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

Theodora Galani[1, 2], George Papastefanatos[1] and Yannis Stavrakas[1]
[1]*Institute for the Management of Information Systems, RC ATHENA, Artemidos 6 & Epidavrou,Marousi, Greece*
[2]*School of Electrical and Computer Engineering, NTUA, Zografou, Athens, Greece*
theodora@dblab.ece.ntua.gr, {gpapas, yannis}@imis.athena-innovation.gr

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

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. Instead, changes should be treated as first-class-citizens. In our view, this involves supporting semantically rich, user-defined changes that we call complex changes, as well as identifying the interrelations between them. 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.

1 INTRODUCTION

The increasing amount of information published on the web poses new challenges for data management. A central issue concerns evolution management. Data published on the web frequently change, as errors may need to be fixed or new knowledge has to be incorporated. Data consumers need to know what changed among versions, as well as how and why it changed. Thus, the need for maintaining data versions and identifying changes becomes evident.

In this paper we focus on interpreting evolution on RDF(S) knowledge-bases, as RDF is the de-facto standard for representing data on the web. A typical approach for handling changes among dataset versions is computing diffs between them, leading to a machine-readable representation of changes based on triple additions and deletions. This approach does not provide any intuition about change semantics or possible relations between them. An ideal approach would compute human-readable, semantically rich changes along with any interrelations between them.

For example, consider a simplified ontology representing a company's employees, as in Figure 1. Figure 1(a) depicts the initial version, while Figure 1(b) the version after the modifications. Note that classes are in bold font. Each employee is described by her name, salary, position and optionally grade and projects assigned. Employees are organised in a hierarchical structure, depicting position hierarchy, as each one refers to another. In Figure 1(b), modified parts are depicted in light grey. Initially, employee "theo" is leading a small team of programmers, comprising of "mary" and "kate" working on project A. Later, he gets an excellent appraisal turning his grade from 9 to 10. As a result, he gains a salary increase. Also, he gets promoted to a manager. The promotion leads to an additional salary increase and overall the salary doubles. As the business goes well, a new employee has to be hired in order to organize the increasing team responsibilities. As a result, a new team leader is added, "nick", serving as senior employee, guiding "mary" and "kate", and reporting to the manager. From now on, the projects are assigned to him and thus they are moved from "mary" and "kate" to him.

Computing the diff between these two versions totally misses capturing change semantics and dependencies. Understanding the intentions behind data modifications can be even more complicated for large datasets, where changes are numerous and dispersed. Instead, Figure 2 depicts an intuitive and descriptive representation of how data changed. Figure 2(a) depicts the modifications regarding "theo", while Figure 2(b) regarding "nick". Each

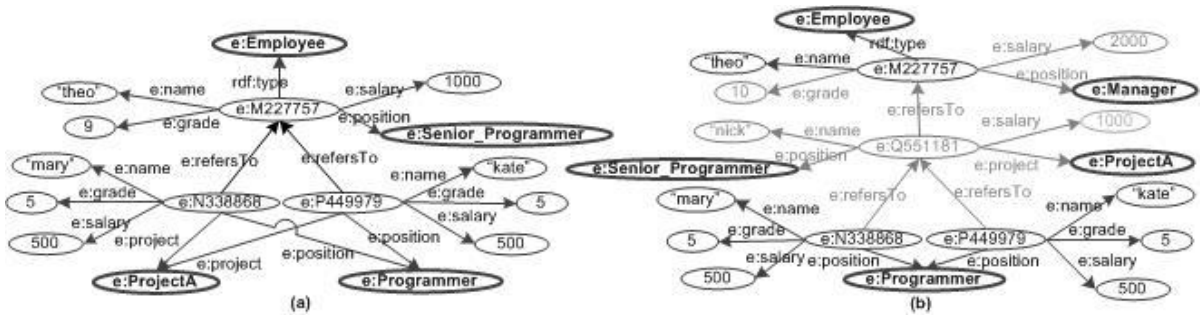

Figure 1: Simplified part of an employee ontology. (a) Initial version. (b) Version after modifications.

node represents a *change instance* detected between the aforementioned dataset versions. Change instances on leaf nodes (in grey) are fine-grained and model-specific, meaning that they do not comprise of other change instances and their semantics suit to the RDF data model. Each one corresponds to an added or deleted triple, having a suitable name and descriptive parameters. They are *simple change instances*. The rest change instances (in white) are coarse-grained and application/data-specific, meaning that they demonstrate structure and semantics suitable to the employee example. The hierarchical structure indicates that a change instance is build on top of others, demonstrating relations and dependencies among changes. They are *complex change instances*.

