Formalizing High-Level Change Detection for RDF/S KBs

V. Papavassiliou[1,2], G. Flouris[1], I. Fundulaki[1], D. Kotzinos[1,3], and V. Christophides[1,2]
email: {papavas, fgeo, fundul, kotzino, christop}@ics.forth.gr

[1] FORTH-ICS, Greece
[2] University of Crete, Greece
[3] TEI of Serres, Greece

Abstract. An increasing number of scientific communities rely on Semantic Web ontologies to share and interpret data within and across research domains. These common knowledge representation resources are usually developed and maintained manually and essentially co-evolve along with experimental evidence produced by scientists worldwide. Detecting automatically the differences between (two) versions of the same ontology in order to store or visualize their deltas is a challenging task for e-science. In this paper, we focus on languages allowing the formulation of concise and intuitive deltas, which are expressive enough to describe unambiguously any possible change and that can be effectively and efficiently detected. We propose a specific language that provably exhibits those characteristics and provide a change detection algorithm which is sound and complete with respect to the proposed language. Finally, we provide a promising experimental evaluation of our framework using real ontologies from the cultural and bioinformatics domains.

1 Introduction

An increasing number of scientific communities rely on Semantic Web ontologies to share and interpret data within and across research domains (e.g., Bioinformatics, Medicine and Health, Environmental and Earth Sciences, Astronomy, Cultural Informatics and Humanities[1]). These community ontologies are usually developed and maintained manually while essentially co-evolve along with experimental evidence produced by scientists worldwide. Managing the differences (*deltas*) of ontology versions has been proved to be an effective and efficient method in order to synchronize them [5] or to explain the evolution history of a given ontology [13]. In this paper, we are interested in automatically detecting *both schema and data changes* occurring between asynchronously produced ontology versions.

Unless they are assisted by collaborative ontology development tools [8,9], ontology editors are rarely able or willing to systematically record the changes performed to obtain an ontology version. In particular, when there is no central authority responsible for ontology curation, manually created deltas are often absent, incomplete, or even

[1] www.geneontology.org, www.biopax.org, www.nlm.nih.gov/research/umls, www.co-ode.org/galen,sweet.jpl.nasa.gov/ontology, archive.astro.umd.edu/ont, cidoc.ics.forth.gr.

erroneous [22]. Existing ontology diff tools, such as PromptDiff [14], SemVersion [23] and others [24] aim to satisfy this need. These tools are essentially based on a *language of changes*, which describes the semantics of the different change operations that the underlying algorithm understands and detects.

In its simplest form, a language of changes consists of only two *low-level* operations, *Add(x)* and *Delete(x)*, which determine individual constructs that were added or deleted [23,24]. In [10,14,15,17,19,22], *high-level* change operations are employed, which describe more complex updates, as for instance the insertion of an entire subsumption hierarchy. A high-level language is preferable than a low-level one for explaining and managing ontology evolution, as it is more *intuitive*, *concise*, *closer to the intentions* of the ontology editors and *captures more accurately* the semantics of a change [10,21].

However, detecting high-level change operations introduces a number of issues. As the detectable changes get more complicated, so does the detection algorithm; complicated changes involve complicated detection procedures which may be inefficient, based on matchers [6] or other heuristic-based techniques [10] that make it difficult to provide any formal guarantees on the detection properties. Another problem stems from the fact that it is impossible to define a complete list of high-level changes [10], so there is no agreed "standard" set of operations that one could be based on. Moreover, it is difficult to specify a language of changes that will be both high-level and able to handle all types of modifications (even fine-grained ones) upon an ontology.

The main contributions of our work are:

- the introduction of a *framework* for defining changes and of a *formal language of changes* for RDF/S ontologies [2,12] which considers operations in both data and schema and satisfies several desirable properties;
- the design of an *efficient* change detection algorithm which is *sound* and *complete* with respect to the proposed language;
- the experimental evaluation of our framework using *real ontologies* from the cultural (CIDOC [4]) and biological domains (GO [7]).

The paper is organized as follows: Section 2 presents a motivating example that will be used for visualization purposes throughout the paper. In Section 3, we introduce the basic notions of RDF [12] and RDFS [2] as well as our language of high-level changes and show that the language and the proposed detection algorithm satisfy several desirable properties. Section 4 describes changes which require heuristics and matchers in order to be detected, thus extending our basic framework to include operations that are interesting in practice. Section 5 presents our experimental findings on real ontologies and section 6 discusses related work. We conclude in Section 7. Finally, in the Appendix a full list of the defined basic and composite changes can be found.

2 Motivating Example

In Figure 1 an example inspired from the CIDOC Conceptual Reference Model [4] is depicted, which is a core ontology intended to facilitate the integration, mediation and interchange of heterogeneous cultural heritage information.

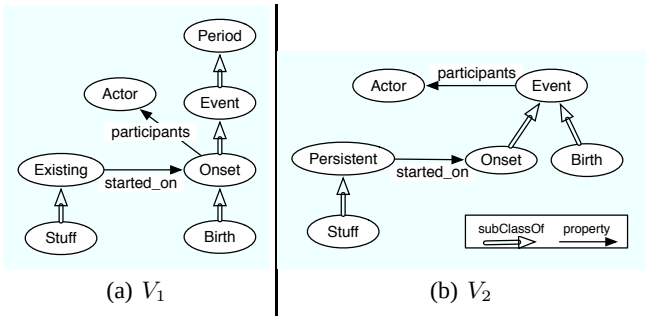


Fig. 1. Motivating Example

Table 1 shows the added and deleted triples (the low-level delta) as well as the high-level change operations that our approach will detect. The table makes clear that even though the low-level delta contains all the changes that have been performed, it is not really useful as it captures the syntactical manipulations that led to the change, rather than the intentions of the editor. Our work captures the fact that by "aggregating" several low-level changes into more *coarse-grained*, *concise* and *intuitive* high-level changes, one can produce more useful deltas (see the third column of Table 1).

For instance, consider the change in the domain of property $participants$ from class $Onset$ to $Event$ (Figure 1). The low-level delta reports two "changes", namely the deletion and the insertion of a domain for the property whereas the reported high-level operation *Generalize_Domain* combines them into one, capturing also the fact that the old domain is a *subclass of* the new domain (by exploiting semantical information in the two versions). A similar case appears in the change of the position of $Birth$ in the subsumption hierarchy which our framework reports as *Pull_Up_Class* and in the deletion of class $Period$ where the deletion of all edges originating from, or ending in, the deleted class are combined in a single operation. Regarding the latter, only a subclass relation is deleted ($Event$), whereas other relations, such as superclasses, supertypes, subtypes, comments and labels are absent (denoted by empty sets in Table 1). In total, in the example of Figure 1 only 4 high-level changes will be reported as opposed to 12 low-level ones.

Apart from being more concise, the reported high-level changes are also more intuitive. For example, the *Generalize_Domain* operation provides the additional information that the new domain is a superclass of the old. This may be useful for the evaluation and understanding of the change performed. For example, if we know only that a domain changed we cannot presume anything about the validity of the existing data, but if we know that the domain changed to a superclass we can assume, according to RDF/S specification, that validity is not violated [10].

Another interesting example is the *Rename_Class* operation, which is reported instead of the deletion of class $Existing$ and the subsequent addition of $Persistent$. Unlike the changes discussed so far, the detection of *Rename* (as well as other opera-

Added Triples (Low-Level Delta)	Deleted Triples (Low-Level Delta)	Detected Changes (High-Level Delta)
(participants, domain, Event)	(participants, domain, Onset)	Generalize_Domain(participants, Onset, Event)
(Birth, subClassOf, Event)	(Birth, subClassOf, Onset)	Pull_up_Class(Birth, Onset, Event)
–	(Period, type, class)	Delete_Class(Period, ∅, {Event}, ∅, ∅, ∅, ∅)
–	(Event, subClassOf, Period)	
(Stuff, subClassOf, Persistent)	(Stuff, subClassOf, Existing)	Rename_Class(Existing, Persistent)
(started_on, domain, Persistent)	(started_on, domain, Existing)	
(Persistent, type, class)	(Existing, type, class)	

Table 1. Detected Low-level and High-level Changes (From Figure 1)

tions, such as *Merge* and *Split*) requires the use of a matcher that would identify the two concepts ($Existing$ and $Persistent$) to be the same entity using heuristics.

Note also that all the triples of the low-level delta (in Table 1) are associated with one, and only one, high-level change. This allows the partition of low-level changes into well-defined high-level changes. In addition, we are able to do such a partition in only one way. The combination of these two properties guarantees that the detection algorithm will be able to handle all possible low-level deltas in a *deterministic* manner, i.e., that any set of low-level changes between two versions would be associated with *one, and only one*, set of high-level changes. The latter requirement calls for a careful definition of the change operations and is not directly related to the detection algorithm *per se*. Thus, we claim that the detection algorithm should be based on the defined language of changes, instead of the other way round.

