Supporting Complex Changes in RDF(S) Knowledge Bases

Theodora Galani[1+], Yannis Stavrakas[1*], George Papastefanatos[1+], Giorgos Flouris[2*]

[1]Institute for the Management of Information Systems, RC ATHENA, GREECE
[2]Institute of Computer Science, FORTH, GREECE
{theodora, yannis, gpapas}@imis.athena-innovation.gr,
fgeo@ics.forth.gr

Abstract. The dynamic nature of web data brings forward the need for maintaining data versions as well as identifying semantically rich changes between them. In this paper, we advocate the need for supporting complex changes in evolving RDF(S) knowledge bases. We outline the basic challenges and provide solution insights through a real-world example from the field of biology.

Keywords: change management, data evolution, rdf(s)

1 Introduction

The increasing amount of information published on the web poses new challenges for data management. A central issue concerns evolution management, as the dynamic nature of data brings forward the need for maintaining data versions as well as identifying changes between them. For example, biologists often use ontologies in order to curate their data from multiple domains of interest like anatomy, diseases, biomedical investigations, etc. These ontologies are frequently updated as errors may need to be fixed or new knowledge about the state of the art may need to be incorporated. As a result, curators of depending ontologies are interested in understanding the evolution history in order to learn more about the changes that have taken place on the respective domain of interest.

In this paper, we argue that understanding data evolution should involve high-level, semantically rich, user-defined changes that we call *complex changes*. Formalizing complex changes involves facing the challenges of modeling, defining, detecting and querying changes. Although the concept of complex changes is not bound to any specific data model, in this paper we focus on RDF(S) knowledge bases, as RDF is a de-facto standard for representing data on the web. The goal of this paper is to highlight the main challenges as well as possible solution insights towards a framework that makes changes first class citizens.

* Supported by the EU-funded ICT project "DIACHRON" (agreement no 601043).
+ Supported by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

The paper outline is as follows. In section 2 we discuss in detail the challenges for supporting complex changes. In section 3 we provide an end-to-end real world example that demonstrates important aspects of our approach to the aforementioned problems. Finally, in section 4 we conclude the paper.

2 Challenges and Roadmap

Modeling changes. An approach for modeling changes in RDF(S) knowledge bases would be determining the added and deleted triples between versions. However, this is not sufficient for understanding data evolution. Human-readable, high-level changes should be employed. In this case, two basic issues must be taken into consideration.

- Granularity of changes. Fine-grained or coarse-grained changes? Fine-grained changes have the advantage of describing primitive changes, while coarse-grained changes provide more semantics and conciseness by grouping primitive changes in logical units.
- Semantics of changes. Model-specific or data- and application- specific changes? Model-specific changes describe modifications that may appear in a specific representation model. They constitute a fixed set of generic changes. On the other hand, data- or application- specific changes represent user-defined changes that suit on specific use-case scenarios. Supporting user-defined changes has the advantage of allowing different interpretations of evolution.

In order to tackle the above issues, we distinguish between *simple* and *complex* changes. Simple changes constitute a fixed set of fine-grained, model-specific changes. Complex changes are coarse-grained, user-defined, application-specific changes.

In previous works [2, 4, 6, 10], various lists of predefined changes have been proposed, usually distinguished into fine-grained and coarse-grained changes. In [6] formal semantics are defined guaranteeing useful properties. In [1] an approach for modeling changes as sequences of triples is proposed.

Defining changes. A declarative language for defining changes is needed for supporting user-defined complex changes. The language expressiveness should be investigated. A complex change definition should consist of a finite, non-empty list of simple or (already defined) complex changes, and a set of constraints over these changes. The supported constraints may filter parameter values, express pre- or post- conditions, relate change parameters, pose cardinality constraints (e.g. there must be at least one change of a specific type) and allow or not overlaps among changes.

In [8] a language for defining high-level changes, called Change Definition Language, has been proposed. Defined changes are detected over a version log [7] using temporal queries, assuming that the version log is populated as modifications apply. In [9] a framework for defining changes using SPARQL query features is presented as an extension of [6].

Detecting changes. As new dataset versions are periodically released, simple and complex changes can be detected among versions. In [4] a fixed-point algorithm for comparing ontology versions has been proposed. The algorithm is based on heuristic-

based matchers, introducing uncertainty to the results. On the other hand, in [6] the detection process does not introduce any uncertainty to the results. In our approach, we need to identify the rules for mapping complex change definitions into processes that return instances of the respective change patterns. The performance of the detection process has to be investigated with respect to the number of changes and the type of constraints in complex change definitions, as well as the dataset versions' size and the number of changes performed between them.

Querying changes. In our view, querying data evolution should be based on data as much as on changes. Changes, like data, can appear in the query body to express complex conditions, like the fact that an entity has been modified in a specific manner, or can be returned by the query in order to retrieve explicit change instances. Some interesting query types that should be supported are the following:

- Retrieve changes among versions, or restrict selected changes by the type of change or the elements that they have affected or the versions between which they are detected.
- Retrieve elements, given that changes of specific type have affected them at specific versions.

3 An End-to-End Example

The Experimental Factor Ontology (EFO) [3] provides a systematic description of many data elements available in EBI[1] databases, and for external projects. It combines parts of several biological ontologies regarding anatomy, disease and chemical compounds in order to support data annotation, analysis and visualization. EFO is frequently updated as new classes are added, while others are changed or made obsolete. Classes in EFO are described by metadata like class label, definition, synonyms, etc.

Consider that a new class is added into the ontology. This class is also assigned with a class label, a textual definition and synonyms of the class label. The class label corresponds to rdfs:label annotation property, the textual definition corresponds to the efo:definition property and the synonym to the efo:alternative-term property. Note that for simplicity and space limitations we consider only these operations.

Modeling changes. These changes are fine-grained and can be described by model-specific operations. The addition of a new class can be modeled as *Add_Type_Class(c)*, where *c* is the new class. The addition of a new label can be modeled as *Add_Label(c, l)*, where *c* is the respective class holding the new label *l*. The addition of a new definition or synonym corresponds to an addition of a new property and can be modeled as *Add_Property_Instance(s, p, o)*, where *p* is the new property which is assigned to class *s* with value *o*. In our approach, these are simple changes. We can rely on [6] for defining simple changes by selecting a minimal set of primitive changes on RDF(S) having the properties of completeness and unambiguity.

Notice that *Add_Property_Instance* suits all possible properties, while in this scenario the assigned properties are of two specific types: efo:definition and

[1] http://www.ebi.ac.uk/

efo:synonym. It is more suitable to have intuitive changes regarding the specific properties involved, like *Add_Definition* and *Add_Synonym*. Also, the discussed modifications are likely to appear jointly. As a result, it may be useful to demonstrate these changes as a unit. Therefore, they can be grouped into one change named *Add_Annotated_Class*. In our approach, these are examples of complex changes.

Defining changes. The complex changes *Add_Definition*, *Add_Synonym* and *Add_Annotated_Class* can be defined as follows:

```
CREATE COMPLEX CHANGE Add_Definition(class, definition) {
   CHANGE LIST Add_Property_Instance(class, prop, definition);
   SELECTION FILTER prop='efo:definition';   };
CREATE COMPLEX CHANGE Add_Synonym(class, synonym) {
   CHANGE LIST Add_Property_Instance(class, prop, synonym);
   SELECTION FILTER prop='efo:alternative_term';   };
CREATE COMPLEX CHANGE Add_Annotated_Class(class, label, defini-
tion, synonym) {
   CHANGE LIST Add_Type_Class(class), Add_Label(class, label),
Add_Definition(class, definition), Add_Synonym(class, synonym)
*;   };
```

The name and parameters of each defined complex change are declared right after the *CREATE COMPLEX CHANGE* clause. In the *CHANGE LIST* clause the contained simple or complex changes are declared. Note that the asterisk (*) beside *Add_Synonym* in *Add_Annotated_Class* definition indicates that there might be zero, one, or more such changes, one for each added synonym, posing a cardinality constraint. Defining *Add_Definition* and *Add_Synonym* includes a constraint, declared in the *SELECTION FILTER* clause, filtering the property type. In *Add_Annotated_Term*, the parameter name *class* is used among the contained changes, indicating that they refer to the same actual class.

Detecting changes. As ontology versions are periodically released, we can identify the changes that have occurred among versions. Simple changes have to be detected first. Notice that *Add_Definition* and *Add_Synonym* are defined in terms of simple changes, while *Add_Annotated_Class* includes complex changes too. Therefore, *Add_Definition* and *Add_Synonym* should be detected first by evaluating their definitions over the detected simple change instances, while *Add_Annotated_Class* next as it depends on complex change instances too. Alternatively, *Add_Annotated_Class* can be expressed in terms of simple changes, by substituting *Add_Definition* and *Add_Synonym* changes with their definitions. In this way, all complex change definitions can be evaluated over the detected simple change instances. The detected simple and complex change instances constitute a hierarchy of changes, where the user can see the changes themselves as well as how they are interconnected.

Querying changes. For querying changes, SPARQL can be extended with suitable keywords. The following query gives an example of querying changes. It returns all classes that have been added and annotated between versions 2.45 and 2.46. For this example, we assume that defined changes and detected instances are represented in an ontology of changes as in [9]. Notice that the requested classes are the value of

co:aac_p1 parameter of *Add_Annotated_Class*. Also, *change_span* is a function that verifies whether the complex change instance (*?c*) is detected between the requested versions. Finally, the *FROM CHANGES ON DATASET* clause declares that the triples pattern concerns changes regarding a specific dataset <*D*>.

```
SELECT ?class
FROM CHANGES ON DATASET <D>
WHERE {
  ?c rdf:type co:Add_Annotated_Class; co:aac_p1 ?class.
  FILTER change_span(?c BETWEEN VERSION 2.45 AND 2.46). }
```

4 Conclusions

In this paper we advocated the need for formalizing complex changes over RDF(S) knowledge bases and outlined the basic challenges that have to be faced to realize our vision. An example inspired from the biological domain is used to motivate the need for complex changes and present the basic concepts of a possible solution. Nevertheless, supporting complex changes may be useful in any evolving domain.

5 References

1. S. Auer and H. Herre. A versioning and evolution framework for RDF knowledge bases. In Perspectives of Systems Informatics, 6th International Andrei Ershov Memorial Conference on, 2007.
2. M. Klein. Change management for distributed ontologies. Ph.D. thesis, Vrije University, 2004.
3. J. Malone, E. Holloway, T. Adamusiak, M. Kapushesky, J. Zheng, N. Kolesnikov, A. Zhukova, A. Brazma, H. Parkinson. Modeling Sample Variables with an Experimental Factor Ontology. Bioinformatics 26(8):1112-1118, 2010.
4. N. F. Noy, and M. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. In Proceedings of the 18th National Conference on Artificial Intelligence, 2002.
5. G. Papastefanatos, Y. Stavrakas, and T. Galani. Capturing the history and change structure of evolving data. 7th International Conference on Advances in Databases, Knowledge, and Data Applications, 2013.
6. V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in RDF(S) KBs. ACM Trans. Database Syst., 38(1), 2013.
7. P. Plessers and O. De Troyer. Ontology change detection using a version log. In Proceedings of the 4th International Semantic Web Conference, 2005.
8. P. Plessers, O. De Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. J. Web Sem. 5(1): 39-49, 2007.
9. Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas. A flexible framework for defining, representing and detecting changes on the data web. CoRR abs/1501.02652, 2015.
10. L. Stojanovic. Methods and tools for ontology evolution. Ph.D. thesis, University of Karlsruhe, 2004.

Capturing the History and Change Structure of Evolving Data

George Papastefanatos, Yannis Stavrakas, Theodora Galani
IMIS, RC ATHENA
Athens, Greece
{gpapas,yannis,theodora}@imis.athena-innovation.gr

***Abstract*—Evo-graph is a model for data evolution that encompasses multiple versions of data and treats changes as first-class citizens. A change in evo-graph can be compound, comprising disparate changes, and is associated with the data items it affects. In previous papers, we have shown that recording data evolution with evo-graph is very useful in cases where the provenance of the data needs to be traced, and past states of data need to be re-assessed. We have specified how an evo-graph can be reduced to the snapshot holding under a specified time instance, we have given an XML representation of evo-graph called evoXML, and we have presented how interesting queries can be answered. In this paper, we explain how evo-graph is used to record the history of data and the structure of changes step by step, as the current snapshot evolves. We present C2D, a novel framework that implements the concepts in the paper using XML technologies. Finally, we experimentally evaluate C2D for space and time efficiency and discuss the results.**

Keywords-data evolution; change modeling

I. INTRODUCTION AND PRELIMINARIES

Data published on the Web undergo frequent changes due to advancements in knowledge and due to the cooperative manner of their curation. Users of scientific data, in particular, would like to go beyond revisiting past data snapshots, and review how and why the recorded data have evolved, in order to re-evaluate and compare previous and current conclusions. Such an activity may require a search that moves backwards and forwards in time, spread across disparate parts of a database, and perform complex queries on the semantics of the changes that modified the data. The need for accounting for past changes and tracing data lineage is evident not only in scientific data, but also in a wide range of web information management domains.