Consider the change instances Add_Grade and Delete_Grade in Figure 2(a). They serve as specializations of the model specific Add_Property _Instance and Delete_Property_Instance, respectively. This holds for all similar change instances regarding employee properties. Employee_Positive_Appraisal instance *contains* them, modelling the positive evaluation that took place. Similarly, Employee_Promotion_Manager and Employee_Salary_Increase group change instances providing richer semantics on how data changed. Employee_ Salary_Increase is caused by Employee_Positive_Appraisal and Employee_ Promotion_Manager. Causality is modelled on top of these changes through Salary_Increase_after_ Positive_Appraisal and Salary_Increase_after_ Promotion_Manager. These change instances are *overlapping* as they share a common part, Employee_Salary_Increase, modelling that they cause the same effect on data. Similar properties are demonstrated on change instances of Figure 2(b). Add_Employee instance groups all properties related to a newly added employee. Add_ Senior_Employee instance is a specialization of Add_ Employee, where the added employee (with id e:Q551181, i.e. "nick") reports to a manager (with id e:M227757, i.e. "theo") and serves as a leader to other employees (with ids e:N338868 and e:P449979, i.e. "mary" and "kate"). This is modelled by the position he gets in the hierarchy, via Add_Reference instances. Also, Add_Senior_ Employee instance contains a Move_Project_ Assignment instance, as project A is moved from "mary" and "kate" to "nick", and Delete_Reference instances as these employees initially had "theo" as a leader. These changes are secondary and may happen when adding a senior employee.

In this paper, we argue that for understanding data evolution, changes should be treated as first-class-citizens. In our view, this involves supporting human-readable, semantically rich, user-defined changes, named *complex changes*. These changes are application/data-specific and coarse-grained, defined over primitive and model-specific changes, named *simple changes*. Modelling explicitly complex changes provides additional information for interpreting past data, while supporting user-defined changes allows interpreting evolution in multiple ways. On top of this, supporting *interrelated* complex changes, through nesting and overlaps, is an additional feature that enriches the complex changes' expressivity. A complex change may be part of another, may generalize/ specialize another, may cause another or may provide supplementary interpretation of evolution. Section 3 contains the basic concepts of our approach. Given these concepts, we provide a declarative language for defining complex changes (Section 4). We then define a process for detecting complex change instances among dataset versions (Section 5). Both the language and detection algorithm are influenced by our main contribution of supporting interrelated complex changes. Section 2 discusses related work and Section 6 concludes the paper.

2 RELATED WORK

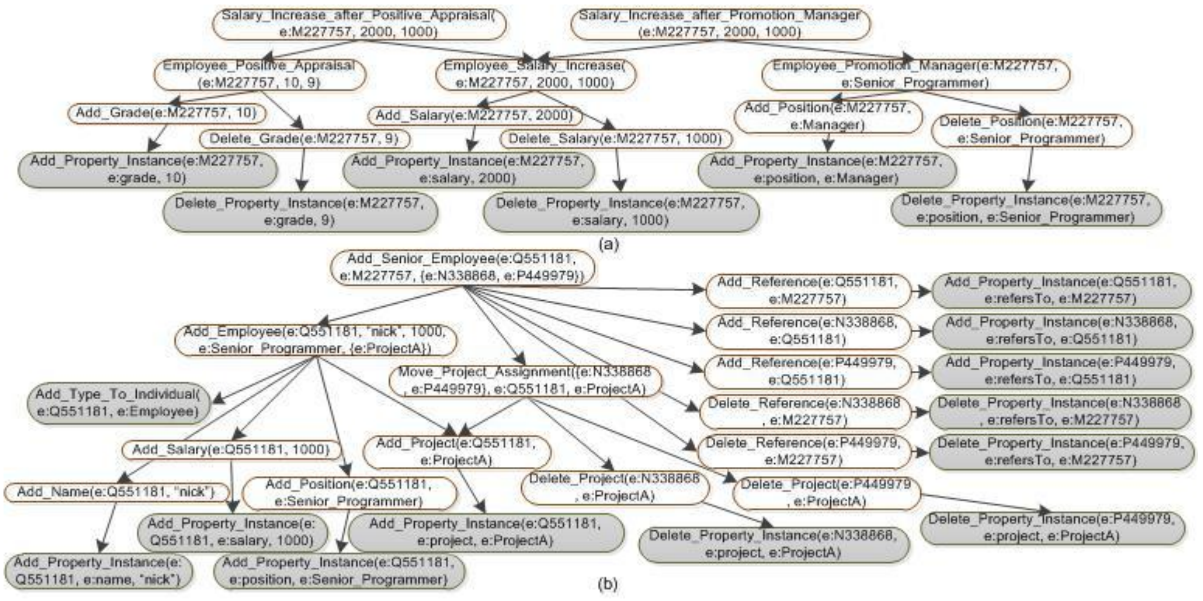
Salary_Increase_after_Positive_Appraisal(e:M227757, 2000, 1000)
Salary_Increase_after_Promotion_Manager(e:M227757, 2000, 1000)
Employee_Positive_Appraisal(e:M227757, 10, 9)
Employee_Salary_Increase(e:M227757, 2000, 1000)
Employee_Promotion_Manager(e:M227757, e:Senior_Programmer)
Add_Grade(e:M227757, 10)
Delete_Grade(e:M227757, 9)
Add_Salary(e:M227757, 2000)
Delete_Salary(e:M227757, 1000)
Add_Position(e:M227757, e:Manager)
Delete_Position(e:M227757, e:Senior_Programmer)
Add_Property_Instance(e:M227757, e:grade, 10)
Add_Property_Instance(e:M227757, e:salary, 2000)
Add_Property_Instance(e:M227757, e:position, e:Manager)
Delete_Property_Instance(e:M227757, e:grade, 9)
Delete_Property_Instance(e:M227757, e:salary, 1000)
Delete_Property_Instance(e:M227757, e:position, e:Senior_Programmer)
(a)
Add_Senior_Employee(e:Q551181, e:M227757, {e:N338868, e:P449979})
Add_Reference(e:Q551181, e:M227757)
Add_Property_Instance(e:Q551181, e:refersTo, e:M227757)
Add_Reference(e:N338868, e:Q551181)
Add_Property_Instance(e:N338868, e:refersTo, e:Q551181)
Add_Reference(e:P449979, e:Q551181)
Add_Property_Instance(e:P449979, e:refersTo, e:Q551181)
Delete_Reference(e:N338868, e:M227757)
Delete_Property_Instance(e:N338868, e:refersTo, e:M227757)
Delete_Reference(e:P449979, e:M227757)
Delete_Property_Instance(e:P449979, e:refersTo, e:M227757)
Add_Employee(e:Q551181, "nick", 1000, e:Senior_Programmer, {e:ProjectA})
Move_Project_Assignment({e:N338868, e:P449979}, e:Q551181, e:ProjectA)
Add_Type_To_Individual(e:Q551181, e:Employee)
Add_Salary(e:Q551181, 1000)
Add_Project(e:Q551181, e:ProjectA)
Delete_Project(e:N338868, e:ProjectA)
Delete_Project(e:P449979, e:ProjectA)
Add_Name(e:Q551181, "nick")
Add_Position(e:Q551181, e:Senior_Programmer)
Add_Property_Instance(e:Q551181, e:salary, 1000)
Add_Property_Instance(e:Q551181, e:project, e:ProjectA)
Delete_Property_Instance(e:N338868, e:project, e:ProjectA)
Delete_Property_Instance(e:P449979, e:project, e:ProjectA)
Add_Property_Instance(e:Q551181, e:name, "nick")
Add_Property_Instance(e:Q551181, e:position, e:Senior_Programmer)
(b)