Defining a language with the above properties is a challenging task, because it requires establishing a tradeoff between partly conflicting requirements. On the one hand, coarse-grained operations are necessary in order to achieve concise and intuitive deltas. On the other, fine-grained operations are necessary in order to capture subtle differences between a pair of versions. The existence of both fine-grained and coarse-grained operations in the language may allow the association of the same set of low-level changes with several different sets of high-level ones, thus jeopardizing determinism. In the next section, we will describe a language and a detection algorithm that avoids these problems and provably satisfies the above properties, while being efficient.

3 Change Detection Framework, Language and Algorithm

3.1 Formal Definitions

The representation of knowledge in RDF [12] is based on triples of the form (*subject*, *predicate*, *object*). Assuming two disjoint and infinite sets $\mathbf{U}$, $\mathbf{L}$, denoting the URIs and literals respectively, $\mathcal{T} = \mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$ is the set of all triples. An *RDF Graph* V is defined as a set of triples, i.e., $V \subseteq \mathcal{T}$. In this paper, we ignore unnamed resources, also called *blank nodes*.

RDFS [2] introduces some built-in classes (resource, class, property) which are used to determine the *type* of each resource. Following the approach of [20], we assume that each resource is associated with one type determined by the triples that the resource

participates in. The available types are individual, schema class (class for brevity), property, metaclass, metaproperty.

The typing mechanism allows us to concentrate on nodes of RDF Graphs, rather than triples, which is closer to ontology curators' perception and useful for defining intuitive high-level changes. RDFS [2] provides also *inference semantics*, which is of two types, namely *structural inference* (provided mainly through the transitivity of subsumption relations) and *type inference* (provided by the typing system, e.g., if p is a property, the triple $(p, \mathsf{type}, \mathsf{property})$ can be inferred). Per [2], structural inference semantics is given by the inference rules in Table 2. Typing inference semantics is given by the rules in Table 3.

Transitivity of subClassOf

$$I_d^{(1)}: \frac{(C_1, \mathsf{subClassOf}, C_2), (C_2, \mathsf{subClassOf}, C_3)}{(C_1, \mathsf{subClassOf}, C_3)}$$

Transitivity of subPropertyOf

$$I_d^{(2)}: \frac{(P_1, \mathsf{subClassOf}, P_2), (P_2, \mathsf{subClassOf}, P_3)}{(P_1, \mathsf{subClassOf}, P_3)}$$

Transitivity of class instantiation

$$I_d^{(3)}: \frac{(x, \mathsf{type}, C_1), (C_1, \mathsf{subClassOf}, C_2)}{(x, \mathsf{type}, C_2)}$$

Transitivity of property instantiation

$$I_d^{(4)}: \frac{(P_1, \mathsf{subPropertyOf}, P_2), (x_1, P_1, x_2)}{(x_1, P_2, x_2)}$$

Table 2. RDFS Structural Inference Rules

x is an *individual*	$(x, \mathsf{type}, \mathsf{resource})$
x is a *schema class*	$(x, \mathsf{type}, \mathsf{class})$, $(x, \mathsf{subClassOf}, \mathsf{resource})$
x is a *property*	$(x, \mathsf{type}, \mathsf{property})$
x is a *metaclass*	$(x, \mathsf{type}, \mathsf{class})$, $(x, \mathsf{subClassOf}, \mathsf{class})$
x is a *metaproperty*	$(x, \mathsf{type}, \mathsf{class})$, $(x, \mathsf{subClassOf}, \mathsf{property})$

Table 3. RDFS Typing Inference Rules

The notion of validity has been described in various fragments of the RDFS language, and is used to overrule certain triple combinations. The validity constraints that we consider in this work concern the *type uniqueness*, i.e., that each resource has a unique type, the acyclicity of subClassOf and subPropertyOf relations and that the subject and object of the instance of some property should be correctly classified under the domain and range of the property respectively. For a full list of the related validity constraints, see [20].

An RDF Graph satisfying the validity constraints is called a *valid* RDF Graph. Given a valid RDF Graph V, the RDF Graph containing all triples that are either explicit or can be inferred from explicit triples in V (using *both* types of inference), is called the *closure* of V and denoted by $Cl(V)$. An *RDF/S Knowledge Base* (*RDF/S KB*) V is a

valid RDF Graph which is closed with respect to *type inference*, i.e., it contains all the triples that can be inferred from V using type inference.

Consider a pair of asynchronously produced RDF/S KBs ($V_1, V_2 \subseteq \mathcal{T}$). A change can be defined on the basis of the triples that must exist (or not exist) in V_1 and/or V_2 in order for it to be detected. This notion is formalized as follows:

Definition 1. *Let V_1, V_2 be two RDF/S KBs. The* low-level delta *between V_1, V_2, denoted by $\Delta(V_1, V_2)$ (or simply Δ) is a pair of sets of triples defined as: $\Delta(V_1, V_2) = \langle V_2 \setminus V_1, V_1 \setminus V_2 \rangle$. For brevity, we will use the notation Δ_1, Δ_2 for $V_2 \setminus V_1, V_1 \setminus V_2$ respectively ($\Delta_1 \subseteq \mathcal{T}$, $\Delta_2 \subseteq \mathcal{T}$).*

Note that Δ_1 corresponds to the triples *added* in V_1 to get V_2, and Δ_2 corresponds to the triples *deleted* from V_1 to get V_2. The low-level delta alone may not be enough to fully capture the intuition behind a change: sometimes we need to consider conceptual information that remained unchanged (see, e.g., the change of the domain of $participants$ in Figure 1 and the subsequent analysis in Section 2). Therefore, the definition of the detection semantics should consist of the triple(s) that must exist in the low-level delta, as well as of a set of conditions that must hold (in V_1 and/or V_2) in order for the detection to take place:

Definition 2. *A* change c *is defined as a triple $\langle \delta_1, \delta_2, \phi \rangle$, where:*

- $\delta_1 \subseteq \mathcal{T}$*:* required added triples*. Corresponds to the triples that should be in V_2 but not in V_1 (i.e., in Δ_1), in order for c to be detected.*
- $\delta_2 \subseteq \mathcal{T}$*:* required deleted triples*. Corresponds to the triples that should be in V_1 but not in V_2 (i.e., in Δ_2), in order for c to be detected.*
- ϕ*:* required conditions*. Corresponds to the conditions that should be true in order for c to be detected. A condition is a logical formula consisting of atoms of the form $t \in V$ or $t \notin V$, where $t \in \mathcal{T}$ and V is of the form V_i or $Cl(V_i)$ for $i \in \{1, 2\}$.*

For simplicity, we will denote by $\delta_1(c)$ ($\delta_2(c)$) the required added (deleted) triples of a change c, and by $\phi(c)$ the required conditions of c. The changes defined in this work form a language of changes, that we denote by $\mathcal{L}$. Changes are classified into basic and composite; in the end of this report, one can find a full list of all the basic and composite changes in $\mathcal{L}$. In this work, we restrict our attention to changes for which $\delta_1 \cup \delta_2 \neq \emptyset$ and $\delta_1 \cap \delta_2 = \emptyset$. The first condition guarantees that at least *something* must be in Δ_1 or Δ_2 (i.e., something has actually changed between V_1 and V_2). The second condition guarantees that no change would require the addition and deletion of the same triple to happen at the same time.

As discussed in Section 2, both fine-grained and coarse-grained high-level changes are necessary in order to support determinism, conciseness and intuitiveness. For this reason, we follow a common approach in the literature [10,17,21] and classify high-level changes into *basic* and *composite*. Basic changes are fine-grained and describe a change in one node or edge of the RDF/S KB taking into account RDF/S semantics. On the other hand, composite changes are coarse-grained and closer to the user's intuition, as they describe, in a concise way, changes affecting several nodes and/or edges of the RDF/S KB. The introduction of the two levels should be done carefully,

as it may cause problems with determinism. For instance, in the motivating example (Figure 1 and Table 1), the deleted triple $(participants, \mathsf{domain}, Onset)$ could be associated with the basic change *Delete_Domain(participants,Onset)*, as well as with the composite change *Generalize_Domain(participants,Onset,Event)* (see Appendix). This double association would jeopardize determinism, because the same low-level delta would correspond to two different high-level deltas. To avoid this problem, we define two different notions, *detectability* and *initial detectability*, and postulate that the detection of composite changes takes precedence over the detection of basic ones.

Definition 3. *Consider a composite change c_c and a basic change c_b. Then, we say that c_c* subsumes *c_b iff $\delta_1(c_b) \subseteq \delta_1(c_c), \delta_2(c_b) \subseteq \delta_2(c_c)$ and $\phi(c_c) \vdash \phi(c_b)$. The set of subsumed basic changes for each composite change is denoted by $\Sigma(c_c)$.*

In the tables at the end of this document, one can see the basic changes that are subsumed by each composite change.