Motivating Example. We will use an example taken from Biology: the revision in the classification of diabetes, which was caused by a better understanding of insulin [12]. Initially, diabetes was classified according to the age of the patient, as *juvenile* or *adult onset*. As the role of insulin became clearer two more subcategories were added: *insulin dependent* and *non-insulin dependent*. All *juvenile* cases of diabetes are *insulin dependent*, while *adult onset* may be either *insulin dependent* or *non-insulin dependent*. In Fig. 1, the leftmost image depicts a tree representation of the initial diabetes classification, while the rightmost the revised classification. These two representations, however, do not provide any information about which parts of the data evolved and how, which changes led from one version to another, or what changes were applied on which parts of the data. Recording change operations in a log or discovering deltas out of successive versions, like many systems do, do not solve the problem; in most cases isolated operations are impossible to interpret a posteriori. This is because they usually form more complex, semantically coherent changes, each comprising many small changes on disparate parts of the data.

We argue that in systems where evolution issues are paramount, changes should not be treated solely as transformation operations on the data, but rather as *first class citizens* retaining structural, semantic, and temporal characteristics. In previous work, we proposed a graph model, *evo-graph* [16], and its XML representation, *evoXML* [17], capturing the relationship between evolving data and changes applied on them. A key characteristic is that it explicitly models changes as first class citizens and thus, enables querying data and changes in a uniform way. In what follows, we discuss some preliminary concepts on evo-graph and then present the contribution and structure of this paper.

Snap-graph. We assume that data is represented by a rooted, node-labeled, leaf-valued graph called *snap-graph*. A snap-graph S (V, E) consists of a set of nodes V, divided into complex and atomic, with atomic nodes being the leaves of the graph, and a set of directed edges E. At any time instance, the snap-graph undergoes arbitrary changes.

Evo-graph. An *evo-graph* G is a graph-based model that captures all the instances of an evolving snap-graph across time, together with the actual change operations responsible for the transitions. It consists of the following components:

- *Data nodes*, divided into *complex* and *atomic*: $V_D = V_D^c \cup V_D^a$.
- *Data edges* depart from every complex data node, $E_D \subseteq (V_D^c \times V_D)$.
- *Change nodes* are nodes that represent change events. Change nodes are depicted as triangles, to distinguish from circular data nodes. They are divided into *complex* and *atomic* (denoting basic change operations): $V_C = V_C^c \cup V_C^a$.
- *Change edges* connect every complex change node to the (complex or atomic) change nodes it encompasses: $E_C \subseteq (V_C^c \times V_C)$.
- *Evolution edges* are edges that connect each change node with two data nodes, specifically the version before and after the change: $E_E \subseteq (V_D \times V_C \times V_D)$.

Intuitively, the evo-graph consists of two interconnected graphs: a data graph comprising the different versions of

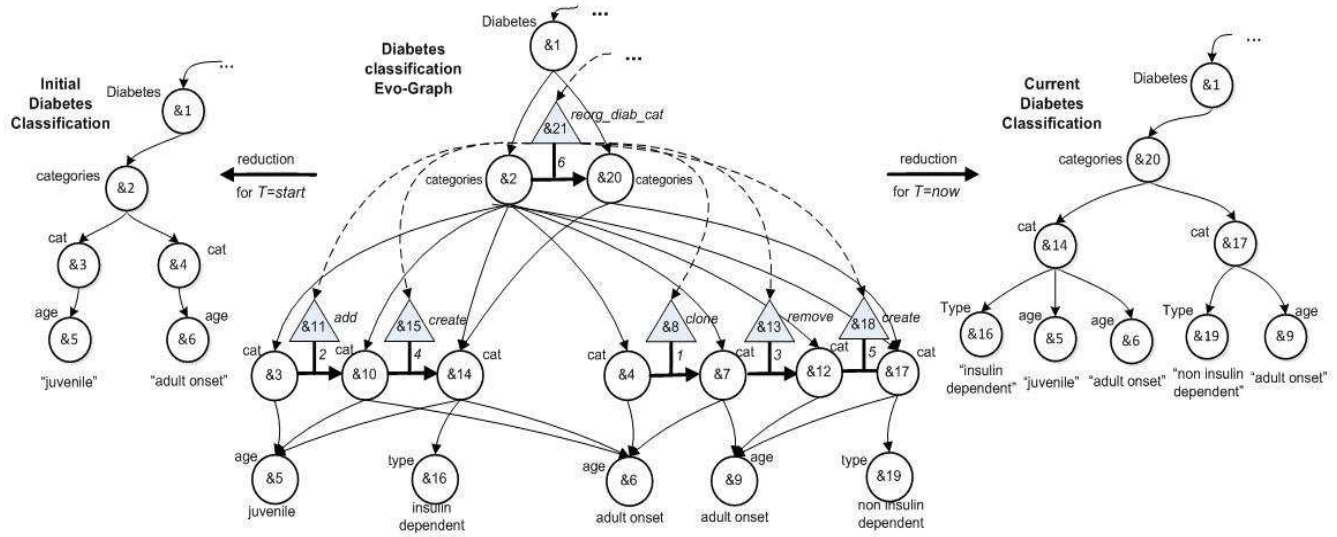

Figure 1. Snap Graphs of diabetes classification before (left) and after (right) revision and the corresponding evo-graph (middle).

data, and a tree of changes. The data graph defines the structure of data, while the change graph defines the structure of changes. These two graphs interconnect via evolution edges. Consequently, there are two roots: the data root, r_D, and the change root, r_C. Moreover, we annotate change nodes with a timestamp denoting the time instance that the specific change occurred. These timestamps are used for determining the validity timespan of all data nodes and data edges in the evo-graph. Evo-graph can be reduced to a snap-graph holding under a specified time instance through the *reduction* process [16]. A snap-graph is actually a trivial case of an evo-graph, consisting of a set of data nodes $V \subseteq V_D$ and a set of data edges $E \subseteq E_D$.

As an evo-graph example consider the middle image in Fig. 1, which represents the revision in the diabetes classification from the graph of Fig. 1 left to the graph of Fig. 1 right. The revision process is denoted by the complex change *reorg_diab_cat*, (node &21) composed by 5 basic snap changes (in the order they occurred): *clone* (node &8), *add* (node &11), *remove* (node &13), *create* (node &15), and *create* (node &18). Note the use of evolution edges; in the case of *add* the evolution edge is annotated with the timestamp 2 and connects node &3 (initial version) with node &10 (version after adding the child node &6). Node &10 is still a child of node &2, but for simplicity the relevant edge is omitted. The reduction of the evo-graph for T=*start* results in the snap-graph of the leftmost image of Fig. 1, while for T=*now* in the snap-graph of the rightmost image of Fig. 1. For the complete definitions of basic snap changes see section 2.1.

EvoXML. In [17] we have shown how evo-graph can be represented in an XML format, called *evoXML*. TABLE I. presents an evoXML example. Due to space limitations, the evoXML example covers up to time instance 1 of the evo-graph in Fig. 1; specifically it includes only the *clone* operation (node &8) in lines 12-15, 20. Notice that the edge from node &7 to node &6 (which actually denotes that &6 remains a child of the next version of node &4) is represented through the evoXML reference *evo:ref* in line 13, which points to the element in line 10. Also notice how the change node &8 is represented in line 20.

Querying Evolution. Finally, in [16],[17] we have outlined *evo-path*, an XPath extension that help us posing regular queries over data snapshots as well as time- and change-aware queries on evo-graph. We have also shown how evo-path expressions can be evaluated on evoXML via equivalent XQuery expressions. Evo-path takes advantage of the complex change information in order to retrieve and relate data that are otherwise distant and irrelevant to each other. Queries expressed on evo-graph include:

- Temporal queries on the history of data nodes, like "which is the structure of categories before the time instance 6"?
  `Evo-path: //Diabetes/categories [ts() not covers {now}]`
- Evolution queries on changes applied to data nodes, like "which changes are associated with the change responsible for the reorganization of diabetes categories" (node &21)?
  `Evo-path: <//reorg_diab_cat/*>`
- Causality queries on relationships between change nodes and data nodes, like "what are the previous versions of all data nodes that changed due to the reorganization of diabetes categories"?
  `Evo-path: //* [evo-before() <//reorg_diab_cat>]`

Contribution and Structure. In this paper, we first define a set of *basic changes* on the snap-graph, and how these can be combined to construct *complex changes* (section 2). We then define a *set of basic operations on the evo-graph, and a translation from snap-graph changes to evo-graph operations*, such that as changes occur on the snap-graph, the evo-graph grows to represent those changes together with all the successive snap-graph versions (section 2). Furthermore, we introduce the *C2D framework* (section 3), a prototype system that implements the concepts introduced in this paper, and progressively builds the evo-graph as changes take place on the current snap-graph. We present

TABLE I. EVOXML FOR TIME INSTANCE 1.

```
1  <evo:evoXML xmlns=""
2    xmlns:evo="http://web.imis.athena-innovation.gr/projects/c2d">
3     <evo:DataRoot evo:id="dataroot">
4       <Diabetes evo:id="1">
5         <categories evo:id="2">
6           <cat evo:id="3">
7             <age evo:id="5">juvenile</age>
8           </cat>
9           <cat evo:id="4">
10            <age evo:id="6">adult onset</age>
11          </cat>
12          <cat evo:id="7" evo:ts="1" evo:previous="4">
13            <age evo:ref="6"/>
14            <age evo:id="9">adult onset</age>
15          </cat>
16        </categories>
17      </Diabetes>
18    </evo:DataRoot>
19    <evo:ChangeRoot evo:id="changeroot">
20      <clone evo:id="8" evo:tt="1" evo:before="4" evo:after="7"/>
21    </evo:ChangeRoot>
22  </evo:evoXML >
```

and discuss a detailed *experimental evaluation* of C2D (section 3). Finally, we review the related work (section 4) and we conclude the paper (section 5).

II. ACCOMMODATING BASIC AND COMPLEX CHANGES IN EVO-GRAPH

A. Snap Basic and Complex Change Operations

In this section, we define the basic change operations applied on a snap-graph S(V,E) (*snap changes* for short) and present how they can be employed to define complex changes. We consider the following snap changes:

- *create(v^P, v, label, value)*. Creates a new atomic node *v* with a given *label* and *value* and connects it with its parent node v^P. If v^P is an atomic node, it becomes complex.
- *add(v^P, v)*. Adds the edge *(v^P, v)* to E, effectively adding *v* as a child node of v^P. The nodes v^P, *v* must already exist in V. If v^P is an atomic node, it becomes complex.
- *remove(v^P, v)*. Removes the edge *(v^P, v)* from E. If *v* has no other incoming edges, it is removed from V. If v^P has no other children, it becomes an atomic node with the default value (empty string).
- *update(v, newValue)*. Updates the value of an atomic node *v* to *newValue*.
- *clone(v^P, v^{source}, v^{clone})*. Creates a new data node v^{clone} with the same label/value as v^{source}, and a deep copy of the subtree under v^{source} as a subtree under the node v^{clone}. The node v^P must be a parent of v^{source}. The edge (v^P, v^{clone}) is added to E, making v^{clone} a sibling of v^{source}.

The above definitions describe the effect of each snap change to the current snap-graph. These changes leave the snap-graph in any possible consistent state. Note that the effect of the *clone* snap-change is to create a deep copy of a subtree under the same parent node. Although *clone* can be expressed as a sequence of other snap changes, we chose to consider it as a basic operation. The reason is that deep copy is difficult to express using successive *create* operations, while at the same time it is an essential operation for expressing complex changes like *move-to*, and *copy-to*.

A *complex change* applied on a node of the snap-graph is a sequence of basic and other complex change operations that are applied on the node itself or/and the node's descendants, and allows us to group operations in semantically coherent sequences. Applying a complex change on a snap-graph involves the application of each constituent change in the order they appear. Consider the complex change *reorg_diab_cat* applied on *categories* node of the leftmost image of Fig. 1. This change is expressed as a sequence of five basic snap changes, as follows:

```
reorg-diab-cat (&2) {
   clone (&4, &6, &9)
   add (&3, &6)
   remove (&4, &6)
   create (&3, &16, "type", "insulin dependent")
   create (&4, &19, "type", "non insulin dependent") }
```

B. Capturing Versions and Changes with Evo-graph

In our approach, snap changes are not actually applied on the snap-graph, but on the evo-graph. This is shown in Fig. 2, which illustrates the effects of snap changes to the evo-graph. Fig. 2 depicts three images for each snap change; the leftmost image shows the initial snap-graph before the change, the rightmost image shows the current snap-graph after the snap change, and the middle image shows the evo-graph fragment encompassing both snapshots, together with the change. Notice that these snap-graph fragments are actually *reductions* [16] of the respective evo-graph under different time instances. Thus, the *create* operation in Fig. 2 actually causes node &4 to be added under the parent node &5, and not under &2, as would be the case if *create* was applied directly on the snap graph. This is a technical issue tackled with at the implementation level, and does not introduce any ambiguities.