Figure 2: Hierarchy of detected simple and complex change instances (in grey and white fill respectively) of the employee ontology of Fig. 1. (a) Change instances regarding "theo". (b) Change instances regarding "nick".

A number of works focus on computing the differences between knowledge bases. In (Berners-Lee and Connolly, 2004) an ontology for representing differences between RDF graphs, in the form of insertions and deletions, is proposed. RDF graphs comparison is discussed, as well as updating a graph from a set of differences. In (Volkel et al., 2005) two diff algorithms are proposed: one computing a structural diff (as the set-based difference of the triples explicitly recorded into the two graphs), and one semantic diff (taking into consideration the semantically inferred triples). In (Franconi et al., 2010), an approach for computing a semantic diff is proposed, focusing on propositional logic knowledge bases but also being applicable to more expressive logics. A number of desired properties are discussed, like semantic diff uniqueness, the principal of minimal change, the ability to undo changes and version reconstruction. Similar properties are supported in (Zeginis et al. 2011), which focuses on computing deltas over RDF(S) knowledge bases. In (Noy and Musen, 2002; Klein, 2004), a fixed point algorithm for detecting ontology change is proposed. It employs heuristic-based matchers, introducing uncertainty to results.

Other works focus on supporting human-readable changes. In (Papavasileiou et al., 2013), a set of predefined high-level changes for RDF(S) knowledge bases and an algorithm for their detection are proposed. Changes verify the 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 (Roussakis et al., 2015), an extension of (Papavasileiou et al., 2013) is proposed, providing a more generic change definition framework, based on SPARQL queries. In (Plessers, De Troyer and Casteleyn, 2007), Change Definition Language is proposed for defining and detecting changes over a version log using temporal queries. In (Auer and Herre, 2007) a framework for supporting evolution in RDF knowledge bases is discussed. Changes are triple additions and deletions and aggregated triples, resulting in a hierarchy of changes. However, neither a detection process, nor a specific language of changes is defined. In (Klein, 2004), an extension of (Noy and Musen, 2002) is proposed for detecting some of the proposed basic and composite changes. In general, (Klein, 2004), (Papavasileiou et al., 2013) and (Stojanovic, 2004) provide human readable changes in similar categories regarding granularity and semantics.

