] of the MASTRO data integration system which is not our contribution.

The evaluation we performed was based on two scenarios. One scenario with synthetic queries automatically constructed and one scenario with real queries captured from related projects and publications. For measuring query answering time, we used 10 relations in the data sources, with 10 rows each and 10 mappings between each ontology version and the local schemata.

We have to note that a comparison with other systems was not possible since there is no known implementation that allows query answering over multiple data integration systems that use different ontology versions.

### 6.2 Synthetic Evaluation

In the synthetic scenario we automatically generated random queries using CIDOC-CRM v.4.2. We created 20 queries for each one of the following categories: queries with 1, 3, 7 and 20 triple patterns. The synthetic evaluation was performed only for queries formulated using CIDOC-CRM ontology since the queries using the GO ontology ask for instances of only one GO-term (GO ontology is mostly a taxonomy) and rich queries including several properties cannot be produced.

#### 6.2.1 Scalability

Query rewriting between ontology versions depends on the query size and the number of changes among those versions (assuming fixed number of ontological constraints which is usually the case in bibliography). So, initially, we measured the time required for query expansion and valid rewriting for all combinations of query sub-goals and changes detected among ontology versions. Then, we measured the time for query evaluation in those cases. The results, for synthetic queries, are shown on the left of Figure 12. The times for query expansion, valid rewriting and query evaluation are shown at the bottom, middle and top part of each bar respectively.

[^9]: The evaluation of query answering over the source databases depends both on the size of the rewritten query, and the number of source relations mapped to the query atoms.
We can observe that the queries with a small number of triple patterns (1 or 3) can be answered almost instantly (max 0.221 sec). However, query answering time increases significantly for queries containing more than 7 triple patterns. Moreover, as queries become larger, the dominant time becomes the time required for valid rewriting. This is due to the exponential number of queries produced in the expansion phase, shown on Figure 13. Finally, the total query answering time increases as the number of change operations and the query sub-goals increase as well, which is in line with our theoretical expectations.

6.2.2 Impact

In this subsection we present the results of rewriting input queries between ontology versions, as the number of triples in each query increases. For rewriting queries from the ontology version 4.2 to the version 3.4.9 the results are presented in Figure 14 whereas from the ontology version 4.2 to the ontology version 3.2.1 the results are shown in Figure 15. For each query size there are three bars. The first bar shows the percentage of equivalent and the minimally-containing queries that could be produced, and the second bar the percentage of equivalent and the minimally-generalized queries. The third bar presents the percentage of the queries that could be answered “as-is” without any changes when they were issued to the data integration system that used previous ontology versions.

Figure 12. Average execution time for different query size/ontology versions on a) synthetic queries (left) and b) real queries (right)

**Figure 13.** The average number of sub-queries to be evaluated for the different number of triple patterns.

### 6.2.2 Impact

In this subsection, we present the results of rewriting input queries between ontology versions, as the number of triples in each query increases. For rewriting queries from the ontology version 4.2 to the version 3.4.9, the results are presented in Figure 14, whereas from the ontology version 4.2 to the ontology version 3.2.1, the results are shown in Figure 15. For each query size, there are three bars. The first bar shows the percentage of equivalent and the minimally-containing queries that could be produced, and the second bar shows the percentage of equivalent and the minimally-generalized queries. The third bar presents the percentage of the queries that could be answered “as-is” without any changes when they were issued to the data integration system that used previous ontology versions.

**Figure 14.** Query rewriting with over 309 change operations

**Figure 15.** Query rewriting over 711 change operations
From the charts we can see that the number of queries that could be answered “as-is” from the previous ontology versions decreases as the number of change operations and the query size increase. This is reasonable as the more query triples and change operations the more likely it is to be affected a part of the query. For example, looking at Figure 14, only the 40% of queries with 20 triple patterns could be answered “as-is” after 309 change operations, whereas we could produce the equivalent rewritings to the past ontology version for all of them. Even after 711 change operations for queries with 20 triple patterns we could produce rewritings for the 40% of the input queries as shown in Figure 15. However, as more changes are present the number of equivalent rewritings that can be produced drops and over-approximations should be provided.

6.3 Pragmatic Evaluation

To check the effectiveness of our system on real cases we used two sets of queries: 21 template queries for CIDOC-CRM coming from hundreds of user queries (9 query templates from [50] and 12 query templates from project 3d-COFORM) and the 38 most popular GO queries as they have been identified and provided from the AmiGO search engine.

6.3.1 Scalability

Firstly, we tried to identify the average query answering time for the real CIDOC-CRM queries. The results are shown in Figure 12 on the left. We can see that the average time to produce end evaluate a rewriting even after 711 change operations is less than 5 sec which shows the scalability of our approach. Notice that, on average, the time required for query expansion is greater than the time to perform the valid rewriting and query evaluation in all cases. Moreover, as the number of change operations doubles the same happens to the time for valid rewriting which is in line with the complexity of our algorithm.

The results for queries formulated using GO are presented in Figure 16. The average execution time for GO is less than 17 sec and is justified from the large size of the ontology. Most of the time is spent calculating the expansion of the queries since our system has to consider the inclusion dependencies of 28000 classes. Moreover, only 2,16 sec is spent (at the worst case) for valid rewriting and 0,4 sec for query evaluation.