In order to implement snap changes on an evo-graph G we introduce the following evo-graph operations:

- *addDataNode (v_D^P, v_D, label, value)*. Creates a new atomic data node v_D as a child of v_D^P with a given *label* and a *value*. If v_D^P is an atomic node, it turns into complex.
- *addDataEdge (v_D^P, v_D)*. Creates a new data edge from node v_D^P (parent) towards node v_D (child). The two nodes must already exist in V_D. If v_D^P is an atomic node, it turns into complex.
- *applyAtomicChange(v_D^1, v_D^2, value, v_C, v_C^P, label, timestamp)*. This operation "evolves" node v_D^1 to node v_D^2, as the result of applying a snap change. First, a new atomic data node v_D^2 with the same label as v_D^1 and a given value is created, and is connected as a child of all the current parents of v_D^1. Then, a new atomic change node v_C with the *label* and *timestamp* is created, and is connected as a child of node $v_C^P \epsilon V_C^c$. The *label* denotes one of the snap changes defined previously. Finally, a new evolution edge $e = (v_D^1, v_C, v_D^2)$ is created between the data nodes v_D^1, v_D^2 and the change node v_C.

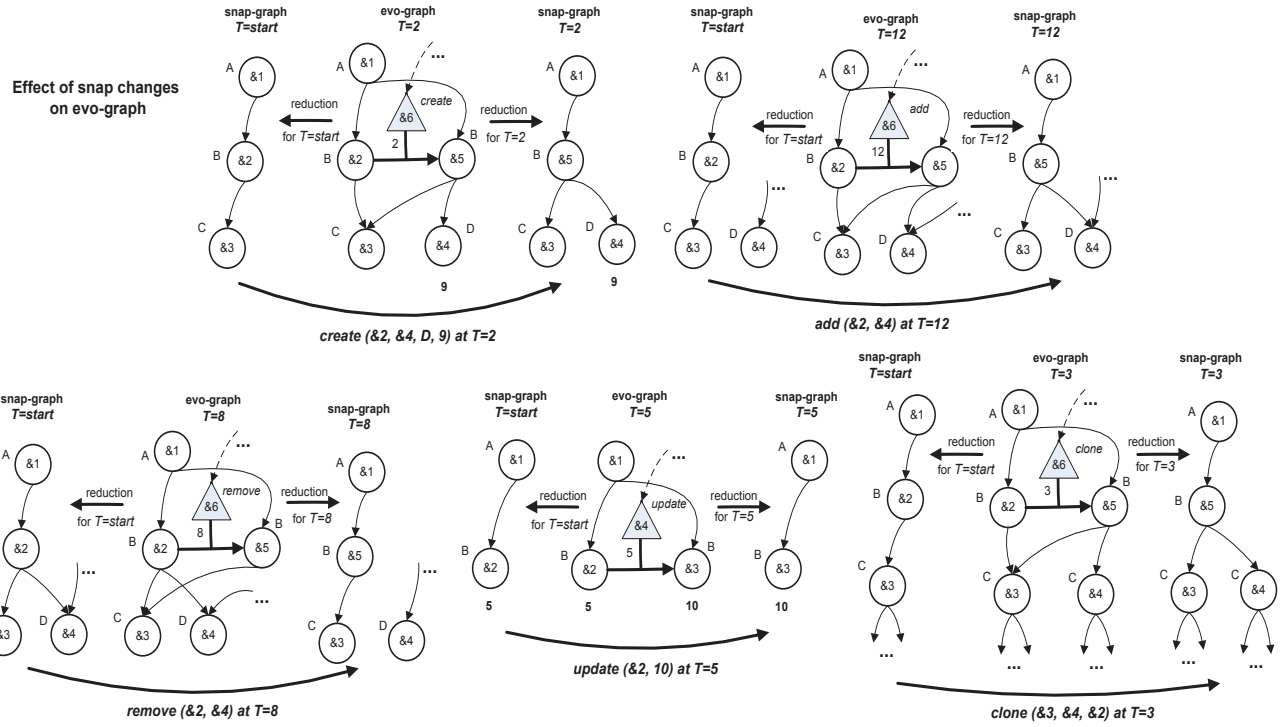

Figure 2. Effect of snap change operations on the evo-graph.

- *applyComplexChange*(v_D^1, v_D^2, v_C, v_C^p, *label, timestamp,* {v_C^1, v_C^2, ..., v_C^n}). This operation "evolves" node v_D^1 to node v_D^2, as the result of applying a complex change operation on the snap-graph. First, a new atomic data node v_D^2 with the same label as $v_D^1$and the default value (empty string) is created, and is connected as a child of all the current parents of v_D^1. A new *complex* change node v_C with the *label* and *timestamp* is created, and is connected as a child of the complex change node $v_C^p \in V_C^c$. The *label* is the name of the complex change and can be any string. After that, v_C is connected as a parent of the change nodes {v_C^1, v_C^2, ..., v_C^n}. Finally, a new evolution edge e=(v_D^1, v_C, v_D^2) is created between the data nodes v_D^1, v_D^2 and the change node v_C.

Note that we employ two separate evo-graph operations for applying snap-graph basic and complex changes. For complex changes, the *applyComplexChange* is used, which creates a new *complex* change node, a new version for the affected data node, a new evolution edge connecting the change node and the two data node versions and finally connects the complex change node as the parent of its constituent change nodes. For basic changes, the *applyAtomicChange* is used, which creates a new *atomic* change node, a new version of the data node that is affected by the change, and a new evolution edge. The exact implementation of each snap change in terms of evo-graph operations is given in TABLE II. .

For each snap change in TABLE II. , a *timestamp* is given (appears as *t*) and, if this change is part of a complex change, the parent complex change (v_C^P) is also specified. If no parent complex change is specified, we assume the parent is the change root r_C. Note, that all snap change implementations in TABLE II. start with *applyAtomicChange*, which creates the corresponding change node and the associated data node in evo-graph.

TABLE II. ACCOMMODATING SNAP CHANGES IN EVO-GRAPH.

	create (v_D^P, v_D, ***label, value***), ***t***, v_C^P
1	{ applyAtomicChange(v_D^P, v'_D^P, '',v_C, v_C^P, 'create', t);
2	for $v_i \in$getCurrentChildren(v_D^P)
3	addDataEdge (v'_D^P,v_i);
4	*// create the new data node* and *connect it to the new parent node*
5	addDataNode *(v'_D^P, v_D, label, value);* }
	add (v_D^P, v_D), ***t***, v_C^P
1	{ applyAtomicChange(v_D^P, v'_D^P, '',v_C, v_C^P, 'add', t);
2	*//connect the new parent node to all current children plus* v_D
3	for $v_i \in$(getCurrentChildren(v_D^P)$\cup v_D$)
4	addDataEdge (v'_D^P,v_i) ; }
	remove (v_D^P, v_D), ***t***, v_C^P
1	{ applyAtomicChange(v_D^P, v'_D^P, '',v_C, v_C^P, 'remove', t);
2	*//connect the new parent node to all current children except for* v_D
3	for $v_i \in$(getCurrentChildren (v_D^P)-v_D)
4	addDataEdge (v'_D^P,v_i); }
	update (v_D, ***newValue***), ***t***, v_C^P
1	{ applyAtomicChange(v_D, v'_D, newValue,v_C, v_C^P, 'update', t) }
	clone (v_D^P, v_D^{source}, v_D^{clone}), ***t***, v_C^P
1	{ applyAtomicChange(v_D^P, v'_D^P, '',v_C, v_C^P, 'clone', t);
2	for $v_i \in$(getCurrentChildren *(v_D^P)*
3	addDataEdge (v'_D^P,v_i);
4	*//clone the source data node*
5	addDataNode *(v'_D^P, v_D^{clone}, v_D^{source}.label, v_D^{source}.value);*
6	*//create a deep copy of the cloned node*
7	for $v_i \in$getCurrentChildren (v_D^{source})
8	addDataNode(v_D^{clone}, v'_{i_s}, v_i.label, v_i.value);
9	*repeat step* 7 *for* $v_D^{source} = v_i$ *and* v_D^{clone}=v'_i }

III. IMPLEMENTATION AND EVALUATION

A. The C2D Framework

We have implemented all above concepts into the C2D (standing for Complex Changes in Data evolution) framework. C2D has been developed in Java, on top of Berkeley DB XML [3], an embedded XML database used to manage the evoXML representation of evo-graphs. In C2D, changes applied on the snap-graph are fed into a process that populates the evo-graph. A snap change is always applied on the current snap-graph (represented in XML in C2D), which is actually produced as a reduction [16] of the evo-graph for the time instance T=*now*. This flow is depicted in Fig. 3. The top layer in Fig. 3 is the *view layer*, where changes are launched. The purpose of the *logical model* layer is to guide the translation processes between the view layer and the *storage representation layer*, where changes actually take place.

Change operations on the evo-graph are implemented as XML update operations on the corresponding evoXML. Expressing evo-graph operations with the XQuery Update language is straightforward. For example the *addDataNode (&17, &19, "type", "non insulin dependent")* operation is expressed with the following XQuery Update *insert* expression on the evoXML.

```
insert node <type evo:id="19">non insulin  dependent </type>
into
/evo:evoXML/evo:DataRoot/Diabetes/categories/cat[evo:id="17"]
```

B. Experimental setting

Our goal was to examine how our approach depends on a number of factors that characterize the data. We first examined the space efficiency of evoXML for various configurations, regarding: the structure of the initial XML tree, the type of snap changes, and the selectivity of the elements. We also examined the performance of the reduction process with respect to the size of the evoXML file. Note that the comparison with other versioning approaches [4], [6], [7] was not pursued, as these works do not consider the role of changes as first class citizens in storing and querying evolving data.

Experiments were performed over synthetic XML data, on a PC with Intel Core 2 CPU 2.26 GHz, and 4.00 GB of RAM. The initial XML file was generated with [19] and contained about 10^5 elements, over which 10^4 snap changes were sequentially applied as XQuery Update statements. A new version was assumed after every 1000 changes; therefore 10 successive versions have been created for each setting. We recorded the size (in terms of the number of XML elements) of each "snap" version, and the size of the evoXML file at the same instance. Furthermore, we examined the performance of the reduction process for the current snapshot (T=*now*), and the initial snapshot (T=*start*).

Regarding the structure of the initial data, we used two XML files with the same number of elements: (a) one corresponding to a snap-graph with a "deep" tree structure (denoted s_1) with five levels and elements having a fan-out of 10, and (b) a file with a "broad" tree structure (denoted s_2) with only two levels and elements with a fan-out of

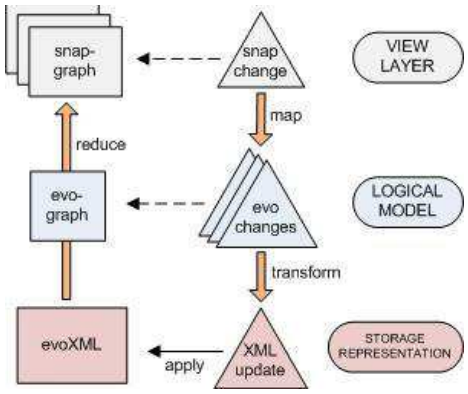

Figure 3. C2D framework overview.

about 330 elements. We have applied three sets of snap changes: (a) equal percentage for all changes except *clone* (denoted t_1), (b) 80% *update* and 20% *create* and *remove* (denoted t_2), and (c) equal percentage for all changes including *clone* (denoted t_3). Finally, concerning elements selectivity, changes have been applied either on all elements (denoted n_1) or on a fixed set of pre-selected elements so that each element is affected by 5 changes on average per version (denoted n_2).

We have examined the following combinations of the above parameters: (t_1n_1), (t_3n_1), (t_2n_1), and (t_2n_2) for each of s_1, s_2. t_1n_1 captures the typical case when random changes are uniformly applied on all elements. t_3n_1 is similar to t_1n_1, but it also includes *clone*. We have separately examined the *clone* operation, as it may arbitrarily result in the addition of a large amount of data. t_2n_1 captures the case where most (80%) change operations are *update* on random leaf elements, and only 20% are create or *remove*. Finally, t_2n_2 is like the previous case except that changes are concentrated on a pre-selected fixed set of elements.

Intuitively, we expect that the size of the evoXML depends on the number of snap changes performed. We also expect that it depends on the average fan-out of the snap-graph, while it remains insensitive to its average height. This is due to the way that each snap change operation is implemented on the evo-graph. Next, we present and discuss the results.

C. Results and Discussion

In Fig. 4 (a) and (b) we present the evoXML sizes per version. Subsequently, we discuss how this size is affected by the aforementioned configurations parameters.