Our approach focuses on human readable changes. A visionary work was presented in (Galani et al., 2015). Similar to (Klein, 2004), (Papavasileiou et al., 2013), (Roussakis et al., 2015) and (Stojanovic, 2004) we assume primitive changes, as simple changes, and groupings of them, as complex changes. Instead of providing a predefined list of complex changes, we support user-defined complex changes in order to capture richer semantics and multiple interpretations of evolution, as (Plessers, De Troyer and Casteleyn, 2007) and (Roussakis et al., 2015). Our main contribution is supporting interrelated complex changes providing a language for defining complex changes over simple or other complex changes and an appropriate detection algorithm. (Plessers, De Troyer and Casteleyn, 2007) and (Roussakis et al., 2015) do not support interrelated complex changes.

3 SIMPLE AND COMPLEX CHANGES

Modelling changes as first class citizens involves taking into account granularity and semantics of changes. Granularity 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 poses the question of having model-specific or application/data-specific changes. Model-specific changes describe modifications that appear in a specific model, constituting a fixed set of generic changes. Application/data-specific changes suit specific use-cases and may be user-defined, allowing multiple interpretations of evolution.

As a result, 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/data-specific changes providing richer semantics on how data changed. Definitions 1 and 2 formally define simple and complex changes.

Definition 1: A simple change s is a tuple (n, p), where:

- n is the name of s, which must be unique.
- p is the list of descriptive parameters of s, where each one has a unique name within s.

Definition 2: A complex change c is a quadruple (n, p, D, F), where:

- n is the name of c, which must be unique and different from the simple change names.
- p is the list of descriptive parameters of c, where each one has a unique name within c.
- D is the set of simple (D_S) and complex changes (D_C) that c comprises of, where $D = D_C \cup D_S$, $D_C \cap D_S = \emptyset$ and $D \neq \emptyset$.
- F is the set of constraints (F_C) that changes in D verify and bindings (F_B) specifying the parameters in p, where $F = F_C \cup F_B$ and $F_C \cap F_B = \emptyset$. Constraints are on changes (F_C^{car}) or change parameters (F_C^{par}), where $F_C = F_C^{car} \cup F_C^{par}$ and $F_C^{car} \cap F_C^{par} = \emptyset$.

For simple changes we rely on (Papavasileiou et al., 2013). Appendix summarizes the simple changes considered. They verify completeness and unambiguity properties, constituting a first layer of human-readable changes. Simple changes are additions, deletions and terminological changes (rename, split, merge) of RDF(S) entities (classes, properties, individuals). As stated, simple changes are fine-grained, i.e. they cannot be decomposed in more granular changes. This holds for additions/ deletions, but not for terminological changes, as they can be expressed as additions/ deletions plus extra conditions. For example, a class rename can be considered as an add class plus a delete class, which have the same "neighbourhood" (properties, connections to classes). However, we prefer them as simple changes in order to distinguish at simple change level real additions/deletions from virtual ones representing terminological changes. Thus, simple changes' set is not minimal.

A complex change is defined in terms of simple or other complex changes verifying constraints. Constraints specialize its meaning and are divided into those defined on changes and those on change parameters. Bindings specify complex change parameter values. Section 4 includes more details.

The ultimate goal of supporting simple and complex changes is detecting actual instances between dataset versions. Detection process leads into instantiating change parameters with values, indicating that specific data elements have been affected by a change in a specific manner. Definitions 3 and 4 define simple and complex change instances. Figure 2 presents simple and complex change instance examples.

Definition 3: A simple change instance of a simple change (n, p), is a tuple (n, v) where v is an instantiation of the parameters p.

Definition 4: A complex change instance of a complex change (n, p, D, F), is a tuple (n, v) where v is an instantiation of the parameters p.

For simple change detection we rely on (Papavasileiou et al., 2013). For complex changes we provide an algorithm in Section 5. Definition 5 defines when a complex change instance is detected. Definitions 6 and 7 define possible relations among change instances, as interrelations between changes are reflected on them.

Definition 5: Let $c = (n, p, D, F)$ be a complex change and V_b and V_a two dataset versions. A complex change instance $c_i = (n, v)$ is detected if for all changes in D instances are detected between V_b and V_a, forming D_i, such that constraints in F_C are verified on D_i, V_b and V_a, bindings in F_B applied on D_i form v and D_i is maximal.

We say that the set of change instances D_i corresponding to c_i verifies the complex change c.

Definition 6: Let c_i be an instance of complex change c and D_i the corresponding set of change instances verifying c. c_i contains the change instances in D_i.

Definition 7: Let c_i and c_i' be two complex change instances, where c_i does not contain c_i' and vice versa. They are overlapping if they both contain at least one common simple or complex change instance.

Containment is transitive. Complex change instances may form a hierarchy due to containment and overlaps, as in Figure 2.

4 A LANGUAGE FOR DEFINING COMPLEX CHANGES

We believe that an intuitive, user-friendly language based on change semantics should be provided for defining complex changes. Complex change definitions are then used for detecting respective instances. In this section, we propose a declarative language for defining complex changes. We provide its syntax by means of EBNF specification (Table 1) and some illustrative examples (Table 2) concerning the employee ontology in Figure 1.