We have to note that the average number of triple patterns for CIDOC queries was 10 whereas the average number of queries forwarded for query evaluation were 630 for CIDOC and 1589 for GO.

6.3.2 Impact

To illustrate the impact of our approach we present the percentage of equivalent, minimally-containing and minimally-generalized queries that our system could produce for the two set of queries. The results are shown in Figure 16 and Figure 17.

For CIDOC-CRM we observe that as the number of changes increases, the percentage of the queries that can be answered “as-is” drops to 33% whereas in GO as the number of changes increases the percentage of queries that can be answered as is remains the same 94%. This may seem peculiar because of the higher number of change operations that we have in the case of the Gene Ontology. However, if we carefully examine the corresponding change operations in each case we can easily identify that they change only a small percentage of the GO ontology (10% of the entire ontology was changed by the 3482 changes), whereas for the CIDOC-CRM the 711 change operations changed 54% of the entire ontology.

Moreover, we can identify that the number of equivalent rewritings we can produce drops as the number of changes increases in both cases. However, in GO we can produce a smaller percentage of equivalent rewritings compared to the CIDOC-CRM ontology. This is due to the fact that the GO ontology usually evolves by adding GO terms (which are translated in delete change operations when trying to produce rewritings to the previous ontology versions). And since the queries using GO ontology involve only one GO term, when this term is deleted we cannot produce minimally-containing rewritings. This is because the query produced is not safe any more. However, in most of the cases the deleted term had a superclass that could be queried instead. That’s why we could get minimally-generalized queries instead. These two test cases show the flexibility of our solution in different kind of ontologies and queries, and the great practical value of our approach.

An interesting observation made from our experiments was that the higher the level in the hierarchy of the queried classes and properties, the more probable was not to be able to produce an equivalent rewriting, since the expansion of the query used more terms of the ontology. Moreover, we noticed that we could find equivalent rewritings for all “star” queries whereas this was not true for all “path” queries.

---

10 http://www.3d-coform.eu/
11 http://amigo.geneontology.org/cgi-bin/amigo/go.cgi
7. DISCUSSION & CONCLUSION

In this paper, we argue that ontology evolution is a reality and data integration systems should be aware and ready to deal with that. To that direction, we presented a novel approach that allows query answering under evolving ontologies without mapping redefinition between each ontology version and the corresponding data sources.

Our architecture is based on a module that can be placed on top of any traditional ontology-based data integration system, enabling ontology evolution. It does so by using a high-level language of changes to model ontology evolution and uses those changes in order to rewrite not the mappings but the query itself among ontology versions. We have to note that the existence of other languages satisfying the properties of uniqueness, completeness, non-ambiguity is not ruled out. In fact we could use any other high-level language of changes satisfying those properties and our results would be exactly the same since the effectiveness is affected by non-information preserving changes which are defined independently of the high-level language of changes used.

Moreover, our approach is more general than trying to use schema composition and inversion to answer queries over multiple ontology versions. That is, because in the latter case complex mechanisms should be employed to produce the composition of the mappings between the ontology versions and the underlying data sources, depending also on the type of the mappings used on the underlying data integration systems. In our approach however, the underlying data integration systems are seen as “black-boxes” and our algorithms are independent of the type of mappings used between sources and ontology versions. Finally, in our case inversion is always possible to be produced whereas this is not guaranteed in mapping inversion. This is due to the fact that we consider inversion (composition) on a layer on top where always can find the inverse (composition) of any sequence of changes efficiently.

The potential impact of our approach is witnessed by being able to successfully provide rewritings on the worst case for the 88% of the CIDOC-CRM queries (after 711 change operations) and for the 97% of the GO queries (after 3482 change operations) among ontology versions. On the other hand if our system was not used, only a small percentage of the initial queries would be successful. For most of the queries, query answering is achieved within 5sec using a simple workstation, which also shows the usability and the scalability of our approach. We have to note that in the case of GO we materialized the ontology in order to minimize further the query execution time. The great benefit of our approach is the simplicity, modularity and the short deployment time it requires. It is only a matter of providing a new ontology version to our system to be able to use it to formulate queries that will be answered by data integration systems independent of the ontology version used.

As future work, several challenging issues need to be further investigated. For example, local schemata may evolve as well, and the ontologies used as global schema may contain inconsistencies. An interesting topic would be to extend our approach for OWL ontologies or to handle the full expressiveness of the SPARQL language. The latter would be a difficult task, since if we allowed OPT and FILTER operations, we would no longer have union of conjunctive queries and the problem in some cases might be undecidable. Moreover, if we leave the semantics from [38] we would have to deal also with duplicate treatment which would make the problem even more difficult. It becomes obvious that ontology evolution in data integration is an important topic and several challenging issues remain to be investigated in near future.

Acknowledgements

We would like to thank the reviewers for their valuable comments. This work has been supported by the eHealthMonitor and EURECA projects and has been partly funded by the European Commission under contracts FP7-287509 and FP7-288048.

8. REFERENCES


[46] A. Cali, D. Lembo, R. Rosati, On the decidability and complexity of query answering over inconsistent and