File structure. For all configurations, better space efficiency is achieved for s_1. For smaller fan-outs (s_1), the evoXML has a smoother increase in size than for large fan-outs (s_2). A snap change occurring on an element adds *evo:ref* elements for all of its children (i.e. fan-out) that are still valid in the new version. Hence, the increase in the evoXML size is relative to the average fan-out.

Type of changes. t_2 outperforms t_1 and t_3. The majority of changes in t_2 are *update*, which have a smaller impact on the evoXML size. Again, the key point is the number of new elements that each change adds. Observe from TABLE II. that all changes add at least two new elements;

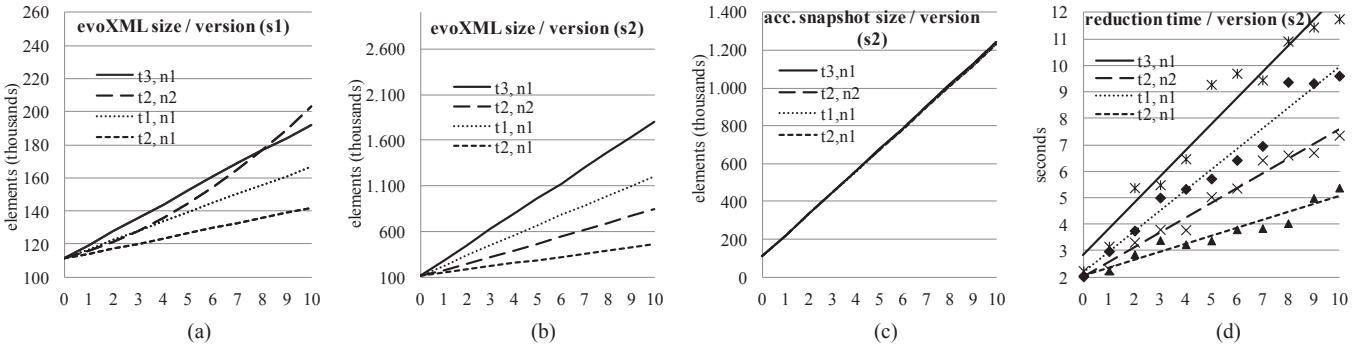

Figure 4. evoXML size (a), (b), accumulative snapshot size (c) and current snapshot reduction time (d) per version for various configurations.

one evolved data element and one change element. *update* adds only these two elements, whereas *create* and *add* insert one additional element for the new child, plus *evo:ref* elements for its siblings. *remove* results in inserting *evo:ref* elements in the evoXML for all the siblings of the removed element. Finally, *clone* adds a variable number of elements according to the height and average fan-out of the subtree that is cloned. On the other hand, the percentage of *create* and *remove* in t_1 is higher. In t_3, the use of *clone* further increases the file size by creating a deep copy of the subtree of the elements on which it is applied.

Selectivity of elements. Applying changes randomly on all elements (n_1) seems to have a smoother impact on the increase of the file size (e.g., compare t_2n_1 and t_2n_2 for each of s_1, s_2). This is due to the fact that changes are uniformly distributed over all the elements. On the other hand, the increase is higher when changes are targeting a fixed set of elements (n_2). Changes in t_2n_2 are sequentially applied on the same elements, i.e., *create* is applied on the same elements, increasing the number of their children and thus the number of *evo:ref* elements to be inserted when a subsequent *create* occurs on the same element.

Overall, the evoXML size depends almost linearly on the number of the snap changes applied, given that the average fan-out is constant. Moreover, the increase rate of the evoXML size is proportional to the average fan-out of its elements. This is more evident in t_2n_2 for s_1, where the average fan-out of the elements sustaining changes increases significantly per version, resulting in a boost in the evoXML size, whereas in s_2 the fan out increase rate is much smoother.

In Fig. 4 (c) we present the accumulative size of the snapshots produced per version. This approach can be considered as an alternative to evoXML. For space reasons, we only depict the series for s_2, as s_1 shows a similar trend. The accumulative size of all snapshots per version is significantly bigger than the evoXML size, for all runs over s_1. The same holds for all configurations of s_2, except for t_3n_1 where many *evo:ref* elements are added in the evoXML file. Note that the overlap of the series is due to the small variance in the accumulative snapshot size between configurations.

Regarding the performance of our reduction algorithm, we have measured the time the reduction process takes for producing the current and the initial snapshots. The results for the current snapshot for s_2 are shown in Fig. 4 (d), where the mark signs are the recorded time values, and the series are the trends for each configuration. A safe conclusion is that the reduction time depends mostly on the evoXML size. For small file sizes, the reduction performs the same for all versions. In addition, the increase rates in time are similar for both the current and the initial snapshot, for both s_1 and s_2. Therefore, the time instance parameter of the reduction process does not affect the reduction performance.

Concluding, both space and time efficiency are mostly affected by the average fan-out, which deteriorates as more changes are applied. That is mainly because of the evo:ref elements that are added for all children of an element that "evolves". Still, our approach is much more efficient than retaining separately every different version. Future optimizations will take into consideration the above and will aim to encode evo:ref elements and to the overall compression of the file.

IV. Related Work

Numerous approaches have been proposed for the management of evolving semistructured data. One of the early works [6] proposes DOEM, an extension of OEM capable of representing changes, such as *Create Node*, *Add Arc*, *Remove Arc* and *Update Node*, as annotations on the nodes and the edges of the OEM graph. In [10], the authors employ a *diff* algorithm for detecting changes between two versions of an XML document and storing them either as edit scripts or deltas. For each new version, they calculate the deltas with the previous and retain only the last version and the sequence of deltas. A similar approach is introduced in [7], where instead of deltas calculation, a referenced-based identification of each object is used across different versions. New versions hold only the elements that are different from the previous version whereas a reference is used for pointing to the unchanged elements of past versions. In [9] the authors propose MXML, an extension of XML that uses context information to express time and models multifaceted documents. Recently, there are works that deal with change modeling [15] and detection [13] in semantic data, in which the aforementioned problems are applied to ontologies and RDF.

Most approaches employ temporal extensions for the lifespan of different versions of documents. In [1], [6], the authors enrich data elements with temporal attributes and extend query syntax with conditions on the time validity of the data. In [14], the authors model an XML document as a directed graph, and attach transaction time information at the edges of the graph. Techniques for evaluating temporal queries on semistructured data are presented in [8], [18]. In [8] the authors propose a temporal query language for adding valid time support in XQuery. In [18] the notion of a temporally grouped data model is employed for uniformly representing and querying successive versions of a document. In [11], the authors extend this technique for publishing the history of a relational database in XML and employ a set of schema modification operators (SMOs) for representing the mappings between successive schema versions. In [1] the problem of archiving curated databases is addressed. The authors develop an archiving technique for scientific data that uses timestamps for each version, whereas all versions are merged into one hierarchy. This is in contrast with approaches that store a sequence of deltas and apply a large number of deltas for retrieving backwards the history of an element. Lastly, [5] deals with provenance in curated databases. All user actions for constructing a target database are recorded as sequences of insert, delete, copy and paste operations stored as provenance links from current data towards previous versions of the target database or external source databases.

Compared to the above approaches, our model introduces a change-based perspective for evolving data, in which changes are not derived by data versions but are modeled as first class citizens together with data. In our view, changes are not described through diffs or transformations with edit scripts between document versions, but are complex objects operating on data, and exhibit structural, semantic, and temporal properties. Change-centric modeling of evolving semistructured data can provide additional information about *what*, *why*, and *how* data has evolved over time.

V. Conclusions

In this paper, we showed how a data model called evo-graph can be used to progressively capture the structure of changes and the history of data. We believe that capturing structured changes within a data model enables a range of very useful queries on the provenance of data, and on the semantics of data evolution. We defined basic and complex changes over snap-graph, and described the process of building evo-graph step by step, as changes occur on the current snap-graph. We outlined C2D, a framework based on XML technologies that implements the ideas presented in this paper. We evaluated C2D using synthetic XML data for its space and time efficiency, and discussed the results.

Acknowledgment

This research has been co-financed by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

References

[1] T. Amagasa, M. Yoshikawa, S. Uemura, “A Data Model for Temporal XML Documents”, In DEXA 2000.

[2] P. Amornsinlaphachai , N. Rossiter and M. A. Ali, “Translating XML Update Language into SQL”, Journal of Computing and Information Technology, 2006, 2, 91–110.

[3] Berkeley DB XML. http://www.oracle.com/technetwork/database/berkeleydb/overview/index.html. 19 June 2012.

[4] P. Buneman, S. Khanna, K. Tajima, W.C. Tan, ”Archiving Scientific Data”, ACM Transactions on Database Systems, Vol. 20, pp 1-39, 2004.

[5] P. Buneman, A. P. Chapman, J. Cheney, “Provenance Management in Curated Databases”, In SIGMOD’06.

[6] S. Chawathe, S. Abiteboul, J. Widom, “Managing Historical Semistructured Data”, Journal of Theory and Practice of Object Systems, Vol. 24(4), pp.1-20, 1999.

[7] S-Y. Chien, V. J. Tsotras, C. Zaniolo, “Efficient Management of Multiversion Documents by Object Referencing”, In VLDB 2001.

[8] D. Gao, R. T. Snodgrass, “Temporal Slicing in the Evaluation of XML Queries”, In VLDB 2003.

[9] M. Gergatsoulis, Y. Stavrakas, “Representing Changes in XML Documents using Dimensions”, In 1st International XML Database Symposium, (XSym 2003).

[10] A. Marian, S. Abiteboul, G. Cobena, L. Mignet, “Change-Centric Management of Versions in an XML Warehouse”, In VLDB 2001.

[11] H.J. Moon, C. Curino, A. Deutsch, C.Y. Hou, C. Zaniolo, “Managing and querying transaction-time databases under schema evolution”, In VLDB 2008.

[12] National research council - Committee on Frontiers at the Interface of Computing and Biology. Catalyzing Inquiry at the Interface of Computing and Biology. Edited by J. C. Wooley, H. S. Lin., National Academies Press, 2005.

[13] V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, V. Christophides, “On Detecting High-Level Changes in RDF/S KBs”, In ISWC 2009.

[14] F. Rizzolo, A. A. Vaisman, “Temporal XML: modeling, indexing, and query processing”, In VLDB J. 17(5): 1179-1212 (2008).

[15] F. Rizzolo, Y. Velegrakis, J. Mylopoulos, S. Bykau, “Modeling Concept Evolution: a Historical Perspective”, In ER 2009.

[16] Y. Stavrakas, G. Papastefanatos, “Supporting Complex Changes in Evolving Interrelated Web Databanks”, In In CoopIS 2010.

[17] Y. Stavrakas, G. Papastefanatos, “Using Structured Changes for Elucidating Data Evolution”, In DaLi’11 (with ICDE 2011).

[18] F. Wang, C. Zaniolo, “Temporal Queries in XML Document Archives and Web Warehouses”, In TIME 2003.

[19] Xmlgener: Synthetic XML data generator. http://code.google.com/p/xmlgener/.

[20] XQuery Update Facility 1.0. http://www.w3.org/TR/xquery-update-10/, W3C Recommendation, 17 March 2011.

A Language for Defining and Detecting Complex Changes on RDF(S) Knowledge Bases

Theodora Galani, George Papastefanatos, Yannis Stavrakas

Institute for the Management of Information Systems, RC ATHENA, GREECE
{theodora, gpapas, yannis}@imis.athena-innovation.gr

Abstract. The dynamic nature of web data brings forward the need for maintaining data versions as well as identifying changes between them. In this paper, we deal with problems regarding understanding evolution, focusing on RDF(S) knowledge bases, as RDF is a de-facto standard for representing data on the web. We argue that revisiting past snapshots or the differences between them is not enough for understanding how and why data evolved, especially in cooperative environments. Instead, changes should be treated as first-class-citizens. In our view, this involves supporting semantic rich, user-defined changes that we call complex changes. In this paper, we present our perspective regarding complex changes, propose a declarative language for defining complex changes for RDF(S) knowledge bases, and show how this language is used to detect complex change instances among dataset versions.

Keywords: change management, data evolution, rdf(s)

1 Introduction

The increasing amount of information published on the web poses new challenges for data management. A central issue concerns evolution management, as the dynamic nature of data brings forward the need for maintaining data versions as well as identifying changes between them. Especially in cooperative environments, where dependencies between data appear, evolution management becomes more evident. For example, biologists rely on the web for publishing research results. They often use ontologies in order to curate their data from multiple domains of interest like anatomy, diseases, biomedical investigations, etc. These ontologies are frequently updated as errors may need to be fixed or new knowledge about the state of the art may need to be incorporated. As a result, curators of depending ontologies are interested in understanding the evolution history in order to learn more about the changes that have taken place on the respective domain of interest. In such cases, simply revisiting past snapshots and the differences between versions may not be enough.