A complex change definition is composed by a heading and a body. The heading contains a unique name and a list of descriptive parameters. The body contains a list of changes that the complex change comprises of, constraints on the changes appearing in the list and their parameters, and parameter bindings declaring how complex change parameters are evaluated. Constraints and bindings are optional. A complex change definition is nested if complex changes appear in its change list. Thus, complex changes are defined as interrelated. Constraints are divided into cardinality, testing value, relational, pre/post-conditions and functions.

Cardinality constraint determines whether zero, one or multiple instances of a specific change are to be grouped into a complex change. In case of one or multiple change instances, the change is defined as mandatory. In case of zero instances the change is defined as optional, and if no instance is detected, the respective complex change can be still detected. Thus, complex changes are flexible and tolerant in partially performed modifications of minor significance. Posing a cardinality constraint is optional. If it is not defined, the default case is one change instance for the respective change. The following notations hold: at least one change instance "+", zero or one "?", zero or more "*".

Parameter bindings determine how complex change parameters are evaluated. In general, a complex change parameter equals a parameter of a change in its change list. However, recall that due to cardinality constraints multiple change instances of a specific change type may be grouped. In such case, a complex change parameter equals the union of the parameter values for all the change instances of a specific type grouped. As a result, complex change parameters are distinguished into those that evaluate into type set and those that evaluate into scalar values. In order to distinguish the parameter types, parameters evaluating into scalar values start with a lowercase letter, while those evaluating into sets with an uppercase letter. Parameter bindings are optional, in case they can be inferred by repeating each parameter into the contained changes and respective constraints.

Testing value constraints, relational constraints, pre/post-conditions and functions are constraints defined on change parameters. Testing value constraints limit a parameter value against a given constant, while relational constraints involve two change parameters defining how changes are connected. For these constraints binary operators are supported. Pre/post-conditions define how parameters are related in the version before (V_b) or after (V_a) the change, stating whether a triple must or must not exist in the version before or after. If a triple may be inferred in a version, this is denoted by the flag "inferred". Constraints may also be in the form of predefined functions of return type boolean. For example consider common functions on strings, like contains, which checks whether a string contains another given string. Constraints may form composite conditions, when combined in boolean expressions using logical and, or, not.

As complex changes are used in nested definitions and complex change parameters may evaluate into set or scalar values, we support binary operators between sets and between sets and scalar values. Also, in order to write conditions on set elements we use quantified expressions, which may be in the form $\{\forall, \exists, \nexists\}\, x \in X{:}\, f(x)$ or $\{\forall, \exists, \nexists\}\, x \in X{:}\, \{\forall, \exists, \nexists\}\, y \in Y{:}\, f(x,y)$, where $f(x)$ and $f(x,y)$ are constraints on parameters evaluating into scalar values.

Table 1: The EBNF specification of the complex change definition language.

```
complex-change-definition = 'CREATE COMPLEX CHANGE' heading '{' body '}'';' ;
heading = name '(' parameter-list ')' ;
parameter-list = identifier {', ' identifier} ;
body = change-list ['; ' filter-list] ['; ' binding-list] ;
name = STRING ;
identifier = LETTER {LETTER|DIGIT} ;
change-list = 'CHANGE LIST' change {', ' change} ;
change = change-heading [cardinality] ;
change-heading = change-name '(' parameter-list ')' ;
change-name = name | NAMES OF SUPPORTED SIMPLE CHANGES ;
cardinality = '+'|'?'|'*' ;
filter-list = 'FILTER LIST' or-filter-expr {', ' or-filter-expr} ;
or-filter-expr = and-filter-expr {'||' and-filter-expr} ;
and-filter-expr = neg-filter-expr {'&&' neg-filter-expr} ;
neg-filter-expr = ['!'] filter-expr ;
filter-expr = bracketed-expr | expr ;
bracketed-expr = '(' or-filter-expr ')' ;
expr = [quantification] constraint ;
quantification  =  'for'  ('each'|'some'|'none')  identifier  'in'  identifier  ':'  ['for'
('each'|'some'|'none') identifier 'in' identifier ':'] ;
constraint = test-val-constr | rel-constr | pre-post-constr | fun-constr ;
test-val-constr = identifier bin-op constant ;
rel-constr = identifier bin-op identifier ;
pre-post-constr = '(' (identifier | value) ', ' (identifier | value) ', ' (identifier |
value) ')' ['inferred'] ('in' | 'not in') ('Vb' | 'Va') ;
fun-constr = name '(' parameter-constant-list ')' ;
bin-op = '=' | '!=' | '>' | '<' | '>=' | '<=' | 'subSet' | 'properSubset' | 'superSet' |
'properSuperset' | 'in' | 'not in' ;
parameter-constant-list = (identifier | constant) {', ' (identifier | constant)} ;
constant = set | value ;
set = '{' value-list '}' ;
value-list = value {', ' value} ;
value = URI | LITERAL ;
binding-list = 'BINDING LIST' binding {', ' binding} ;
binding = binding-equality | binding-union ;
binding-equality = identifier '=' identifier ;
binding-union = identifier '<-' identifier ;
```

Table 2: Complex change definitions regarding the employee ontology of Figure 1.