The detection semantics of changes is as follows:

Definition 4. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2)$ and a change c. Then, c is* initially detectable *iff $\delta_i(c) \subseteq \Delta_i$, $i \in \{1, 2\}$, and $\phi(c)$ is true.*
If c is a composite change, then c is detectable *iff it is initially detectable.*
If c is a basic change, then c is detectable *iff it is initially detectable and there is no initially detectable composite change (say c') for which $b \in \Sigma(c')$.*

In our running example, *Change_Domain(participants,Onset,Event)* is not initially detectable (thus, not detectable) because its conditions are not true (specifically, the part: $(Onset, \mathsf{subClassOf}, Event) \notin Cl(V_1) \vee (Onset, \mathsf{subClassOf}, Event) \notin Cl(V_2)$). On the other hand, *Generalize_Domain(participants,Onset,Event)* is initially detectable; given that it is a composite change, it is also detectable. Finally, the basic change *Delete_Domain(participants,Onset,Event)* is initially detectable, but not detectable (because *Generalize_Domain(participants,Onset,Event)* is initially detectable).

So far, we were only concerned with detection semantics of changes. However, changes can also be applied upon RDF/S KBs, where the application and detection semantics of a set of changes should be consistent. To be more precise, given two RDF/S KBs V_1, V_2, the application (upon V_1) of the delta computed between them should give V_2, irrespective of the order of application of the changes [24]. Therefore, we also need to define the application semantics of changes:

Definition 5. *Consider an RDF/S KB V and a change c. The* application *of c upon V, denoted by $V \bullet c$ is defined as: $V \bullet c = (V \cup \delta_1(c)) \setminus \delta_2(c)$.*

As an example, the application of *Generalize_Domain(participants,Onset,Event)* would lead to the addition of the triple $(participants, \mathsf{domain}, Event)$ and the deletion of $(participants, \mathsf{domain}, Onset)$.

3.2 Formal Results on the Proposed Language of Changes

A language of changes can be simply defined as a set of changes. This set corresponds to the set of changes that the language admits. Consequently:

Definition 6. *A language of changes is a set of changes (as defined in Definition 2).*

We argue that a language of changes should satisfy a number of desiderata in order for it to be useful in real-world applications:

- *Completeness.* Every modification upon an ontology must be representable by a set of changes of the language. In other words, for any given pair of versions, there should be at least one set of changes that describes their differences. This property guarantees that any possible change can be captured by the given language, so the detection algorithm can always process the input and return a corresponding delta.
- *Non-ambiguity.* All changes in the language must be non-ambiguous in the sense that there should not be more than one set of changes that could describe the same evolution of an ontology to a newer version. More specifically, each low-level change should be associated with one, and only one high-level change, and each set of low-level changes should be associated with one, and only one set of high-level ones. This property is necessary in order to establish a deterministic detection process and to avoid ambiguities that would jeopardize any effort for a formal or automated treatment of deltas. Combined with completeness, this property guarantees that there is a total function connecting pairs of versions with deltas.
- *Reversibility.* Every change in the language must have a unique reverse in the sense that both ways of the evolution process must be supported. This way, by keeping only the newest version in a repository and the set of changes that led to it, all previous versions can be reproduced.

The languages that satisfy the aforementioned desiderata are called *rational* languages. In the following section we will prove that our proposed language is a rational language and denote it by $\mathcal{L}$. It should be made clear that there is no unique way to define a rational language. Consequently, we do not claim that the language we propose is the only one satisfying the above properties. In fact, there are many such languages. However, when creating a language considering only the desired theoretical properties, one may end up with a non-intuitive, artificial language without practical use in real-world scenarios. Thus, the intuitiveness of the changes that the language contains is also important and was a critical factor in our design. Having this in mind we can define another property that a language of changes must support and that is the property of *usefulness*. A language of changes must be useful to the process of capturing the development cycle of an ontology in the sense that the reported set of changes should be concise and intuitive. The property of conciseness means that the number of reported changes should be as small as possible without losing readability and expressiveness. The intuitiveness corresponds to the similarity of the language to the original intentions of the editor who performed the changes, or that of a user who would analyze the changes manually. However, this property cannot be theoretically defined or proved to hold for a language as there is no way to define a formal metric to compare languages among them based on how *useful* their changes are. The only possible way to measure the *usefulness* (intuitiveness and conciseness) of a language of changes would have to be through user studies and the examination of the development cycle of real-world ontologies (see section 5).

The following theorem proves that $\mathcal{L}$ satisfies the property of *completeness* as it states that all low-level changes will be associated with at least one detectable high-level change:

Lemma 1. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2) = \langle \Delta_1, \Delta_2 \rangle$ and some $i \in \{1, 2\}$ and $t \in \Delta_i$. If there is some $c \in \mathcal{L}$, such that c: initially detectable and $t \in \delta_i(c)$, then there is some $c' \in \mathcal{L}$, c': detectable and $t \in \delta_i(c')$.*

Proof. If c is detectable as well, then set $c' = c$. If c is not detectable, then, by the definition, c is a basic change, and there is some composite change that subsumes it; thus, by Definition 3, $t \in \delta_i(c')$.

Theorem 1. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2) = \langle \Delta_1, \Delta_2 \rangle$ and the set $C = \{c \in \mathcal{L} | c : detectable\}$. Then, for any $i \in \{1, 2\}$ and $t \in \Delta_i$, there is some $c \in C$ such that $t \in \delta_i(c)$.*

Proof. We will consider all the different triple patterns. By Lemma 1, it suffices to show that for each such pattern there will be at least one initially detectable change with the properties stated in the theorem:

1. $t = (x, \mathsf{type}, y) \in \Delta_1$, where $x, y \in \mathbf{U}$ and $y \notin \{\mathsf{class}, \mathsf{property}, \mathsf{resource}\}$. By definition, $t \in V_2, t \notin V_1$. Given that $t \in V_2$, x is either an individual, or a schema class, or a property or a metaclass or a metaproperty(in V_2). Let us suppose that x is an individual; then, y is a schema class, and, by the type inference and the definition of *Add_Type_To_Individual*, we conclude that for $c =$*Add_Type_To_Individual(x,y)* it holds that $t \in \delta_1(c)$ and that c is initially detectable. Similarly, if x is a schema class, set $c =$*Add_Type_To_Class(x,y)* and if x is a property, set $c =$ *Add_Type_To* _Property(x,y). Finally, if x is a meta class, set $c =$*Add_Type_To_Metaclass(x,y)* and if x is a meta property, set $c =$ *Add_Type_To_Metaproperty(x,y)*
2. $t = (x, \mathsf{type}, y) \in \Delta_2$, where $x, y \in \mathbf{U}$ and $y \notin \{\mathsf{class}, \mathsf{property}, \mathsf{resource}\}$. We use the same reasoning as above for the operations *Delete_Type_From_Individual*, *Delete_Type_From_Class*, *Delete_Type_From_Property*, *Delete_Type_From_Metaclass*, *Delete_Type_From_Metaproperty*.
3. $t = (x, \mathsf{subClassOf}, y) \in \Delta_1$, where $x, y \in \mathbf{U}$ and $y \notin \{\mathsf{class}, \mathsf{property}, \mathsf{resource}\}$. By definition, $t \in V_2$, so x is either a class, or a metaclass, or a metaproperty in V_2. Depending on the case, and by type inference, we can conclude that either $c =$ *Add_Superclass(x,y)*, or $c =$ *Add_SuperMetaclass(x,y)*, or $c =$ *Add_SuperMetaproperty(x,y)* is initially detectable and $t \in \delta_1(c)$.
4. $t = (x, \mathsf{subClassOf}, y) \in \Delta_2$, where $x, y \in \mathbf{U}$ and $y \notin \{\mathsf{class}, \mathsf{property}, \mathsf{resource}\}$. We use the same reasoning as above for the operations *Delete_Superclass(x,y)*, *Delete_SuperMetaclass(x,y)*, and *Delete_SuperMetaproperty(x,y)*.
5. $t = (x, \mathsf{subPropertyOf}, y) \in \Delta_1$, where $x, y \in \mathbf{U}$. By definition, $t \in V_2$, so x is a property in V_2. By type inference, we can conclude that for $c =$*Add_Superproperty(x,y)*, c is initially detectable and $t \in \delta_1(c)$.
6. $t = (x, \mathsf{subPropertyOf}, y) \in \Delta_2$, where $x, y \in \mathbf{U}$. Same as above, we can conclude that for $c =$*Delete_Superproperty(x,y)*, c is initially detectable and $t \in \delta_1(c)$.