As an example, consider Experimental Factor Ontology (EFO) [6] which provides a systematic description of many data elements available in EBI[1] databases, and for

[1] http://www.ebi.ac.uk/

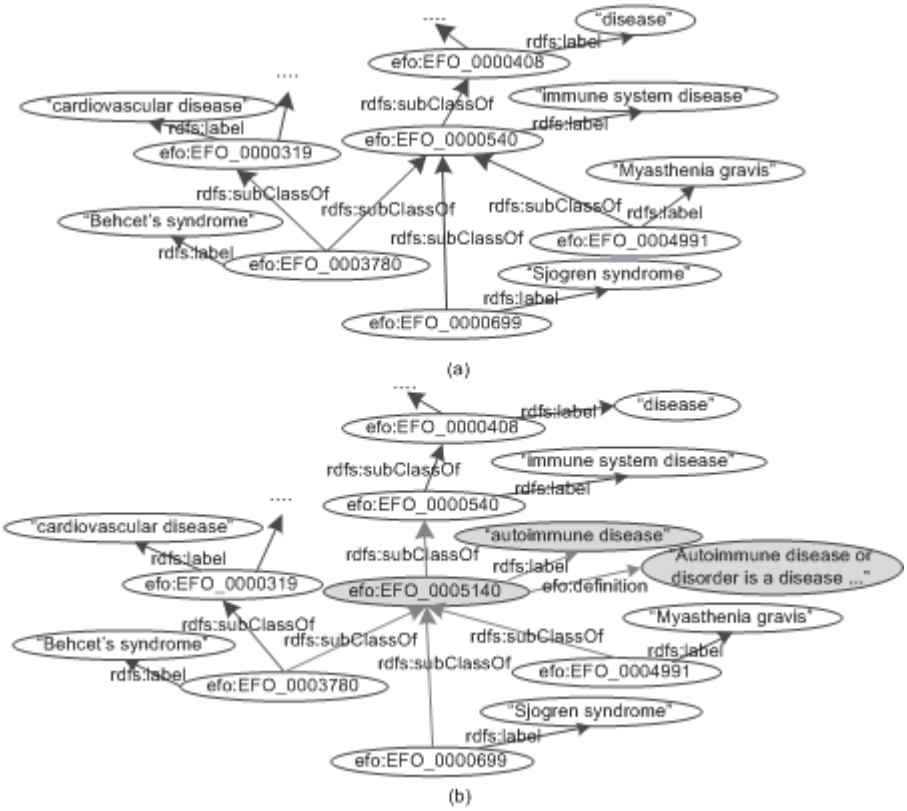

Fig. 1. Simplified part of EFO data. (a) The previous version. (b) The next version.

external projects. It combines parts of several biological ontologies and it is frequently updated as new terms are added, while others are changed or made obsolete. Terms in EFO are assigned with descriptive metadata, like labels, textual definitions, synonym labels and others. A new EFO version is released every month. External users and curators want to know what changed and how from one version to another.

Consider Fig. 1 as a simplified example. Fig. 1 (a) depicts a part of EFO regarding diseases, while Fig. 1 (b) the same part in the next published EFO version. Each term is represented by a class, which is annotated by a descriptive label and with other metadata which are not depicted for clarity. Also, there is a hierarchical organization of terms. In the previous version "Behcet's syndrome", "Sjogren syndrome" and "Myasthenia gravis" are types of "immune system diseases", which is a type of the generic term "diseases". In the next version, a new term is added. The modifications from one version to the other are depicted in grey color. The term efo:EFO_0005140 is added and annotated with the label "autoimmune diseases" and a definition. It is also posed into the hierarchy, being a generalization of "Behcet's syndrome", "Sjogren syndrome" and "Myasthenia gravis" and a specific type of "immune system diseases". This modification indicates that the knowledge regarding the classification of these

diseases has been reviewed. In this scenario, external users and curators would like to know that in the new version there is a new annotated term, which is also added into the hierarchy, or more specifically that it serves as a generalization of already existing terms constituting a new immune system disease category. Maintaining successive versions does not facilitate the discovery of what changed and why, especially in a large ontology. Even computing the differences between them as deltas, i.e. added and deleted triples, is not enough for understanding the change semantics. Instead, human readable changes should be supported to capture the meaning behind the modifications. Table 1 presents such changes. Notice that simple changes are primitive, while complex changes have richer semantics attempting to interpret user intention.

In this paper we argue that for understanding data evolution changes should be treated as first-class-citizens. In our view, this involves supporting semantically rich, user-defined changes that we call *complex changes*. Modeling complex changes explicitly can provide additional semantic information for interpreting past data. Defining complex changes for being detected between dataset versions allows interpreting evolution in multiple ways. We present the basic concepts regarding our perspective on complex changes. This involves that complex changes group simple changes, they may be interrelated constituting a hierarchical model, and they may be mutually exclusive. We provide a declarative language for defining complex changes, given these concepts. Also, we describe the process of detecting complex change instances among dataset versions. Note that although the notion of complex changes is not bound to any specific data model, we focus on RDF(S) knowledge bases, as RDF[2] is a de-facto standard for representing data on the web.

The paper outline is as follows. In section 2 we discuss related work. In section 3 we define the basic concepts of our approach on complex changes. In section 4 we present our language for defining complex changes and give examples. In section 5

Table 1. Simple and complex change instances on EFO data of Fig. 1.

<table>
<tr><th>Simple Changes</th><th colspan="4">Complex Changes</th></tr>
<tr><td>Add_Type_Class(efo:EFO_0005140)</td><td rowspan="2"></td><td rowspan="2">Add_Annotated_Class(
efo:EFO_0005140,
"autoimmune disease",
"Autoimmune disease or ...")</td><td rowspan="5">Add_Class_to_Hierarchy(
efo:EFO_0005140,
"autoimmune disease",
{efo:EFO_0003780,
efo:EFO_0000699,
efo:EFO_0004991},
{efo:EFO_0000540})</td><td rowspan="10">Add_Generalization_Class(
efo:EFO_0005140,
"autoimmune disease",
{efo:EFO_0003780,
efo:EFO_0000699,
efo:EFO_0004991})</td></tr>
<tr><td>Add_Label(efo:EFO_0005140, "autoimmune disease")</td></tr>
<tr><td>Add_Property_Instance(efo:EFO_0005140,
"Autoimmune disease or ...", efo:definition)</td><td>Add_Definition(
efo:EFO_0005140
, "Autoimmune
disease or ...")</td></tr>
<tr><td>Add_Superclass(efo:EFO_0005140,
efo:EFO_0000540)</td><td rowspan="7"></td><td rowspan="7"></td></tr>
<tr><td>Add_Superclass(efo:EFO_0003780,
efo:EFO_0005140)</td></tr>
<tr><td>Add_Superclass(efo:EFO_0000699,
efo:EFO_0005140)</td></tr>
<tr><td>Add_Superclass(efo:EFO_0004991,
efo:EFO_0005140)</td></tr>
<tr><td>Delete_Superclass(efo:EFO_0003780,
efo:EFO_0000540)</td><td rowspan="3"></td></tr>
<tr><td>Delete_Superclass(efo:EFO_0000699,
efo:EFO_0000540)</td></tr>
<tr><td>Delete_Superclass(efo:EFO_0004991,
efo:EFO_0000540)</td></tr>
</table>

[2] http://www.w3.org/TR/rdf-primer/

we present the detection process for complex changes defined by the presented language. Finally, in section 6 we conclude the paper.

2 Related work

A number of works focus on computing the differences between knowledge bases. In [2] an ontology for representing differences, in the form of insertions and deletions, between RDF graphs is proposed. The problem of comparing RDF graphs is discussed, as well as updating a graph from the set of differences. In [13] two diff algorithms are proposed: (i) one computing a structural diff which is the set-based difference of the triples explicitly recorded into the two graphs, and (ii) one semantic diff which also takes into consideration the semantically inferred triples. In [3], an approach for computing a semantic diff between knowledge bases is proposed for reporting differences in logical meaning. It focuses on propositional logic knowledge bases, but it is also applicable to more expressive logics. A number of desired properties are discussed, like the uniqueness of the semantic diff, the principal of minimal change, the ability to undo changes and version reconstruction. Similar properties are also supported in [14], which focuses on computing deltas over RDF(S) knowledge bases. In [7, 5], a fixed point algorithm for detecting ontology change is presented. The algorithm employs heuristic-based matchers, introducing uncertainty to results.

Other works focus on supporting human-readable changes. In [9], a set of predefined high-level changes for RDF(S) knowledge bases is proposed, as well as an algorithm for their detection. The proposed changes verify the useful properties of completeness and unambiguity, for guaranteeing that every added or deleted triple is consumed by one detected high-level change and that detected high-level changes are not overlapping respectively. In [11], an extension of [9] is proposed, by providing a more generic change definition framework, based on SPARQL[3] queries, as well as an ontology of changes. In [10], Change Definition Language is proposed in order to define and detect changes over a version log using temporal queries. [1] discusses a framework for supporting evolution in RDF knowledge bases. Changes are additions and deletions of triples, as well as aggregated triples, resulting in a hierarchy of changes. However, neither a detection process, nor a specific language of changes is defined. In [5], an extension of [7] is proposed for detecting some of the proposed basic and composite changes. In general, in [5], [9] and [12] a number of human readable changes in similar categorizations given granularity and semantics are presented.

Our approach focuses on human readable changes. A short visionary work was presented in [4]. Similar to [5], [9] and [12] we assume primitive changes, which we call simple changes, and groupings of them, which we call complex changes. For simple changes we rely on [9]. Our contribution concerns our perception on complex changes. Instead of providing a predefined list of complex changes, we propose a language for arbitrarily defining them in order to capture richer semantics and multiple interpretations of evolution. Unlike [10], complex changes are defined over simple

[3] http://www.w3.org/TR/sparql11-query/

changes or other complex changes, resulting in a hierarchical construct. Also, complex changes may be defined as mutually exclusive, for supporting alternative perceptions of evolution. These features set us apart from [9] and [11]. Finally, we provide a mechanism for detecting defined complex changes.

3 Simple and complex changes

Modeling changes as first class citizens involves taking into consideration two basic issues: granularity and semantics of changes. Granularity of changes poses the question of having fine-grained or coarse-grained changes. Fine-grained changes have the advantage of describing primitive changes, while coarse-grained changes provide semantics and conciseness by grouping primitive changes in logical units. Semantics of changes poses the question of having model-specific or data- and application- specific changes. Model-specific changes describe modifications that may appear in a specific representation model. They constitute a fixed set of generic changes. Data- or application- specific changes represent changes that suit on specific use-case scenarios and may be user-defined, allowing multiple interpretations of evolution.

In order to tackle the aforementioned issues, we distinguish between simple and complex changes. Simple changes constitute a fixed set of fine-grained, model-specific changes. Complex changes are coarse-grained, user-defined, application-specific changes.

Definition 1 provides a formal definition for simple changes. Table 2 summarizes the simple changes we assume. They are defined in [9], verifying completeness and unambiguity, constituting a first layer of human-readable changes.

Definition 1: A simple change is a tuple $(n, \boldsymbol{p})$, where:

- n is the simple change name, which must be unique.
- $\boldsymbol{p}$ is a list of parameters of the simple change, where each one is defined with a name p_i which is unique within the change.

Simple changes are additions, deletions and terminological changes (rename, split, merge) of RDF(S) identified entities, which are classes, properties and individuals. As it is already stated simple changes are fine-grained, meaning that they cannot be decomposed into more granular changes. This holds for the changes in the form of additions/deletions, but not for terminological changes, as they can be expressed in terms of additions/deletions plus some extra conditions. For example, a class rename can be considered as an add class plus a delete class, where these two classes have the same "neighborhood" (defined properties, connections to classes). However, we prefer them to be considered as simple changes in order to distinguish at simple change level "real" additions and deletions from "virtual" ones representing terminological changes. As a result, simple changes' set is not minimal.

An actual simple change instance is identified through a detection process (as described in [9]) and is defined as the instantiation of the change parameters with actual values. As an example of simple change instances consider the ones given in Table 1.

Table 2. Simple Changes on RDF(S) knowledge bases.