`CREATE COMPLEX CHANGE Add_Grade(x, g) { CHANGE LIST Add_Property_Instance(x, prop, g);   FILTER LIST prop="e:grade";   };`
`CREATE COMPLEX CHANGE Employee_Positive_Appraisal(x, ng, og) { CHANGE LIST Add_Grade(x, ng), Delete_Grade(x, og);   FILTER LIST ng>og;   };`
`CREATE COMPLEX CHANGE Employee_Promotion_Manager(x, op) { CHANGE LIST Add_Position(x, np), Delete_Position(x, op);   FILTER LIST np=e:Manager;   };`
`CREATE COMPLEX CHANGE Employee_Salary_Increase(x, ns, os) { CHANGE LIST Add_Salary(x, ns), Delete_Salary(x, os);   FILTER LIST ns>os;   };`
`CREATE COMPLEX CHANGE Salary_Increase_after_Positive_Appraisal(x, ns, os) { CHANGE LIST Employee_Salary_Increase(x, ns, os), Employee_Positive_Appraisal(x, ng, og); };`
`CREATE COMPLEX CHANGE Salary_Increase_after_Promotion_Manager(x, ns, os) { CHANGE LIST Employee_Salary_Increase(x, ns, os), Employee_Promotion_Manager(x, op); };`
`CREATE COMPLEX CHANGE Move_Project_Assignment(S, c, val) { CHANGE LIST Add_Project(c, val), Delete_Project(s, val) +;   BINDING LIST S←s;   };`
`CREATE COMPLEX CHANGE Add_Employee(x, name, salary, position, grade, Project) { CHANGE LIST Add_Type_To_Individual(x, t), Add_Name(x, name), Add_Salary(x, salary), Add_Position(x, position), Add_Grade(x, grade)?, Add_Project(x, project)*; FILTER LIST t=e:Employee;   BINDING LIST Project←project;   };`
`CREATE COMPLEX CHANGE Add_Senior_Employee(seniorx, m, X) { CHANGE LIST Add_Employee(seniorx, name, salary, position, grade, Project), Add_Reference(seniorx, m), Add_Reference(x, seniorx)+, Delete_Reference(x, psx)*, Move_Project_Assignment(S, seniorx, val)*;   FILTER LIST for each s in S: (s,e:refersTo,seniorx) in Va, (m,e:position,e:Manager) in Va;   BINDING LIST X←x;   };`

Table 2 contains complex change definitions regarding the changes of the employee ontology in Figure 1 discussed in introduction. Add_Grade models the case where a new grade property with value g is assigned to employee x. The changes it comprises of are declared in the "change list", while constraints in the "filter list". Add_Grade is a specialization of simple change Add_Property_ Instance, where the property equals to "e:grade". This is a testing value constraint over parameter prop. Notice that no binding is defined explicitly, as they are inferred by repeating complex change parameters as parameters of the changes in change list. Besides Add_Property_Instance no cardinality constraint is defined, meaning that cardinality one is inferred. Similar complex change definitions for all employees' properties can be given, but are omitted due to space limitations. Employee_ Positive_Appraisal models the case when an employee x gets a new grade, ng, greater than the old one, og. It comprises of Add_Grade, so that the new grade is assigned to the employee, and Delete _Grade, so that the old grade is removed, both referring to the same employee x. A relational filter compares the new and the old grade. Empployee_Salary_Increase is similarly defined. Employee_Promotion_Manager models the case when an employee x becomes a manager. Add_ Position assigns the new position to x and Delete_ Position deletes the old one. A testing value constraint specifies the new position as e:Manager.

The complex change Salary_Increase_after_ Positive_Appraisal comprises of Employee_Salary _Increase and Employee_Positive_Appraisal, modelling the case when a salary increase of employee x is caused after receiving positive appraisal. Thus, complex changes are grouped due to a causality relation. A similar concept holds for Salary_Increase_after_Promotion_Manager. These changes both base on Employee_Salary_Increase, as they try to explain why this increase has been caused. Thus, respective instances may overlap, if they both refer to the same employee, like "theo" in Figure 1 and 2. Due to nested definitions the respective instances lead to a hierarchical structure.

Move_Project_Assignment models the case where a project val, initially assigned to a set of employees S, is later assigned to another employee c. It comprises of Add_Project, as the project is assigned to c, and Delete_Project, as the project is deleted from another employee s. Both changes refer to the same project, as val is repeated in both. Besides Delete_Project "+" is noted. This is a cardinality constraint defining that there might be multiple deletions. The project may be initially assigned to multiple employees and then deleted from many of them. In such case, all these Delete_Project instances will be grouped into the respective complex change instance (through detection process). Now, consider that similarly the project can be moved to multiple employees too. This would cause multiple Add_Project instances. But, on Add_Project it is assumed cardinality one. Therefore, only one instance will be grouped in every complex change instance and multiple complex change instances will be detected, one for each Add_Project instance. As a result, supporting cardinality is important in order to define how change instances are grouped. We choose to follow cardinality as in Table 2 in order to construct groupings per project and per employee it has been moved to. Due to cardinality constraint, parameter S holds all employee' ids that the project has been removed from, as defined in the binding list.