7. $t = (x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$, where $x \in \mathbf{U}$. By definition, $t \in V_2$, so x is a schema class in V_2; by type inference, $(x, \mathsf{type}, \mathsf{class}) \in Cl(V_2)$. Moreover, $t \notin V_1$, so x is not a schema class in V_1. If x does not appear at all in V_1, then for $c =$ *Add_Type_Class(x)*, we have that c is initially detectable and $t \in \delta_1(c)$. If x appears in V_1, then it cannot be a schema class, so it is either an individual, or a metaclass, or a metaproperty, or a property; depending on the type of x in V_1, one of the following operations is initially detectable: $c =$ *Retype_Individual_To_Class(x)*, $c =$ *Retype_Metaclass_To_Class(x)*, $c =$ *Retype_Metaproperty_To_Class(x)*, $c =$ *Retype_Property_To_Class(x)*, and it holds that $t \in \delta_1(c)$.
8. $t = (x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$, where $x \in \mathbf{U}$. Using similar arguments, we conclude that x is a schema class in V_1, and, depending on whether x appears in V_2, and its type, one of the following operations is initially detectable: $c =$ *Delete_Type_Class(x)*, $c =$ *Retype_Class_To_Individual(x)*, $c =$ *Retype_Class_To_Metaclass(x)*, $c =$ *Retype_Class_To_Metaproperty(x)*, $c =$ *Retype_Class_To_Property(x)*, and it holds that $t \in \delta_2(c)$.
9. Using similar arguments as in the above two cases, we can handle all the triples of the form $(x, \mathsf{type}, y) \in \Delta_i$, $(x, \mathsf{subClassOf}, y) \in \Delta_i$, where $x \in \mathbf{U}$, $y \in \{\mathsf{resource}, \mathsf{class}, \mathsf{property}\}$, $i \in \{1, 2\}$. All such triples indicate the addition or deletion of an object, or the retyping of an object.
10. $t = (x, \mathsf{domain}, y) \in \Delta_2$, where $x \in \mathbf{U}, y \in \mathbf{U} \cup \{\mathsf{resource}, \mathsf{class}, \mathsf{property}\}$. By definition, $t \in V_2$, so x is a property in V_2. Thus, by the definition of $c =$*Add_Domain(x,y)* and type inference, we conclude that c is initially detectable and $t \in \delta_2(c)$.
11. $t = (x, \mathsf{domain}, y) \in \Delta_1$, where $x \in \mathbf{U}, y \in \mathbf{U} \cup \{\mathsf{resource}, \mathsf{class}, \mathsf{property}\}$. Same as above, for $c =$*Delete_Domain(x,y)*.
12. $t = (x, \mathsf{range}, y) \in \Delta_i$, where $x \in \mathbf{U}, y \in \mathbf{U} \cup \{\mathsf{resource}, \mathsf{class}, \mathsf{property}, \mathsf{literal}\}, i \in \{1, 2\}$. Same as above, for $c =$*Add_Range(x,y)* or $c =$*Delete_Range(x,y)*
13. $t = (x, \mathsf{comment}, y) \in \Delta_1$, where $x \in \mathbf{U}, y \in \mathbf{L}$. By definition, $t \in V_2$, so x is a class in V_2; by type inference, and the definition of *Add_Comment*, we conclude that, for $c =$*Add_Comment(x,y)*, c is initially detectable and $t \in \delta_1(c)$.
14. $t = (x, \mathsf{comment}, y) \in \Delta_2$, where $x \in \mathbf{U}, y \in \mathbf{L}$. Same as above for $c =$*Delete_Comment(x,y)*.
15. $t = (x, \mathsf{label}, y) \in \Delta_i$, where $x \in \mathbf{U}, y \in \mathbf{L}, i \in \{1, 2\}$. Same as in the case of comments, for $c =$*Add_Label(x,y)* or $c =$*Delete_Label(x,y)* (depending on i).
16. $t = (x, y, z) \in \Delta_1$, where $x, y, z \in \mathbf{U}, y \notin \{\mathsf{subClassOf}, \mathsf{type}, \mathsf{domain}, \mathsf{range}, \mathsf{comment}, \mathsf{label}\}$. By definition $t \in V_2$, so y is a property; thus, by type inference, and for $c =$*Add_Property_Instance(x,z,y)* we conclude that c is initially detectable and $t \in \delta_1(c)$.
17. $t = (x, y, z) \in \Delta_2$, where $x, y, z \in \mathbf{U}, y \notin \{\mathsf{subClassOf}, \mathsf{type}, \mathsf{domain}, \mathsf{range}, \mathsf{comment}, \mathsf{label}\}$. Same as above, for $c =$*Delete_Property_Instance(x,z,y)*.

In order to prove that $\mathcal{L}$ satisfies the property of *Non-ambiguity* we must first show that each low-level change is associated with at most one detectable high-level change.

Theorem 2. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2) = \langle \Delta_1, \Delta_2 \rangle$ and two changes $c_1, c_2 \in \mathcal{L}$. Then one of the following is true:*

1. $\delta_i(c_1) \cap \delta_i(c_2) = \emptyset$ *for* $i \in \{1, 2\}$

2. $\delta_i(c_1) \not\subseteq \Delta_i$ *or* $\delta_i(c_2) \not\subseteq \Delta_i$ *for some* $i \in \{1, 2\}$
3. $\phi(c_j)$ *is not true for some* $j \in \{1, 2\}$
4. c_j *is a basic change,* c_k *is a composite change and* $\delta_1(c_j) \subseteq \delta_1(c_k)$, $\delta_2(c_j) \subseteq \delta_2(c_k)$ *and* $\phi(c_k) \vdash \phi(c_j)$ *for some* $j, k \in \{1, 2\}$, $j \neq k$

Proof. (Sketch) For most pairs of changes, condition 1 holds.
For the pairs of basic changes where condition 1 does not hold, the proof uses the validity constraints and in most cases the constraint that each URI must be of one type only. For example, for the pair: c_1 =*Add_Type_Metaclass(a)*, c_2 =*Add_Type_Class(a)*, condition 2 holds because if condition 2 did not hold, we would have (by the required added triples): $(a, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$, $(a, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$, so $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V_2)$, $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V_2)$, which would imply that a is both a class and a metaclass (a contradiction, per our validity rules). Similarly, for the pair: c_1 =*Add_Superclass(a,b)*, c_2 =*Add_SuperMetaclass(a,b)*, condition 3 holds because if condition 3 did not hold, we would have (by the conditions): $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V_2)$, $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V_2)$, which would imply that a is both a class and a metaclass (a contradiction, per our validity rules).
For the pairs of composite changes where condition 1 does not hold the proof goes as above. For example, the conditions of *Pull_Up_Class(a,B,C)* and *Pull_Down_Class(a,B,C)* are mutually exclusive, because the former requires that all the classes in B are subclasses of the classes in C, whereas the latter requires that all the classes in B are superclasses of the classes in C; if both were true, we would have cycles in the class subsumption hierarchy, a contradiction by our validity rules. Thus, in this example, condition 3 holds. Similarly, for the other cases.
Finally, let us consider the case where the changes are of different type, say c_1 is a basic change and c_2 is a composite one. In most cases, it will be the case that c_2 subsumes c_1, so condition 4 is true. For example, set c_1 =*Add_Type_Class(a)* and c_2 =*Add_Class (a,*$P_1, P_2, P_3, P_4, P_5, P_6$*)*. For c_1, c_2 condition 1 does not hold, but condition 4 does. In other cases, one of the other conditions will hold. For example, if c_1 =*Retype_Individual_To_Class(a)* and c_2 as above, condition 1 does not hold but condition 3 does (because c_1 requires that a is an individual in V_1, whereas c_2 requires that a does not exist in V_1).

This theorem shows that the changes in $\mathcal{L}$ have been chosen in such a way that a change is either not detectable, or irrelevant to other detectable changes. In particular, if condition 1 is true then the required added triples of c_1 are disjoint from the ones of c_2 (similar for the deleted); this means that c_1, c_2 cannot be associated with the same low-level change. If conditions 2 or 3 are true then at least one of c_1, c_2 is not detectable (by Definition 4), so, again, a low-level change cannot be associated with both changes. Finally, if condition 4 is true, then one of the changes (c_k) is a composite one and more "general" than the basic one (c_j). Therefore, by Definition 4 again, even if both of them are initially detectable, only c_k will be detectable. The usability of this theorem is to set the conditions that should hold for a change in order to allows us to add it to $\mathcal{L}$ without jeopardizing determinism.

Given this analysis, the following theorem is straightforward and proves that any two changes in $\mathcal{L}$ are non-ambiguous, ergo $\mathcal{L}$ satisfies the property of *Non-ambiguity* :

Theorem 3. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2) = \langle \Delta_1, \Delta_2 \rangle$ and the set $C = \{c \in \mathcal{L} | c : detectable\}$. Then, for any two changes $c_1, c_2 \in C$, it holds that $\delta_i(c_1) \cap \delta_i(c_2) = \emptyset$ for $i \in \{1, 2\}$.*

Proof. Given that $c_1, c_2 \in C$, it follows by the definition of detectability that conditions 2, 3 and 4 of Theorem 2 are not true; thus, condition 1 of Theorem 2 is true, i.e.: $\delta_i(c_1) \cap \delta_i(c_2) = \emptyset$ for $i \in \{1, 2\}$.