Add_Type_Class(a)	Add object a of type rdfs:class.
Delete_Type_Class(a)	Delete object a of type rdfs:class.
Rename_Class(a)	Rename class a to b.
Merge_Classes(A, b)	Merge classes contained in A into b.
Merge_Classes_Into_Existing(A, b)	Merge classes contained in A into b, where b∈A.
Split_Class(a, B)	Split class a into classes contained in B.
Split_Class_Into_Existing(a, B)	Split class a into classes contained in B, where a∈B.
Add_Type_Property(a)	Add object a of type rdf:property.
Delete_Type_Property(a)	Delete object a of type rdf:property.
Rename_Property(a, b)	Rename property a to b.
Merge_Properties(A, b)	Merge properties contained in A into b.
Merge_Properties_Into_Existing(A, b)	Merge properties contained in A into b, where b∈A.
Split_Property(a, B)	Split property a into properties contained in B.
Split_Property_Into_Existing(a, B)	Split property a into properties contained in B, where a∈B.
Add_Type_Individual(a)	Add object a of type rdfs:resource.
Delete_Type_Individual(a)	Delete object a of type rdfs: resource.
Rename_Individual(a, b)	Rename individual a to b.
Merge_Individuals(A, b)	Merge individuals contained in A into b.
Merge_Individuals_Into_Existing(A, b)	Merge individuals contained in A into b, where b∈A.
Split_Individual(a, B)	Split individual a into individuals contained in B.
Split_Individual_Into_Existing(a, B)	Split individual a into individuals contained in B, where a∈B.
Add_Superclass(a, b)	Parent b of class a is added.
Delete_Superclass(a, b)	Parent b of class a is deleted.
Add_Superproperty(a, b)	Parent b of property a is added.
Delete_Superproperty(a, b)	Parent b of property a is deleted.
Add_Type_To_Individual(a, b)	Type b of individual a is added.
Delete_Type_From_Individual(a, b)	Type b of individual a is deleted.
Add_Property_Instance(a_1, a_2, b)	Add property instance of property b.
Delete_Property_Instance(a_1, a_2, b)	Delete property instance of property b.
Add_Domain(a, b)	Domain b of property a is added.
Delete_Domain(a, b)	Domain b of property a is deleted.
Add_Range(a, b)	Range b of property a is added.
Delete_Range(a, b)	Range b of property a is deleted.
Add_Comment(a, b)	Comment b of object a is added.
Delete_Comment(a, b)	Comment b of object a is deleted.
Change_Comment(u, a, b)	Change comment of resource u from a to b.
Add_Label(a, b)	Label b of object a is added.
Delete_Label(a, b)	Label b of object a is deleted.
Change_Label(u, a, b)	Change label of resource u from a to b.

Definition 2: A simple change instance of a simple change $(n, \boldsymbol{p})$, is a tuple $(n, \boldsymbol{v})$ where $\boldsymbol{v}$ is the instantiation of the parameters in $\boldsymbol{p}$.

In many cases, many simple changes are performed together, formulating a logical unit. In such cases, simple changes are not adequate for understanding evolution, as they are too granular. Also, understanding evolution may vary depending on the user's perspective. For these reasons, we introduce complex changes, as user-defined changes that group simple or other complex changes. Definition 3 provides a formal definition for complex changes.

Definition 3: A complex change is a quadruple $(n, \boldsymbol{p}, D, F)$, where:

- n is the complex change name, which must be unique and different from the reserved simple change names.
- $\boldsymbol{p}$ is the list of parameters of the complex change, where each one is defined with a name p_i which is unique within the change.
- D is a set of simple (D_S) and complex changes (D_C) that the complex change comprises of, where $D = D_C \cup D_S$, and $D \neq \emptyset$.
- F is a set of constraints (F_C) and bindings (F_B). Constraints have to be verified in order a complex change to be detected. There are two types of constraints defined over changes: (i) cardinality constraints, (ii) exclusiveness constraints. There are three types of constraints defined over parameters of changes participating in D: (i) testing value constraints, (ii) pre-conditions and post-conditions, (iii) relational constraints. Bindings indicate how complex change parameters are evaluated.

Complex changes that are defined in terms of simple changes only are considered to be of level 0. Those that are defined in terms of simple changes and complex changes of level 0 are considered to be of level 1, etc.

Defined constraints (F_C) specialize the meaning of the complex change. A cardinality constraint states the number of actual change instances of a specific type the complex change comprises of. An exclusiveness constraint states whether the respective change instances can be shared with others. A testing value constraint poses a value constraint on a parameter, while a relational constraint interrelates change parameters. Pre- and post- conditions state conditions that should hold in the version before or after the complex change respectively.

Each defined complex change can be evaluated over a set of simple change instances and respective dataset versions, resulting in possible complex change instances. The process of calculating possible complex change instances constitutes complex change detection. Definition 4 defines complex change instances. As an example of complex change instances consider those in Table 1. Definition 5 defines when a complex change instance is detected.

Definition 4: A complex change instance of a complex change $(n, \boldsymbol{p}, D, F)$, is a tuple $(n, \boldsymbol{v})$ where $\boldsymbol{v}$ is the instantiation of the parameters in $\boldsymbol{p}$.

Definition 5: Let S be a set of simple changes and S_i respective simple change instances for two dataset versions V_b and V_a. Also, let $c = (n, \boldsymbol{p}, D, F)$ be a defined complex change.

- A complex change instance $c_i = (n, \boldsymbol{v})$, where $D_c = \emptyset$, is detected over S_i iff there is a mapping $m: S \rightarrow S_i$ such that $m(D) \in S_i$ and F evaluates true for $m(D)$, V_b and V_a.

- A complex change instance $c_i = (n, \boldsymbol{v})$, where $D_c = C$ a set of defined complex changes and C_i respective complex change instances, is detected over $S_i \cup C_i$ iff there is a mapping $m: S \cup C \rightarrow S_i \cup C_i$ such that $m(D) \in S_i \cup C_i$ and F evaluates true for $m(D)$, V_b and V_a.

Complex change instances may be interrelated, as a complex change may be defined on another or share a common part. On the other hand, complex change instances may not be allowed to be interrelated due to exclusiveness constraints. Consequently, instances become mutually exclusive resulting in alternative results. Definitions 6, 7 and 8 define possible relations between change instances.

Definition 6: Let c_i be a complex change instance of a complex change $c = (n, \boldsymbol{p}, D, F)$.

- We say that c_i contains the simple and complex change instances in $m(D)$.
- We say that c_i contains *exclusively* the simple and complex change instances in $m(D)$ for which exclusiveness constraints are defined in F.

Note that the property of (exclusive) containment is transitive. Therefore, a complex change instance c_i containing (exclusively) a complex change instance c_i' contains (exclusively) all the change instances contained by c_i' too.

Definition 7: Let c_i and c_i' be two complex change instances that c_i does not contain c_i' and vice versa. They are conflicting iff they both contain at least one common simple or complex change instance which must be exclusively contained in at least one of them.

Definition 8: Let c_i and c_i' be two complex change instances that c_i does not contain (exclusively) c_i' and vice versa. They are overlapping iff they both contain at least one common simple or complex change instance and they are not conflicting.

Notice that complex change instances may construct a hierarchy over simple change instances, as one may contain or overlap with another, while conflicts may lead to alternative sets of detected instances. Note that the relationship between complex change instances given in Table 1 depends on how complex changes are defined and will be further clarified in next section.

4 A language for defining complex changes

In this section we provide a declarative language for defining complex changes. We provide the syntax by means of its EBNF specification and a number of examples in order to illustrate its usage.

Complex change definition. It is always composed by a heading and a body. The heading contains the complex change name and a parameter list. The parameter list is a non-empty, comma separated list of parameter names. The body contains: (i) a list of changes that the complex change comprises of, (ii) parameter bindings declaring how complex change parameters are evaluated (which are optional) and (iii) constraints on the parameters of changes that appear in the change list (which are optional too). Regarding constraints, in the selection filter list testing value constraints are

listed, in the relational filter list relational constraints are listed and in the version filter list pre- and post- conditions are listed.

```
complex-change-definition = 'CREATE COMPLEX CHANGE' heading '{' body '}' ';' ;
heading = name '(' parameter-list ')' ;
parameter-list = identifier { ', ' identifier } ;
body = change-list [';' binding-list] [ ';' selection-filter-list ] [ ';' relational-filter-list ] [ ';' version-filter-list ] ';' ;
name = STRING ;
identifier = LETTER { LETTER | DIGIT } ;
```

Change list. It comprises of the changes that the defined complex change groups. The change list is a non-empty, comma separated list of changes plus optionally some declarations which are attributed to each change. These declarations pose cardinality or exclusiveness constraints. Each change is declared by its heading. Simple changes or other already defined complex changes may be in the list.

```
change-list = 'CHANGE LIST' change { ', ' change } ;
change = change-heading [ cardinality ] [ exclusiveness ] ;
change-heading = change-name '(' parameter-list ')' ;
change-name = name | NAMES OF SUPPORTED SIMPLE CHANGES ;
```

Cardinality constraints. They define the number of change instances of a specific type that have to be grouped into a complex change. Posing a cardinality constraint is optional, assuming that if it is not defined the default case is one change instance. The following cases are distinguished: (i) at least one change ("+"), (ii) zero or one change ("?"), (iii) zero or more changes ("*"). Allowing zero changes implies that the specific change is optional, i.e. it might not be detected while the complex change is still possible to be detected. Therefore, a complex change definition is tolerant in partially performed modifications that are considered of minor significance.

```
cardinality = '+' | '?' | '*' ;
```

Exclusiveness constraints. They define whether the instances of a change can be contained by multiple complex changes instances. The following are distinguished: (i) If an instance c_i of a complex change c contains exclusively an instance s_i of a simple change s, then s_i cannot be contained in any other instance c_i' of any complex change, except from those c_i' that contain c_i too. (ii) If an instance c_i of a complex change c contains an instance s_i of a simple change s without explicitly declaring an exclusiveness constraint, then s_i is considered non-exclusively contained. (iii) If an instance c_i of a complex change c contains exclusively an instance c_i' of a complex change c', then it contains exclusively all change instances contained in c_i' too. (iv) If an instance c_i of a complex change c contains an instance c_i' of a complex change c' without explicitly declaring an exclusiveness constraint, then for all change instances contained in c' the exclusiveness constraints declared in c' are considered.

```
exclusiveness = 'ex' ;
```

Binding List. It comprises of rules that define how the complex change parameters are evaluated. A change parameter may evaluate either into a scalar value or a set of values. Despite that parameters evaluating into scalar values may be considered as evaluating into a set of cardinality equal to one, we make the above assumption in order to be able to write conditions on these parameters in a more intuitive and concise way. In order to distinguish the parameter types, parameters that evaluate into

scalar values should start with a lowercase letter, while those that evaluate into sets should start with an uppercase letter.

A complex change parameter may be equal to another, which is defined into the body as a parameter of a contained change. In case of a parameter evaluating into a set, it may also be equal to the union of the parameter values of a change with a cardinality constraint allowing multiple instances (+, *). Parameter bindings are optional, in case they can be inferred by repeating each parameter into the contained changes and respective constraints.

```
binding-list = 'BINDING LIST' binding { ', ' binding } ;
binding = binding-equality | binding-union ;
binding-equality = identifier '=' identifier ;
binding-union = 'for each' identifier ':' identifier '=' identifier 'union' identifier ;
```

Testing value constraints. They restrict a parameter value against given constants.

- For a scalar parameter p a testing value constraint is in the form: $f(p)$, where f is a boolean expression that may contain (i) binary operators ($>, \geq, <, \leq, =, \neq$) between p and a constant c, (ii) predefined functions (e.g. for posing constraints on strings) over p, (iii) logical and, or, not.
- For a parameter P that evaluates into a set of values a testing value constraint is in the form: $F(P)$, where F is a boolean expression that may contain (i) binary set operators ($\supset, \supseteq, \subset, \subseteq, =, \neq$) relating P with a given set S, (ii) existential or universal quantifiers, (iii) logical and, or, not. Quantifiers are used to write constraints on set's elements, which actually are scalar values. Therefore, suitable constraints on set's elements are those for scalar parameters.

In each selection-filter a cardinality prefix may be optionally defined (notice "card-prefix" in the selection filter fragment in the following EBNF part). This is useful if the constraint is defined over a parameter of a change with cardinality constraint '+' or '*'. In this case, the constraint may be verified for each, some or any of the instances. Examples of testing value constraints are the following: $for\ each\ x: x > 5$, $x > 5\ ||\ x < 2$, $x = prefix{:}uri$, $X \subseteq \{uri1, uri2\}$, $for\ any\ in\ X: x > 5$, $containts(x, "value")$, etc.

```
selection-filter-list = 'SELECTION FILTER' selection-filter { ', ' selection-filter } ;
selection-filter = [ card-prefix ] [ quantifier ] sel-or-expr ;
card-prefix = 'for' ('each' | 'some' | 'any') identifier ':' ;
quantifier = 'for' ('each' | 'some' | 'any') identifier ('in' | 'not in') identifier ':' ;
sel-or-expr = sel-and-expr { '||' sel-and-expr } ;
sel-and-expr = sel-neg-expr { '&&' sel-neg-expr } ;
sel-neg-expr = sel-expr | ('!' sel-expr ) ;
sel-expr = sel-bracketed-expr | sel-bin-expr | sel-function ;
sel-bracketed-expr = '(' sel-or-expr ')' ;
sel-bin-expr = identifier bin-operator constant ;
bin-operator = '=' | '!=' | '>' | '<' | '>=' | '<=' | 'subSet' | 'properSubset' | 'superSet' | 'properSuperset' ;
constant = set | value ;
set = '{' value-list '}' ;
value-list = value { ', ' value } ;
```

```
value = URI | LITERAL ;
sel-function = CALL OF PREDEFINED FUNCTIONS ;
```

Relational constraints. They restrict a parameter value by relating it with another parameter.