Add_Employee models the case where a new employee is added with a number of descriptive properties. x is of type e:Employee, as defined in the testing value constraint. Property grade is optionally added, as defined by "?" besides Add_ Grade. Add_Project is optional too, but if it is added there might be many instances ("*"). Add_ Senior_Employee is a specialization of Add_ Employee and thus it is defined on top of it. It models the case when a newly added employee refers to a manager and leads other employees. This is described by e:refersTo property, through Add_Reference changes. The fact that the added employee refers to a manager is defined by the second post-condition. Also, it is likely that projects are moved to the added employee from the employees he leads. This is demonstrated by Move_Project_Assignment and the first post-condition. A quantified expression is used in order to write the post-condition on the elements of set S.

5 COMPLEX CHANGE DETECTION

Complex change detection is the process of identifying complex change instances. It requires as input a set of simple change instances detected between two dataset versions (S_i), the actual dataset versions (before V_b and after V_a) and the complex change definitions that will be evaluated for detecting respective instances (C). We focus on how nested complex change definitions are handled and how constraints are evaluated. In order to implement the language, we translate it into an already implemented language. As this approach concerns RDF data, we choose to rely on SPARQL, which provides similar capabilities to our language. The presented Algorithm involves two steps: the first step handles nested definitions, the second produces complex change instances.

As for the first step, suppose a complex change c whose definition is based on a set of complex changes ($D_C \neq \emptyset$). The detection of c instances depends on detecting the instances of each complex change in D_C and therefore follows their detection. Note that mutually dependent complex changes are not supported. In general, complex change definitions constitute a directed acyclic graph, where nodes represent changes and edges dependencies between them. An edge departing from a complex change c arrives at changes in D_C according to its definition. Thus, detection follows a post-order depth-first scheme on the induced dependency graph by complex change definitions. This is stated in line 2 of proposed Algorithm. postOrderDfs function call runs over the set of complex changes C identifying the dependencies among changes, returning a queue Q of all changes in C, where the order of elements defines the order in which they have to be detected.

As for the second step, for each complex change c in Q, instances are computed (lines 3-10). The main idea is that our language is translated into SPARQL queries. Accordingly, simple and complex change instances and dataset versions are encoded as RDF data, so that constructed SPARQL queries are applied on them. Therefore, for each complex change an appropriate SPARQL query is created through createQuery function call (line 5). For this, changes in $D(c)$, constraints on their parameters and bindings are employed. Bindings indicate how to select complex change parameter values. Cardinality is taken into account to identify whether a change is optional. This query is executed on the detected change instances and dataset versions (line 6) in order to select change instances that verify the defined constraints. The query results are further elaborated, through createInstances function call (line 7), so that selected changes are grouped based on cardinality. Computed instances are added into the set of instances to be reported I (line 8, initialized in line 1), and are combined with simple change instances in order to be available for detecting depending complex change instances (line 9). Finally, the algorithm returns the set of detected complex change instances I (line 11).

Regarding query generation, testing value and relational constraints map to SPARQL filter expressions or nested queries with aggregation (in case they involve parameters evaluating into sets), while pre/post-conditions map to filter exists/not exists expressions over appropriate graphs holding the version before or after the change. Quantified expressions are also mapped to appropriate nested queries. Cardinality "?" and "*" map to optional declaration, indicating that respective changes may not be present. Bindings guide how query variables in select clause, representing complex change parameters, match query variables in where clause.

```
Algorithm: Complex Change Detection
Input: A set of complex changes $C$, a dataset version before $V_b$ and after $V_a$, a set of simple change instances $S_i$
Output: A set of complex changes instances $I$ of $C$
1  $I \leftarrow \{\}$ ;
2  queue $Q \leftarrow postOrderDfs(C)$ ; //complex changes are sorted following dependencies
3  while $!Q.isEmpty()$ do
4    $c \leftarrow Q.dequeue()$ ;
5    $query \leftarrow createQuery(D(c), F(c))$ ;
6    $resultSet \leftarrow exec(query, S_i, V_b, V_a)$ ;
7    $I_c \leftarrow createInstances(resultSet, F_C^{car}(c))$ ;
8    $I \leftarrow I \cup I_c$ ; //report instances
9    $S_i \leftarrow S_i \cup I_c$ ; // instances are available for detecting depending changes
10 end while
11 return $I$ ;
```

Regarding instance generation, the query results have to be iterated so that they are grouped appropriately given cardinality constraints for constructing complex change instances.