The following corollary corresponds to the determinism property of the detection process, as it shows that the set of low-level changes can be fully partitioned into disjoint subsets, each corresponding to one high-level change.

Corollary 1. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2) = \langle \Delta_1, \Delta_2 \rangle$ and the set $C = \{c \in \mathcal{L} | c : detectable\}$. Then:*

- *For all $c_1, c_2 \in C$ it holds that $\delta_i(c_1) \cap \delta_i(c_2) = \emptyset$ for $i \in \{1, 2\}$.*
- *$\bigcup_{c \in C} \delta_i(c) = \Delta_i$ for $i = 1, 2$.*

Proof. The first claim is Theorem 3. For the second, note that for $i = 1, 2$ it holds that $\bigcup_{c \in C} \delta_i(c) \subseteq \Delta_i$ by the definition of detectability and $\bigcup_{c \in C} \delta_i(c) \supseteq \Delta_i$ by Theorem 1.

In the rest of this subsection, we will consider the application of changes and show that the detection and application semantics are such that, given V_1, V_2, the application of the set of detectable changes between them upon V_1 would give V_2. Before showing that, we must generalize Definition 5 to apply for sets of changes; given that elements in a set are unordered, before doing this generalization, we must first guarantee that the order of application does not matter.

Definition 7. *Two changes c_1, c_2 are called* conflicting *iff $(\delta_1(c_1) \cap \delta_2(c_2)) \cup (\delta_1(c_2) \cap \delta_2(c_1)) \neq \emptyset$. A set C of changes is called* conflicting *iff C contains at least one pair of conflicting changes.*

By definition, c_1, c_2 are conflicting iff the detection of c_1 requires the addition (or deletion) of a triple whose deletion (or addition) is required by c_2. For example, *Delete_Domain(participants,Event)* is conflicting with *Change_Domain(participants, Onset,Event)*, because the detection of the former requires the deletion of $(participants,$ domain, $Event)$ whereas the latter requires the same triple to be added. It is easy to see that when applying a conflicting set of changes upon a version, the order matters (e.g., in the above example, depending on the order, $Event$ would, or would not, be the domain of $participants$); however, for non-conflicting sets of changes, the order is irrelevant:

Theorem 4. *Consider an RDF/S KB V and a non-conflicting set of changes $C = \{c_1, \ldots, c_n\}$. Then, for any permutation π over the set of indices $\{1, \ldots, n\}$ it holds that: $(\ldots((V \bullet c_1) \bullet c_2) \bullet \ldots) \bullet c_n = (\ldots((V \bullet c_{\pi(1)}) \bullet c_{\pi(2)}) \bullet \ldots) \bullet c_{\pi(n)}$.*

Proof. Take some permutation π; then:

$(\ldots((V \bullet c_1) \bullet c_2) \bullet \ldots) \bullet c_n =$
$(\ldots(((V \setminus \delta_2(c_1)) \cup \delta_1(c_1)) \bullet c_2) \bullet \ldots) \bullet c_n =$
$\ldots =$
$(\ldots(((((V \setminus \delta_2(c_1)) \cup \delta_1(c_1)) \setminus \delta_2(c_2)) \cup \delta_2(c_1)) \ldots) \setminus \delta_2(c_n)) \cup \delta_1(c_n) =$
$V \setminus (\delta_2(c_1) \cup \delta_2(c_2) \cup \ldots \cup \delta_2(c_n)) \cup (\delta_1(c_1) \cup \delta_1(c_2) \cup \ldots \cup \delta_1(c_n)) =$
$V \setminus (\delta_2(c_{\pi(1)}) \cup \delta_2(c_{\pi(2)}) \cup \ldots \cup \delta_2(c_{\pi(n)})) \cup (\delta_1(c_{\pi(1)}) \cup \delta_1(c_{\pi(2)}) \cup \ldots \cup \delta_1(c_{\pi(n)})) =$
$(\ldots(((((V \setminus \delta_2(c_{\pi(1)})) \cup \delta_1(c_{\pi(1)})) \setminus \delta_2(c_{\pi(2)})) \cup \delta_2(c_{\pi(2)})) \ldots) \setminus \delta_2(c_{\pi(n)})) \cup \delta_1(c_{\pi(n)}) =$
$\ldots =$
$(\ldots(((V \setminus \delta_2(c_{\pi(1)})) \cup \delta_1(c_{\pi(1)})) \bullet c_{\pi(2)}) \bullet \ldots) \bullet c_{\pi(n)} =$
$(\ldots((V \bullet c_{\pi(1)}) \bullet c_{\pi(2)}) \bullet \ldots) \bullet c_{\pi(n)}.$

The manipulations in the above sequence are only possible because of the hypothesis that C is a non-conflicting set of changes (otherwise, we wouldn't be able, e.g., to change the order of $\setminus, \cup$).

Now we can generalize Definition 5; note that the generalization can be made for non-conflicting sets only:

Definition 8. *Consider an RDF/S KB V and a non-conflicting set of changes $C = \{c_1, \ldots, c_n\}$. The* application *of C upon V, denoted by $V \bullet C$, is defined as: $V \bullet C = (\ldots((V \bullet c_1) \bullet c_2) \bullet \ldots) \bullet c_n$.*

Theorem 5 shows that we can apply the detected delta upon one version in order to get the other; given that we cannot define the application of sets of changes for conflicting sets, a critical intermediate result is that the detected delta is always non-conflicting:

Theorem 5. *Consider two RDF/S KBs V_1, V_2 and the set $C = \{c \in \mathcal{L} | c : detectable\}$. Then it can be shown that C is non-conflicting and $V_1 \bullet C = V_2$.*

Proof. Suppose that $C = \{c_1, \ldots, c_n\}$. Set $A = \bigcup_{i=1,\ldots,n} \delta_1(c_i)$, $B = \bigcup_{i=1,\ldots,n} \delta_2(c_i)$. By Corollary 1, $A = \Delta_1, B = \Delta_2$; thus, $A \cap B = \Delta_1 \cap \Delta_2 = (V_2 \setminus V_1) \cap (V_1 \setminus V_2) = \emptyset$, so C is non-conflicting.
Combining the above results and the related definitions:
$V_1 \bullet C = (V_1 \setminus B) \cup A =$
$(V_1 \setminus \Delta_2) \cup \Delta_1 =$
$(V_1 \setminus (V_1 \setminus V_2)) \cup (V_2 \setminus V_1) = V_2.$

An interesting corollary of Theorem 4 is that changes are *composable*, i.e., they can be applied either simultaneously or sequentially:

Theorem 6. *Consider an RDF/S KB V and two sets of changes C_1, C_2 such that C_1, C_2, $C_1 \cup C_2$ are non-conflicting. Then: $(V \bullet C_1) \bullet C_2 = (V \bullet C_2) \bullet C_1 = V \bullet (C_1 \cup C_2)$.*

Proof. This proof is trivial using Theorem 4 and Definition 8.

The last desiderata that $\mathcal{L}$ must satisfy in order to be rational is *Reversibility*, namely the fact that for each change c, there is some change whose application cancels the effects of c.

Definition 9. *A change c_1 is called the* reverse *of c_2 iff $\delta_1(c_1) = \delta_2(c_2)$ and $\delta_2(c_1) = \delta_1(c_2)$.*

Theorem 7. *Consider two changes c_1, c_2 such that c_2 is the reverse of c_1. Then, c_1 is the reverse of c_2 and c_1, c_2 are conflicting.*

It can be shown that the reverse of a change always exists and is unique:

Theorem 8. *Every change in $\mathcal{L}$ has a unique reverse.*

Proof. The proof is obvious by the definition of the changes; at the list of the changes at the end of this document, one can see the reverse of each change (the proof is obvious by the definition).

For example, the reverse of *Change_Domain(participants,Onset,Event)* is *Change_Domain(participants,Event,Onset)*. In the following, we will denote by c^{-1} the reverse of $c \in \mathcal{L}$. Reversibility is useful for returning to the original version of the RDF/S KB:

Theorem 9. *Consider two RDF/S KBs V_1, V_2 and the set $C = \{c \in \mathcal{L} | c : detectable\}$. Set $C^{-1} = \{c^{-1} | c \in C\}$. Then it can be shown that C^{-1} is non-conflicting and $V_2 \bullet C^{-1} = V_1$.*

Proof. Suppose that $C = \{c_1, \ldots, c_n\}$. Then $C^{-1} = \{c_1^{-1}, \ldots, c_n^{-1}\}$.
Set $A = \bigcup_{i=1,\ldots,n} \delta_1(c_i), B = \bigcup_{i=1,\ldots,n} \delta_2(c_i)$.
It is easy to see (by the definition of C^{-1} and Definition 9) that $B = \bigcup_{i=1,\ldots,n} \delta_1(c_i^{-1})$, $A = \bigcup_{i=1,\ldots,n} \delta_2(c_i^{-1})$.
By Theorem 5, C is non-conflicting, so $A \cap B = \emptyset$, so C^{-1} is non-conflicting.
By Corollary 1, $A = \Delta_1, B = \Delta_2$.
Combining the above results and the related definitions:
$V_2 \bullet C^{-1} = (V_2 \setminus A) \cup B =$
$(V_2 \setminus \Delta_1) \cup \Delta_2 =$
$(V_2 \setminus (V_2 \setminus V_1)) \cup (V_1 \setminus V_2) = V_1$

Theorem 9 shows that any set of changes performed upon an RDF/S KB can be canceled by applying the reverse upon the result. This allows for both "undoing" an unwanted change, and reproducing older versions of an RDF/S KB based on the newest versions and the deltas that led to it.