- For two scalar parameters p_1 and p_2 a relational constraint is in the form: $f(p_1, p_2)$, where f is a boolean expression having the same expressiveness with testing value constraints on scalar parameters.
- For two parameters P_1 and P_2 evaluating into sets, a relational constraint is in the form $F(P_1, P_2)$, where F is a boolean expression having the same expressiveness with testing value constraints on parameters evaluating into sets. Note that quantifiers can be used to write conditions interrelating sets element by element.
- For a scalar parameter p_1 and a parameter P_2 evaluating into a set a relational constraint is in the form $p_1 \in P_2$ or $p_1 \notin P_2$.

As in selection filters, cardinality prefixes ("card-prefix-list" fragment) may be optionally defined. Examples of relational constraints are the following: $for\ each\ x: x = \mathrm{y}$, $for\ each\ X: for\ each\ Y: X \subseteq \mathrm{Y}$, $x\ in\ X\ and\ x\ not\ in\ Y$, $containts(x, y)$, etc.

```
relational-filter-list = 'RELATIONAL FILTER' relational-filter { ', ' relational-filter } ;
relational-filter = [ card-prefix-list ] [ quantifier-list ] rel-or-expr ;
card-prefix-list = 'for' ('each' | 'some' | 'any') identifier ':' { 'for' ('each' | 'some' | 'any') identifier ':' } ;
quantifier-list = 'for' ('each' | 'some' | 'any') identifier ('in' | 'not in') identifier ':' { 'for' ('each' | 'some' | 'any')
identifier ('in' | 'not in') identifier ':' } ;
rel-or-expr = rel-and-expr { '||' rel-and-expr } ;
rel-and-expr = rel-neg-expr { '&&' rel-neg-expr } ;
rel-neg-expr = rel-expr | ('!' rel-expr ) ;
rel-expr = rel-bracketed-expr | rel-bin-expr | rel-function ;
rel-bracketed-expr = '(' rel-or-expr ')' ;
rel-bin-expr = identifier rel-bin-operator identifier ;
rel-bin-operator = bin-operator | 'in' | 'not in' ;
rel-function = CALL OF PREDEFINED FUNCTIONS ;
```

Pre- and post- conditions. They state that a triple must or must not exist in the version before or after the change. In place of subject, predicate or object scalar parameters may be. In case of parameters evaluating into sets, conditions on their elements are supported, using quantified expressions. Note that, some triples may be inferred in a version. The flag "inferred" is optionally defined for declaring that the triple may not be explicitly stated in a dataset version. In case inference is not explicitly stated, the default operation is not checking the inferred triples. Examples of pre- and post-conditions are the following: $\{x, predicate, y\}\ inferred\ in\ Va$, $for\ each\ x\ in\ X: for\ each\ y\ in\ Y: (x, prefix, y) not\ in\ Vb$.

```
version-filter-list = 'VERSION FILTER' version-filter { ', ' version-filter } ;
version-filter = [ card-prefix-list ] [ quantifier-list ] ver-or-expr ;
ver-or-expr = ver-and-expr { '||' ver-and-expr } ;
ver-and-expr = ver-neg-expr { '&&' ver-neg-expr } ;
ver-neg-expr = ver-expr | ('!' ver-expr ) ;
```

```
ver-expr = ver-bracketed-expr | ver-tr-expr ;
ver-bracketed-expr = '(' ver-or-expr ')' ;
ver-tr-expr = '{' triple '}' [ inference ] ('in' | 'not in') ('Vb' | 'Va') ;
inference = 'inferred' ;
triple = '(' id-or-val ', ' id-or-val ', ' id-or-val ')' ;
id-or-value = identifier | value ;
```

Next we provide examples of complex change definitions. Table 3 summarizes the definitions of the complex changes regarding Fig.1 and Table 1, while Table 4 provides some alternative definitions. First we discuss changes of Table 3.

Add_Definition models the case where a new definition property with value *d* is assigned to a class *c*. It comprises of a simple change *Add_Property_Instance* and a selection filter restricting the parameter type. Note that any binding is defined, as they are inferred by repeating the complex change parameters into the body.

Add_Annotated_Class models the case where a new class is added and annotated with a label and optionally with a definition property. It comprises of the simple changes *Add_Type_Class* and *Add_Label*, and optionally the complex change *Add_Definition*, as it is denoted by the "?" cardinality constraint. Instead of writing relational constraints, the parameter *c* is repeated among *Add_Type_Class*, *Add_Label* and *Add_Definition*, indicating that they all refer to a specific class.

Add_Class_to_Hierarchy models the case where an annotated class is added and positioned into a class hierarchy, having at least one superclass and one subclass. It groups *Add_Annotated_Class* and multiple instances of *Add_Superclass* simple changes that connect the added class to its superclasses and subclasses. The fact that the added class may be connected to one or more superclasses and subclasses is denoted by a "+" cardinality constraint besides *Add_Superclass*. For the complex change parameters *Superclasses* and *Subclasses* suitable bindings are defined, evaluating into the set of all superclasses (*supC*) and subclasses (*subC*) of *c* respectively.

Move_Property_to_Upper_Class models the case where a common property of a set of classes is moved to a common ancestor class, and it is used in the definition of *Add_Generalization_Class*. It comprises the simple change *Add_Property_Instance* and one or more instances of *Delete_Property_Instance*, one for each class being annotated with the specific property and value. Notice that a version filter is defined, declaring that the classes which are initially annotated with the moved property are subclasses of the new class holding the property. The keyword "*inferred*" states that the property may be moved to an ancestor class rather a direct parent class.

Add_Class_Generalization models the case where a newly added class serves as a generalization class for a set of *Subclasses*. Initially, each subclass was connected to each superclass. This is expressed by a pre-condition constraint in the first version filter. This is not the case after the change, as these connections are deleted. This is expressed by a post-condition constraint in the second version filter, while also *Delete_Superclass* simple changes for the respective connections are considered. Notice the relational constraints that limit grouped *Delete_Superclass* changes to those referring to mentioned connections. The subclasses are connected through the generalization class with the superlasses, as it is shown by the *Add_Class_to_Hierarchy* complex change. Also, any common parameters of the subclasses are optionally moved to

Table 3. Complex change definitions on EFO data of Fig. 1.

CREATE COMPLEX CHANGE Add_Definition (c, d) { CHANGE LIST Add_Property_Instance(c, d, p); SELECTION FILTER p='efo:definition'; };
CREATE COMPLEX CHANGE Add_Annotated_Class (c, l, d) { CHANGE LIST Add_Type_Class(c), Add_Label(c, l), Add_Definition(c, d) ?; };
CREATE COMPLEX CHANGE Add_Class_to_Hierarchy (c, l, Subclasses, Superclasses) { CHANGE LIST Add_Annotated_Class (c, l, d), Add_Superclass (c, supC) +, Add_Superclass (subC, c) +; BINDING LIST for each supC: Superclasses= Superclasses union supC, for each subC: Subclasses= Subclasses union subC; };
CREATE COMPLEX CHANGE Move_Property_to_Upper_Class (C, a, p) { CHANGE LIST Add_Property_Instance (a, p, v), Delete_Property_Instance (c, p, v) +; BINDING LIST for each c: C=C union c; VERSION FILTER for each c: {(c, rdfs:subClassOf, a)} inferred in Va; };
CREATE COMPLEX CHANGE Add_Class_Generalization (c, l, Subclasses) { CHANGE LIST Add_Class_to_Hierarchy (c, l, Subclasses, Superclasses), Move_Property_to_Upper_Class (Subclasses, c, p) *, Delete_Superclass (subC, supC) +; RELATIONAL FILTER for each subC: subC in Subclasses, for each supC: supC in Superclasses; VERSION FILTER for each x in Subclasses: for each y in Superclasses: {(x, rdfs:subClassOf, y)} in Vb, for each x in Subclasses: for each y in Superclasses: {(x, rdfs:subClassOf, y)} not in Va; };

the generalization class. This is stated by assuming zero or more ("*") *Move_Property_to_Upper_Class* complex changes. Suitable parameters are used to demonstrate the initial and final classes holding the property.

Note that *Add_Class_Generalization* definition could be more flexible or restrictive regarding the pre- and post-conditions of superclasses and subclasses. A more flexible alternative would be if some of the *Superclasses* are connected to some of the *Subclasses* in the previous version. Thus, the change may be detected even if there are new superclasses and subclasses of the generalization class c added in the next version. In this case, the first version filter should be altered into: *for some x in Subclasses: for some y in Superclasses: {(x, rdfs:subClassOf, y)} in Vb*. A more restrictive alternative would be if all the subclasses are not connected to any other supeclass except from those in *Superclasses*. In this case, the following version filter should be included in the definition: *for any x in Subclasses: for any y not in Superclasses: {(x, rdfs:subClassOf, y)} in Vb*. A similar version filter should hold for *Va*.

Given these definitions, the complex change instances in Table 1 can be detected. A hierarchy of complex change instances is constructed: *Add_Generalization_Class* instance contains *Add_Class_to_Hierarchy* instance, which in turn contains *Add_Annotated_Class* instance, which contains *Add_Definition* instance.

Another approach would be reporting alternative change instances, instead of a hierarchy. In this way alternative interpretations of evolution are provided. For example, it would be preferable to receive either the addition of an annotated class into a hierarchy of classes or the addition of a generalization class. Therefore, the instances *Add_Class_to_Hierarchy* and *Add_Generalization_Class* should be reported as alternatives. This is feasible by modifying the definitions of *Add_Annotated_Class* and

Table 4. Alternative complex change definitions on EFO data of Fig. 1.

CREATE COMPLEX CHANGE Add_Annotated_Class (c, l, d) { CHANGE LIST Add_Type_Class(c) ex, Add_Label(c, l), Add_Definition(c, d) ?; };
CREATE COMPLEX CHANGE Add_Class_Generalization (c, l, Subclasses) { CHANGE LIST Add_Annotated_Class (c, l, d), Add_Superclass (c, supC) +, Add_Superclass (subC, c) +, Move_Property_to_Upper_Class (SubC, c, p) *, Delete_Superclass (sub, sup) +; BINDING LIST for each subC: Subclasses= Subclasses union subC; RELATIONAL FILTER for each sub: for some subC: sub=subC, for each sup: for some supC: sup=supC for each subC: for each SubC: subC in SubC; VERSION FILTER for each subC: for each supC: {(subC, rdfs:subClassOf, supC)} in Vb, for each subC: for each supC: {(subC, rdfs:subClassOf, supC)} not in Va; };

Add_Generalization_Class as in Table 4. *Add_Type_Class* is contained exclusively into *Add_Annotated_Class*, given the exclusiveness constraint "*ex*" besides the change. *Add_Class_Generalization* is defined in terms of *Add_Annotated_Class*, while the rest changes of Table 3 remain immutable. As a result, the instances of *Add_Annotated_Class*, *Add_Class_to_Hierarchy* and *Add_Class_Generalization* contain exclusively the *Add_Type_Class* instance. Also, *Add_Class_to_Hierachy* instance and *Add_Class_Generalization* instance are conflicting and constitute alternative instances when referring to the same added class c.

5 Complex change detection

Complex change detection is the process of identifying complex change instances between two dataset versions. It is divided into two parts. The first part detects complex change instances, taking into consideration the complex change definition excluding exclusiveness constraints. This process requires as input a set of simple change instances between two dataset versions (for this we rely on [9]), as well as the actual dataset versions. The second part identifies conflicting complex change instances and alternative sets of complex change instances, taking into account the detected instances of the first step and exclusiveness constraints.

Regarding the first part, we consider that detected change instances are represented in an RDF graph. For each change instance the change type, its parameters and respective values, as well as (exclusively) contained change instances are recorded (similarly to [11]). Each complex change definition is evaluated over the RDF graph of already detected instances and the version before or after the changes.

Our first baseline approach for detecting the instances of a complex change c over an RDF graph of detected simple change instances O, given two RDF(S) graphs V_b and V_a of dataset versions is described in Algorithm 1. The instances of each change in the definition of c are selected one-by-one via a SPARQL query on O, taking into consideration selection and version filters defined on its parameters. In case of complex changes c' in the definition of c, the algorithm repeats recursively for c'. The selected instances are combined given the relational and version filters defined between their parameters, resulting in the complex change instances.

Algorithm 1. Detect complex change instances of c - Baseline approach

INPUT: Definition of a complex change c, RDF graph of detected simple change instances O, RDF(S) dataset versions V_b and V_a

OUTPUT: RDF graph I_c of detected complex change instances of c

Step 1. If there are simple changes in the definition of c, then do Step 1.1, else proceed in Step 2.