For example consider the following query, which corresponds to Add_Senior_Employee defined in Table 2. In the select clause we consider query variables corresponding to contained changes' identifiers (?c1, ?c2, ?c3, ?c4 and ?c5) and the values which will be assigned to the complex change instance parameters (?sx, ?m and ?x). In the where clause we consider the changes defined in change list and the constraints defined in filter list. For Delete_Reference and Move_Project _Assignment we use optional parts, due to "*" cardinality constraint. For post-conditions we use appropriate SPARQL filter expressions evaluating over the graph holding V_a. The first post-condition refers to Move_Project_Assignment and thus it is placed into the respective optional part. Also, it involves quantification, which is implemented through a nested query. The query results should be iterated for creating instances. Notice that Add_Employee and the first Add_Reference have cardinality equal to one. Thus, all rows having the same value in the respective query variables (?c1, ?c2) will form one complex change instance.

```
   SELECT ?c1 ?sx ?c2 ?m ?c3 ?x ?c4 ?c5
   WHERE { ?c1 rdf:type ch:Add_Employee;
ch:aep1 ?sx.
   ?c2 rdf:type ch:Add_Reference; ch:ar1
?sx; ch:ar2 ?m.
   FILTER EXISTS {GRAPH <http://employeeVa>
{?m e:position e:Manager.}}
   ?c3 rdf:type ch:Add_Reference; ch:ar1
?x; ch:ar2 ?sx.
   OPTIONAL {?c4 rdf:type ch:Delete_
Reference; ch:dr1 ?x; ch:dr2 ?psx.}
   OPTIONAL {?c5 rdf:type ch:Move_Project_
Assignment; ch:mpap1 ?s; ch:mpap2 ?sx;
ch:mpap3 ?v.{SELECT ?c5 WHERE{?c5 rdf:type
ch:Move_Project_Assignment; ch:mpap1 ?s;
ch:mpap2 ?sx. FILTER NOT EXISTS {GRAPH
<http://employeeVa> {?s e:refersTo
?sx.}}}GROUP BY ?c5 HAVING(count(?s)=0)}}}
```

6 CONCLUSIONS

In this paper we argued that treating changes as first class citizens is a central issue in evolution management. This involves modelling, defining and detecting complex changes. Thus semantically rich changes and their interrelations are supported for interpreting evolution in multiple ways. We proposed our perception regarding complex changes, a declarative language for defining them on RDF(S) knowledge bases and a process for detecting complex change instances. Future work is directed in evaluating our approach in terms of language expressiveness and detection efficiency.

ACKNOWLEDGEMENTS

Supported by the EU-funded ICT project "DIACHRON" (agreement no 601043).

REFERENCES

Auer, S., H. Herre, 2007. A versioning and evolution framework for RDF knowledge bases. In Perspectives of Systems Informatics.

Berners-Lee, T., Connolly, D., 2004. Delta: An ontology for the distribution of differences between RDF graphs.http://www.w3.org/DesignIssues/Diff (version: 2006-05-12).

Franconi, E., Meyer, T., Varzinczak. I., 2010. Semantic diff as the basis for knowledge base versioning. In NMR.

Galani, T., Stavrakas, Y., Papastefanatos, G., Flouris, G., 2015. Supporting Complex Changes in RDF(S) Knowledge Bases. In MEPDaW-15.

Klein, M., 2004. Change management for distributed ontologies. Ph.D. thesis, Vrije University.

Noy, N.F., Musen, M., 2002. PromptDiff: A fixed-point algorithm for comparing ontology versions. In AAAI.

Papastefanatos, G., Stavrakas, Y., Galani, T., 2013. Capturing the history and change structure of evolving data. In DBKDA.

Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D., Christophides, V., 2013. High-level change detection in RDF(S) KBs. In ACM Trans. Database Syst., 38(1).

Plessers, P., De Troyer, O., Casteleyn, S., 2007. Understanding ontology evolution: A change detection approach. In J. Web Sem. 5(1): 39-49.

Roussakis, Y., Chrysakis, I., Stefanidis, K., Flouris, G., Stavrakas, Y., 2015. A flexible framework for understanding the dynamics of evolving RDF datasets. In ISWC.

Stojanovic, L., 2004. Methods and tools for ontology evolution. Ph.D. thesis, University of Karlsruhe.

Volkel, M., Winkler, W., Sure, Y., Kruk, S., Synak, M., 2005. SemVersion: A versioning system for RDF and ontologies. In ESWC.

Zeginis, D., Tzitzikas, Y., Christophides, V., 2011. On computing deltas of RDF/S knowledge bases. In ACM Transactions on the Web.

APPENDIX

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 in A into b, 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 in B, 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 A into b, b∈A. **Split_Property(a,B)**: Split property a into properties contained in B. **Split_Property_ Into_Existing(a,B)**: Split a into properties in B, a∈B. **Add_ Type_Individual(a)**: Add object a of type rdfs:resource. **Delete_Type_Individual(a)**: Delete object a of type rdfs: resource. **Merge_Individuals(A,b)**: Merge individuals contained in A into b. **Merge_Individuals_Into_Existing (A,b)**: Merge A into b, b∈A. **Split_Individual(a,B)**: Split individual a into individuals in B. **Split_Individual_Into_ Existing(a,B)**: Split a into individuals in B, 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 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.