Corollary 2. *Language $\mathcal{L}$ is a rational language.*

Proof. In order for $\mathcal{L}$ to be *rational* it must be proved that it satisfies the properties of *Completeness,Non-ambiguity* and *Reversibility*. The proofs are straightforward using Theorems 1, 3 and 8 respectively.

3.3 Change Detection Algorithm

An essential part of our approach is the detection algorithm for $\mathcal{L}$ which should be efficient, scalable and should correctly return the detectable changes. Consider a high-level change c. In order for c to be detectable, all low-level changes in $\delta_1(c), \delta_2(c)$ must be in Δ, and the respective $\phi(c)$ must be true. The first step of the algorithm is to pick a low-level change (i.e., a triple in Δ_1 or Δ_2), say $(participants, \mathsf{domain}, Onset) \in \Delta_2$ (cf. Figure 1 and Table 1). Regardless of the particular input (V_1, V_2), there are certain high-level changes whose detection cannot be triggered by a given low-level change. For example, the deletion of triple ($participants$, domain, $Onset$) cannot be related to the detection of *Delete_Superclass*, as no low-level change of this form appears in the required deleted triples of *Delete_Superclass* (see Appendix). On the other hand, it can potentially trigger the detection of a *Delete_Domain* or a *Change_Domain* operation if the latter is coupled with some other low-level change in Δ specifying the addition of a new domain for $participants$ (see Appendix).

This kind of reasoning reveals that the assignment of low-level changes to high-level ones is an essential part of the detection algorithm. Moreover, we showed above that there can be "race" conditions between basic and composite changes if a low-level change is assigned to both types of high-level changes and the conditions of both are true. We have overcome this problem by giving precedence to the detection of composite changes as explained in section 3. However, in practical terms, i.e., as far as the algorithm is concerned, we divided the algorithm into two parts based on theorem 10 which shows that if the subsumed basic changes of a composite change are initially detectable then the composite change is detectable given that its conditions in ϕ are true (and the opposite). Using this property, we can detect only basic changes in the first part of the algorithm, whereas in the second part we can detect only composite ones. More specifically, in the first part we only consider basic changes and return those that are initially detectable (see Definition 2) while in the second part, we detect composite changes based on the basic changes we have already detected. It should be noted, that the basic changes returned by the first part of the algorithm are initially detectable and not detectable. This is due to the fact that some of these basic changes may be subsumed by a composite in the second part of the algorithm. Thus, detectable are the basic changes that after the execution of the two parts of the algorithm are initially detectable and have not been subsumed by a composite change. In order to use this policy we must first prove that the detection of a composite change based on its subsumed basic changes is equivalent to its detection based on the low-level changes contained in δ_1, δ_2.

Theorem 10. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2)$ and a composite change $c : Cond(c) = true$. Then $c : detectable \Leftrightarrow \forall b \in \Sigma(c) : detectable$.*

Proof. (Sketch) The proof is divided in two parts. In the first part we will show that if c is detectable then all low-level changes contained in $\delta_1(c), \delta_2(c)$ can be assigned to detectable basic changes. From Definition 4 it holds that $\forall c \in \mathcal{L}$ if $c : detectable$ then $\phi(c)$ = true and $\delta_1(c) \subseteq \Delta_1$ and $\delta_2(c) \subseteq \Delta_2$.
Assume c = *Add_Class(a,*P_1*,*P_2*,*P_3*,*P_4*,*P_5*,*P_6*)* : detectable.

- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(a, \mathsf{type}, \mathsf{class}), (a, \mathsf{subClassOf}, \mathsf{resource})\}$ and $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$

$\delta_1(Add_Type_Class(a)) \subseteq \Delta_1$ and $\phi(Add_Type_Class(a)) \Rightarrow$
$Add_Type_Class(a) : detectable$.

- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(a, \mathsf{subClassOf}, p), \forall p \in P_1\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_1 : \delta_1(Add_Superclass(a, p)) \subseteq \Delta_1$ and
  $\phi(Add_Superclass(a, p))$ = true $\Rightarrow$
  $\forall p \in P_1 : Add_Superclass(a, p)$ is $detectable$.
- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(p, \mathsf{subClassOf}, a), \forall p \in P_2\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_2 : \delta_1(Add_Superclass(p, a)) \subseteq \Delta_1$ and
  $\phi(Add_Superclass(p, a))$ = true $\Rightarrow$
  $\forall p \in P_2 : Add_Superclass(p, a)$ is $detectable$.
- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(a, \mathsf{type}, p), \forall p \in P_3\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_3 : \delta_1(Add_Type_To_Class(a, p)) \subseteq \Delta_1$ and
  $\phi(Add_Type_To_Class(a, p))$ = true $\Rightarrow$
  $\forall p \in P_3 : Add_Type_To_Class(a, p)$ is $detectable$. .
- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(p, \mathsf{type}, a), \forall p \in P_4\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_4 : \delta_1(Add_Type_To_Class(p, a)) \subseteq \Delta_1$ and $\phi(Add_Type_To_Class(p, a))$
  = true $\Rightarrow$
  $\forall p \in P_4 : Add_Type_To_Class(p, a)$ is $detectable$.
- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(a, \mathsf{comment}, p), \forall p \in P_5\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_5 : \delta_1(Add_Comment(a, p)) \subseteq \Delta_1$ and
  $\phi(Add_Comment(a, p))$ = true $\Rightarrow$
  $\forall p \in P_5 : Add_Comment(a, p))$ is $detectable$. .
- $\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \supseteq \{(a, \mathsf{label}, p), \forall p \in P_6\}$ and
  $\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true $\Rightarrow$
  $\forall p \in P_6 : \delta(Add_Label(a, p)) \subseteq \Delta_1$ and
  $\phi(Add_Label(a, p))$ = true $\Rightarrow$
  $\forall p \in P_6 : Add_Label(a, p))$ is $detectable$.

So, if *Add_Class(a,*P_1*,*P_2*,*P_3*,*P_4*,*P_5*,*P_6*)*: $detectable \Rightarrow$
$\forall b \in \Sigma(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) : detectable$. The second part of the proof verifies that if the basic changes contained in $\Sigma(c)$ are $detectable$ then c is also $detectable$.
$\forall b \in \Sigma(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) : detectable \Rightarrow$
$\cup_{\forall b \in \Sigma(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))} \delta_1(b) \subseteq \Delta_1$ and
$\cup_{\forall b \in \Sigma(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))} \delta_2(b) \subseteq \Delta_2 \Rightarrow$
$\{(a, \mathsf{subClassOf}, p), \forall p \in P_1\} \subseteq \Delta_1 \wedge$
$\{(p, \mathsf{subClassOf}, a), \forall p \in P_2\} \subseteq \Delta_1 \wedge$
$\{(a, \mathsf{type}, p), \forall p \in P_3\} \subseteq \Delta_1 \wedge$
$\{(p, \mathsf{type}, a), \forall p \in P_4\} \subseteq \Delta_1 \wedge$
$\{(a, \mathsf{comment}, p), \forall p \in P_5\} \subseteq \Delta_1 \wedge$

$\{(a, \mathsf{label}, p) \forall p \in P_6\} \subseteq \Delta_1$,
$\{(a, \mathsf{type}, \mathsf{class}), (a, \mathsf{subClassOf}, \mathsf{resource})\} \subseteq \Delta_1 \Rightarrow$
$\delta_1(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6)) \subseteq \Delta_1 \Rightarrow$
$Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6) : detectable$ since
$\phi(Add_Class(a, P_1, P_2, P_3, P_4, P_5, P_6))$ = true from hypothesis.

In the same manner, it can be proved that Theorem 10 holds for the rest of the high-level changes.

An interesting corollary of the previous theorem is the following.