Step 1.1. For each simple change s in the definition of c, select all simple change instances of type s from O that verify the selection and version filters defined only on the parameters of s. Let I_s be the RDF graph of the selected instances of type s.

Step 2. If there are complex changes in the definition of c, then do Step 2.1-2.2, else proceed in Step 3.

Step 2.1. For each complex change c' in the definition of c, if the respective complex change instances $I_{c'}$ are computed, continue in Step 2.2. Else repeat Step 1 for each c', considering c as c'.

Step 2.2. For each complex change c' in the definition of c, select all complex change instances from $I_{c'}$ that verify the selection and version filters defined only on the parameters of c'. Delete the initial $I_{c'}$ and let $I_{c'}$ be the RDF graph of the selected instances.

Step 3. Combine the graphs of selected change instances $\{I_1, I_2, I_3, \dots, I_n\}$.

Step 3.1. Select from I_1 and I_2 all the change instances that verify the relational and version filters defined only on the parameters of the respective changes. Delete the initial I_1 and I_2 and let I_{12}^1, I_{12}^2, ..., I_{12}^m be the RDF graphs where each one contains the selected change instances from I_1 and I_2 that verify the constraints.

Step 3.2. Repeat Step 3 for each $\{I_{12}^1, I_3, \dots, I_n\}$, $\{I_{12}^2, I_3, \dots, I_n\}$, ..., $\{I_{12}^m, I_3, \dots, I_n\}$ until all the graphs per set are combined into one graph $I_{12\dots n}^i$.

Step 4. For each combined graph $I_{12\dots n}^i$ create a complex change instance, evaluating its parameters given the bindings in its definition and the change instances it contains which are in $I_{12\dots n}^i$.

Step 5. Return I_c, the graph of the detected complex changes instances of c.

An optimized approach is given in Algorithm 2. It proposes selecting as many changes in c as possible in a combined manner from O and RDF graphs I_c of already detected complex change instances, for fetching the change instances that verify selection, relational and version filters defined on the respective parameters. Then the intermediate results are combined. In this way, the number of queries is reduced, by posing more complex queries. However, a question is whether this method is applicable to all possible defined complex changes. In order to answer this question we have to consider how changes are grouped given cardinality constraints and prefixes. For example, the following cases are suitable for selecting all together the respective changes, taking into account the constraints between them.

1. A change c with cardinality 1 and a set of changes C where each one has cardinality 1 or "?". There might be further constraints between changes in C.
2. A change c with cardinality 1 and a set of changes C where each one has cardinality "+" or "*" and its parameters are related to the parameters of c only with prefixes "for each" or "for any". There might be further constraints over the changes in C or between them.
3. A change c with cardinality "+" or "*" and a set of changes C where each one has cardinality "+" or "*" and its parameters are related to the parameters of c only with prefixes "for each" or "for any". There might be further constraints over the changes in C or between changes in C.

However, typically a set of complex changes C are to be detected. In order to detect all complex changes in C, Algorithm 1 or 2 must run for each complex change in C. Consider that some complex changes have common patterns in their definitions. This means that they may comprise of same changes, or have the same constraints on their parameters, or may be interrelated with the same relational constraints. A typical

Algorithm 2. Detect complex change instances of c - Grouping changes

INPUT: Definition of a complex change c, RDF graph of detected simple change instances O, RDF(S) dataset versions V_b and V_a

OUTPUT: RDF graph I_c of detected complex change instances of c

Step 1. If there are complex changes in the definition of c, then do Step 1.1, else proceed in Step 2.1.

Step 1.1. For each complex change c' in the definition of c, if the respective complex change instances $I_{c'}$ are computed, continue in Step 2.1. Else repeat Step 1 for each c', considering c as c'.

Step 2.1. For a change k and a (maximal) set of changes K in the definition of c that follow the rules 1, 2 or 3, select all change instances of k and changes in K from O and the computed graphs of complex change instances $I_{c'}$ that verify the selection, relational and version filters defined only on their parameters. Let I_{kK}^1, I_{kK}^2, ..., I_{kK}^n be the RDF graphs where each one contains the selected change instances that verify all constraints.

Step 2.2. Repeat Step 2.1 for the remaining changes of c (i.e. excluding k and those in K) until any change in c remains or neither rule 1 nor rule 2 or 3 holds.

Step 2.3. If there are still remaining changes of c, then for each change k select all change instances from O or $I_{c'}$ that verify selection and version filters defined only on its parameters. Let I_k be the RDF graph of the selected change instances of type k.

Step 3. Create all possible sets of selected change instances $\{I_1, I_2, \dots, I_n\}$, by taking all possible combinations of the RDF graphs computed for each change k in Step 2.1, and adding in each set the graphs computed in Step 2.3. For each set combine the graphs, do Step 3.1-3.2.

Step 3.1. Select from I_1 and I_2 all the change instances that verify the relational and version filters defined only on the parameters of the respective changes. Delete the initial I_1 and I_2 and let I_{12}^1, I_{12}^2, ..., I_{12}^m be the RDF graphs where each one contains the selected change instances from I_1 and I_2 that verify the constraints.

Step 3.2. Repeat Step 3 for each $\{I_{12}^1, I_3, \dots, I_n\}$, $\{I_{12}^2, I_3, \dots, I_n\}$, ..., $\{I_{12}^m, I_3, \dots, I_n\}$ until all the graphs per set are combined into one graph $I_{12\dots n}^i$.

Step 4. For each combined graph $I_{12\dots n}^i$ create a complex change instance, evaluating its parameters given the bindings in its definition and the change instances it contains which are in $I_{12\dots n}^i$.

Step 5. Return I_c, the graph of the detected complex changes instances of c.

common pattern example is complex changes. If we overlook this remark, the presented algorithms lead in computing the common patterns once for each complex change that comprises them, leading to an overhead in the detection performance.

In order to avoid this, a preprocessing step for identifying the common patterns between complex change definitions is needed. Algorithm 1 or 2 can be used in order to compute the respective graphs of instances for each pattern. Next, each complex change c in C is detected based on similar algorithms being amended so that Step 1 and 2 are performed for the changes in c excluding any common patterns and Step 3 takes into account already constructed graphs for the common patterns.

Next we proceed in the second step, identifying conflicting instances and find possible alternative solutions, in order to evaluate exclusiveness constraints. For demonstrating and resolving conflicting complex change instances we construct a graph named *exclusiveness graph* $X(V, E)$. Every node in V represents a detected complex change instance. Every edge in E connects two nodes representing conflicting instances. A maximal independent set of an exclusiveness graph X is a set of complex change instances where any pair of them is conflicting. Calculating all possible sets of non-conflicting complex change instances is reduced into listing all maximal independent sets of X. This is a known graph problem, equivalent to listing all maximal cliques in the complementary graph X^c.

Algorithm 3 describes how an exclusiveness graph X is constructed given an RDF graph of detected (simple and complex) change instances O. The set of nodes V equals the union of all detected complex change instances in O (line 3). In order to construct the set of edges E the following steps are followed: For each complex change instance c in O that contains exclusively change instances, the ancestors

Algorithm 3. Exclusiveness graph construction

INPUT: RDF graph of detected complex and simple change instances O
OUTPUT: Exclusiveness graph $X(V, E)$ for O
1 $V = \emptyset, E = \emptyset$
2 **for each** complex change instance $c \in O$ **do**
3 $V = V \cup \{c\}$
4 **if** c contains exclusively changes **then**
5 $A = c \cup Find\_Ancestors(c, O)$
6 **for each** change instance n s.t. c contains exclusively n **do**
7 **if** n is a simple change instance **then**
8 $A' = \emptyset$
9 **for each** complex change instance $c' \in O$ s.t. $c' \neq c$, $c' \notin A$, c' contains/contains exclusively n**do**
10 $localA' = Find\_Ancestors(c', O, A')$, $localA' = localA' \backslash A$, $A' = A' \cup c' \cup localA'$
11 **end for**
12 **for each** $a \in A$ **do for each** $a' \in A'$ **do** $E = E \cup (a, a')$ **end for end for**
14 **else**
15 $S = \emptyset, C = \emptyset$
16 $(S, C) = Find\_Descendants(n, O, S, C)$
17 **for each** $s \in S$ **do**
18 $A' = \emptyset$
19 **for each** complex change instance $c' \in O$ s.t. $c' \notin C$, $c' \notin A$, c' contains/contains exclusively s**do**
20 $localA' = Find\_Ancestors(c', O, A')$, $localA' = localA' \backslash A$, $A' = A' \cup c' \cup localA'$
21 **end for**
22 **for each** $a \in A$ **do for each** $a' \in A'$ **do** $E = E \cup (a, a')$ **end for end for**
23 **end for**
24 **end if**
25 **end for**
26 **end if**
27**end for**
28 **return** (V, E)

of c are found using $Find\_Ancestors$ function (lines 2-5). The ancestors are complex change instances that (exclusively) contain c or any parent of them in the hierarchy of changes. They are stored in variable A together with c. Then for each simple change instance n that c contains exclusively, complex change instances c' (different from c and its ancestors) that (exclusively) contain n are found, as well as all its ancestors ($localA'$) (lines 7-10). Only the uncommon ancestors with those of c have to be taken into consideration in $localA'$, as these are the conflicting ones (line 10). A' summarizes all complex change instances c' and all their uncommon ancestors with c (line 10). Therefore, all combinations of elements of A and A' are conflicting, and for these edges have to be added into E (line 12). Respectively, for each complex change instance n that c contains exclusively, all simple (S) and complex change (C) descendant instances are found by $Find\_Descendants$ function (line 16). The descendant instances are those contained by c or by any of its children in the hierarchy of changes. Then each simple change instance in S is handled as in previous (lines 18-20).

Notice that in case there is a hierarchy of complex change instances with at least one exclusive containment declaration between them, some simple changes may be checked multiple times depending on the number of ancestor complex change instances. In order to avoid this, already checked changes may be marked with suitable flags. Also, the implementation of $Find\_Ancestors$ and $Find\_Descendants$ functions is trivial and thus omitted.

6 Conclusions

In this paper we have argued that treating changes as first class citizens is a central issue regarding evolution management. In our view this involves modeling, defining and detecting complex changes. In this way semantic rich changes are employed for understanding evolution and multiple interpretations of evolution can be supported. We proposed our perception regarding complex changes, as well as a declarative language for defining complex changes for RDF(S) knowledge bases. Also, we provided algorithms for detecting possible complex change instances. Future work is directed in implementing the above ideas and evaluating them over real cases in order to demonstrate the language expressiveness as well as the efficiency of detection alternatives.

Acknowledgements: Supported by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thales. Investing in knowledge society through the European Social Fund.

7 References

1. S. Auer and H. Herre. A versioning and evolution framework for RDF knowledge bases. In Perspectives of Systems Informatics, 2007.
2. T. Berners-Lee and D. Connolly. Delta: An ontology for the distribution of differences between RDF graphs. http://www.w3.org/DesignIssues/Diff (version: 2006-05-12), 2004.
3. E. Franconi, T. Meyer, and I. Varzinczak. Semantic diff as the basis for knowledge base versioning. In NMR, 2010.
4. T. Galani, Y. Stavrakas, G. Papastefanatos, G. Flouris. Supporting Complex Changes in RDF(S) Knowledge Bases. In 1st Diachron Workshop, hosted by ESWC 2015.
5. M. Klein. Change management for distributed ontologies. Ph.D. thesis, Vrije University, 2004.
6. J. Malone, E. Holloway, T. Adamusiak, M. Kapushesky, J. Zheng, N. Kolesnikov, A. Zhukova, A. Brazma, and H. Parkinson. Modeling Sample Variables with an Experimental Factor Ontology. Bioinformatics 26(8):1112-1118, 2010.
7. N. F. Noy, and M. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. In AAAI, 2002.
8. G. Papastefanatos, Y. Stavrakas, and T. Galani. Capturing the history and change structure of evolving data. In DBKDA, 2013.
9. V. Papavasileiou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. High-level change detection in RDF(S) KBs. ACM Trans. Database Syst., 38(1), 2013.
10. P. Plessers, O. De Troyer, and S. Casteleyn. Understanding ontology evolution: A change detection approach. J. Web Sem. 5(1): 39-49, 2007.
11. Y. Roussakis, I. Chrysakis, K. Stefanidis, G. Flouris, and Y. Stavrakas. A flexible framework for defining, representing and detecting changes on the data web. CoRR abs/1501.02652, 2015.
12. L. Stojanovic. Methods and tools for ontology evolution. Ph.D. thesis, University of Karlsruhe, 2004.

13. M. Volkel, W. Winkler, Y. Sure, S. Kruk, and M. Synak. SemVersion: A versioning system for RDF and ontologies. In ESWC, 2005.
14. D. Zeginis, Y. Tzitzikas, and V. Christophides. On computing deltas of RDF/S knowledge bases. ACM Transactions on the Web, 2011.