Corollary 3. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2)$ and a composite change $c : detectable$. Then, $\cup_{b \in \Sigma(c)} \delta_1(b) = \delta_1(c), \cup_{b \in \Sigma(c)} \delta_2(b) = \delta_2(c)$.*

Proof. From Definition 4 it holds that if $c : detectable$ then $\phi(c)$ = true and $\delta_1(c) \subseteq \Delta_1$ and $\delta_2(c) \subseteq \Delta_2$. Moreover, from the previous theorem it holds that $\forall b \in \Sigma(c) : detectable$. This means that $\forall b \in \Sigma(c) : \phi(b)$ = true and $\delta_1(b) \subseteq \Delta_1$ and $\delta_2(b) \subseteq \Delta_2$. Finally, if we use the opposite direction of the equation of the previous theorem it holds that if $\forall b \in \Sigma(c) : detectable$ then $c : detectable$. So, $\cup_{b \in \Sigma(c)} \delta_1(b) = \delta_1(c)$ and $\cup_{b \in \Sigma(c)} \delta_2(b) = \delta_2(c)$

As described above, by dividing the algorithm into two parts, the low-level changes are assigned only to basic changes whereas for the detection of composite changes the previously detected basic changes are used. This assignment is implemented by methods $findPotentialBasicChanges$ and $findPotentialCompositeChanges$ (Algorithms 1, 2). In short, the first method takes a particular triple in Δ (i.e., a low-level change) and determines the basic changes whose detection could, potentially, be triggered (set $potC$) by the given low-level change. On the other hand, the second method takes a particular basic change from the already detected basic changes and determines the composite changes whose detection could, potentially, be triggered (set *potC*). It should be noted that the determination includes not only the operation to be considered but also the parameters of the operation. In order to determine the association of low-level changes to basic ones method $findPotentialBasicChanges$ uses Tables 4, 5 as follows: if the selected triple t is in the left column of Table 4 or 5, then the method checks whether the low-level changes in the middle column appear in Δ. If so, then t *could* trigger the detection of the basic changes in the right column, so these basic changes are put in set *potC*. In our example, *potC* will contain *Delete_Domain(participants,Onset)*. A similar case appears for the execution of method $findPotentialCompositeChanges$ where the selected (initially detectable) basic change of the left column of Table 6 leads to the potentially detectable composite changes in the right column, if the basic changes in the middle column are also initially detectable. In our example, the composite changes *Change_Domain(participants, Onset, Event)* and *Generalize_Domain(participants, Onset, Event)* will be put it *potC* since they are also coupled with the initially detectable basic change *Add_Domain(participants, Event)*.

In order to build these tables the exploitation of assignment functions is needed to determine which low/high level change patterns can be assigned to what high-level

changes. More specifically, to determine the association between triples and basic changes we are going to use the following two functions: for every triple $t \in \Delta$

$$requiredB(t, \Delta) := \{(\delta_1(b), \delta_2(b)) \mid t \in \Delta_i \Rightarrow t \in \delta_i(b), i = 1, 2, b \in \mathcal{L}, b = basic\}$$

and

$$detectedB(\delta'_1, \delta'_2, \Delta) = \{b \mid \delta'_1 = \delta_1(b), \delta'_2 = \delta_2(b), \delta_1(b) \subseteq \Delta_1, \delta_2(b) \subseteq \Delta_2\}$$

For the association between basic and composite changes we are going to use the following functions: for every basic change in $\mathcal{L}$

$$requiredC(b) := \{\Sigma(c) \mid b \in \Sigma(c), c \in \mathcal{L}, c = composite\} = B$$

and

$$detectedC(B, \Delta) = \{c \mid B = \Sigma(c), \forall b \in \Sigma(c) : detectable\}$$

Having defined the above assignment functions, the population of Tables 4, 5 is straightforward. For each basic change, we select one "representative" low-level change, t, from δ_1, δ_2 and use it as the input parameter of function $requiredB$. This low-level change constitutes the left column of Tables 4, 5. The remaining low-level changes in δ_1, δ_2 are the result of function $requiredB$ and constitute the middle column of the tables. Note that the middle column does not contain t as it is already present in the right column. (although it is included in the result of $requiredB$) The right column is populated with the basic changes that have the same δ_1, δ_2 to the respective change and therefore correspond to the same line. These basic changes are returned by function $detectedB(requiredB(t, \Delta))$. Note that all such changes (high-level and low-level) use variables as their parameters, not fixed URIs.

In order to populate Table 6, we proceed in the same manner as before. For each composite change, we select one "representative" basic change, b, from Σ and use it as the input parameter of function $requiredC$. This basic change constitutes the left column of Table 6. The remaining basic changes in Σ are the result of function $requiredC$ and constitute the middle column of the table. Again, b is not included in the middle column of the table. The right column is populated with the composite changes that have the same Σ to the respective change and therefore correspond to the same line. These composite changes are returned by function $detectedC(requiredC(b, \Delta))$. Note that as in the previous case all changes use variables as their parameters, not fixed URIs.

Handle t	requiredB(t)	detectedB
$(x, \mathsf{subClassOf}, z) \in \Delta_1$	–	*Add_Superclass(x,z),* *Add_SuperMetaclass(x,z),* *Add_SuperMetaproperty(x,z)*
$(x, \mathsf{subPropertyOf}, z) \in \Delta_1$	–	*Add_Superproperty(x,z)*
$(x, \mathsf{subClassOf}, z) \in \Delta_2$	–	*Delete_Superclass(x,z),* *Delete_SuperMetaclass(x,z),* *Delete_SuperMetaproperty(x,z)*
$(x, \mathsf{subPropertyOf}, z) \in \Delta_2$	–	*Delete_Superproperty(x,z)*
$(x, \mathsf{type}, z) \in \Delta_1$	–	*Add_Type_To_Class(x,z),* *Add_Type_To_Metaclass(x,z),* *Add_Type_To_Metaproperty(x,z),* *Add_Type_To_Property(x,z),* *Add_Type_To_Individual(x,z)*
$(x, \mathsf{type}, z) \in \Delta_2$	–	*Delete_Type_From_Class(x,z),* *Delete_Type_From_Metaclass(x,z),* *Delete_Type_From_Metaproperty(x,z),* *Delete_Type_From_Property(x,z),* *Delete_Type_From_Individual(x,z)*
$(x, \mathsf{comment}, z) \in \Delta_1$	–	*Add_Comment(x,z)*
$(x, \mathsf{label}, z) \in \Delta_1$	–	*Add_Label(x,z)*
$(x, \mathsf{comment}, z) \in \Delta_2$	–	*Delete_Comment(x,z)*
$(x, \mathsf{label}, z) \in \Delta_2$	–	*Delete_Label(x,z)*
$(x_1, z, x_2) \in \Delta_1$	–	*Add_Property_Instance(x_1, x_2, z)*
$(x_1, z, x_2) \in \Delta_2$	–	*Delete_Property_Instance(x_1, x_2, z)*
$(x, \mathsf{domain}, z) \in \Delta_1$	–	*Add_Domain(x,z)*
$(x, \mathsf{domain}, z) \in \Delta_2$	–	*Delete_Domain(x,z)*
$(x, \mathsf{range}, z) \in \Delta_1$	–	*Delete_Range(x,z)*
$(x, \mathsf{range}, z) \in \Delta_2$	–	*Delete_Range(x,z)*

Table 4. (a) Look-up Table for Basic Changes

Handle t	requiredB(t)	detectedB
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$	*Add_Type_Class(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	*Retype_Property_To_Class(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	*Retype_Individual_To_Class(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$	*Add_Type_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	*Retype_Property_To_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	*Retype_Individual_To_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$	*Add_Type_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	*Retype_Property_To_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	*Retype_Individual_To_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	–	*Add_Type_Property(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$	*Retype_Class_To_Property(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$	*Retype_Metaclass_To_Property(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$	*Retype_Metaproperty_To_Property(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	*Retype_Individual_To_Property(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	–	*Add_Type_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$	*Retype_Class_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$	*Retype_Metaclass_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$	*Retype_Metaproperty_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	*Retype_Property_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$	*Delete_Type_Class(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	*Retype_Class_To_Property(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	*Retype_Class_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$	*Delete_Type_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	*Retype_Metaclass_To_Property(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	*Retype_Metaclass_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$	*Delete_Type_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	*Retype_Metaproperty_To_Property(x)*
$(x, \mathsf{type}, \mathsf{class}) \in \Delta_2$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$ $(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	*Retype_Metaproperty_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	–	*Delete_Type_Property(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$	*Retype_Property_To_Class(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$	*Retype_Property_To_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$	*Retype_Property_To_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{property}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_1$	*Retype_Property_To_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	–	*Delete_Type_Individual(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$	*Retype_Individual_To_Class(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$	*Retype_Individual_To_Metaclass(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{class}) \in \Delta_1$ $(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$	*Retype_Individual_To_Metaproperty(x)*
$(x, \mathsf{type}, \mathsf{resource}) \in \Delta_2$	$(x, \mathsf{type}, \mathsf{property}) \in \Delta_1$	*Retype_Individual_To_Property(x)*
$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$	*Retype_Class_To_Metaclass(x)*
$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$	*Retype_Class_To_Metaproperty(x)*
$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$	*Retype_Metaclass_To_Class(x)*
$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_2$	*Retype_Metaclass_To_Metaproperty(x)*
$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta_2$	*Retype_Metaproperty_To_Class(x)*
$(x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta_1$	$(x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta_2$	*Retype_Metaproperty_To_Metaclass(x)*

Table 5. (b) Look-up Table for Basic Changes

Handle b	**requiredC(b)**	**detectedC**
Add_Type_Class(x)	*Add_Superclass(x,y_i)* *Add_Superclass(z_i,x)* *Add_Type_To_Class(x,k_i)* *Add_Type_To_Individual(l_i,x)* *Add_Comment(x,m_i)* *Add_Label(x,n_i)*	*Add_Class(x,Y,Z,K,L,M,N)* where $Y = \{y_i\}$, $Z = \{z_i\}$, $K = \{k_i\}$, $L = \{l_i\}$, $M = \{m_i\}$, $N = \{n_i\}$
Delete_Type_Class(x)	*Delete_Superclass(x,y_i)* *Delete_Superclass(z_i,x)* *Delete_Type_From_Class(x,k_i)* *Delete_Type_From_Individual(l_i,x)* *Delete_Comment(x,m_i)* *Delete_Label(x,n_i)*	*Delete_Class(x,Y,Z,K,L,M,N)* where $Y = \{y_i\}$, $Z = \{z_i\}$, $K = \{k_i\}$, $L = \{l_i\}$, $M = \{m_i\}$, $N = \{n_i\}$
Add_Superclass(x,y_0)	*Add_Superclass(x,y_i)*	*Group_Classes(Y,x)* where $Y = \{y_0\} \cup \{y_i\}$
Delete_Superclass(x,y_0)	*Delete_Superclass(x,y_i)*, *Add_Superclass(x,z_i)*	*Pull_up_Class(x,Y,Z)*, *Pull_down_Class(x,Y,Z)*, *Move_Class(x,Y,Z)*, *Change_Superclass(x,Y,Z)*, *Ungroup_Classes(Y,x)* where $Y = \{y_0\} \cup \{y_i\}$, $Z = \{z_i\}$
Delete_Type_From_Class(x,y_0)	*Delete_Type_From_Class(x,y_i)*, *Add_Type_To_Class(x,z_i)*	*Reclassify_Class_Higher(x,Y,Z)*, *Reclassify_Class_Lower(x,Y,Z)*, *Reclassify_Class(x,Y,Z)* where $Y = \{y_0\} \cup \{y_i\}$, $Z = \{z_i\}$
Add_Type_Property(x)	*Add_Superproperty(x, y_i)* *Add_Superproperty(z_i, x)* *Add_Type_To_Property(x, k_i)* *Add_Property_Instance(x, l_{i1}, l_{i2})* *Add_Comment(x, m_i)* *Add_Label(x, n_i)*	*Add_Property(x,Y,Z,K,L,M,N)* where $Y = \{y_i\}$, $Z = \{z_i\}$, $K = \{k_i\}$, $L = \{l_i 1\} \cup \{l_i 2\}$, $M = \{m_i\}$, $N = \{n_i\}$
Delete_Type_Property(x)	*Delete_Superproperty(x, y_i)* *Delete_Superproperty(z_i, x)* *Delete_Type_From_Property(x, k_i)* *Delete_Property_Instance(x, l_{i1}, l_{i2})* *Delete_Comment(x, m_i)* *Delete_Label(x, n_i)*	*Delete_Property(x,Y,Z,K,L,M,N)* where $Y = \{y_i\}$, $Z = \{z_i\}$, $K = \{k_i\}$, $L = \{l_i 1\} \cup \{l_i 2\}$, $M = \{m_i\}$, $N = \{n_i\}$
Add_Superproperty(x, y_0)	*Add_Superproperty(x, z_i)*	*Group_Properties_Under(Y,x)* where $Y = \{y_0\} \cup \{y_i\}$
Delete_Superproperty(x, y_0)	*(Delete_Superproperty(x, y_i)*, *Add_Superproperty(x, z_i)*	*Pull_up_Property(x,Y,Z)*, *Pull_down_Property(x,Y,Z)*, *Move_Property(x,Y,Z)*, *Change_Superproperties(x,Y,Z)*, *Ungroup_Properties_Under(Y,x)* where $Y = \{y_0\} \cup \{y_i\}$, $Z = \{z_i\}$
Delete_Type_From_Property(x, y_0)	*(Delete_Type_From_Property(x, y_i)*, *Delete_Type_From_Property(x, z_i)*	*Reclassify_Property_Higher(x,Y,Z)*, *Reclassify_Property_Lower(x,Y,Z)*, *Reclassify_Property(x,Y,Z)* where $Y = \{y_0\} \cup \{y_i\}$, $Z = \{z_i\}$
Delete_Domain(x,z)	*Add_Domain(x,y)*	*Change_Domain(x,y,z)* , *Generalize_Domain(x,y,z)*
Delete_Range(x,z)	*Add_Range(x,y)*	*Change_Range(x,y,z)*, *Generalize_Range(x,y,z)*, *Specialiaze_Range(x,y,z)*, *Change_To_Datatype_Property(x,y,z)*, *Change_To_Object_Property(x,y,z)*

Table 6. Look-up Table for Composite Changes

Using the tables, the implementation of $findPotentialBasicChanges$ and $findPotentialCompositeChanges$ is easy. When $findPotentialBasicChanges$ is called for a specific low-level change $t \in \Delta_i$ ($i = 1, 2$), we search the left column of Tables 4, 5 for low-level changes that fit the description of the low-level change under question (line 2). If such a line is found, we search Δ to find all the low-level changes that the middle column requires (lines 3-10). If all such changes are found, then the basic change in the right column of the respective line is (potentially) an initially detectable change, so it is included in the result (lines 11-14). Note that if all the changes in the middle and left column are found in Δ (i.e., all the changes in δ_1, δ_2), then there will be no remaining free variables in the respective line, so the change b that is included in the result ($potC$) contains no variables. At the end of the process, $potC$ is returned, which contains changes b for which $\delta_1(b) \subseteq \Delta_1, \delta_2(b) \subseteq \Delta_2$. In order to determine whether b is initially detectable, one more check will be required (by the main algorithm), namely to determine whether $\phi(b)$ is true. When $findPotentialCompositeChanges$ is called for a specific detected basic change in $basic_changes$, we search the left column of Table 6, for basic changes that fit the description of the basic change under question (line 2). If such a line is found, we search $basic_changes$ to find all the basic changes that the middle column requires (lines 3-10). If all such changes are found, then the composite change in the right column of the respective line is (potentially) a detectable change, so it is included in the result (lines 11-14). At the end of the process, $potC$ is returned, which contains changes c for which $\forall b \in \Sigma(c) : initially\ detectable$. In order to determine whether c is detectable, one more check will be required (by the main algorithm), namely to determine whether $\phi(c)$ is true.

Algorithm 1: findPotentialBasicChanges(t, Δ)

```
 1: potC := ∅
 2: for all lines in Tables 4, 5 whose left column equals t do
 3:     toCheck := set of low-level changes in the middle column of Tables 4, 5
 4:     flag := true
 5:     for all t_0 ∈ toCheck do
 6:         if t_0 does not appear in Δ then
 7:             flag := false
 8:             break
 9:         end if
10:     end for
11:     if flag = true then
12:         b := the basic change in the right column of the respective line in Tables 4, 5.
13:         potC := potC ∪ {b}
14:     end if
15: end for
16: return potC
```

Algorithm 2: findPotentialCompositeChanges($b, basic_changes$)

```
1:  potC := ∅
2:  for all lines in Table 6 whose left column equals b do
3:      toCheck := set of basic changes in the middle column of Table 6
4:      flag := true
5:      for all b_0 ∈ toCheck do
6:          if b_0 does not appear in basic_changes then
7:              flag := false
8:              break
9:          end if
10:     end for
11:     if flag = true then
12:         c := the composite change in the right column of the respective line in Table 6.
13:         potC := potC ∪ {c}
14:     end if
15: end for
16: return potC
```

Finally, the full change detection algorithm is shown below (Algorithm 3). The first step of the algorithm is to pick a low-level change (i.e., a triple in Δ_1 or Δ_2). Then $findPotentialBasicChanges$ is called (line 4), in order to perform the first "filtering" of the basic changes that need to be considered. The changes returned by $findPotentialBasicChanges$ are not necessarily detectable or initially detectable; each of them represents one basic change whose detection *could* be triggered by the existence of the low-level change selected in line 3. Determining whether such a basic change is initially detectable or not is done in line 6 by checking the conditions (this is done for all changes in $potC$, per line 5). Once such a change is found, no further checks need to be made, as we know (per Theorem 3) that no other initially detectable change could be associated with the low-level change under question. For this reason, we add the change to the set of (initially) detected basic changes $basic_changes$, we remove from the Δ (in line 8) all the low-level changes that were associated with the detected change (so they are no longer relevant for future detections), and stop the process of checking (line 9). Once we have iterated over the whole Δ, set $basic_changes$ will contain all the initially detected basic changes.

In order to detect composite changes we proceed as follows: First, we pick a basic change from $basic_changes$ (line 13) and call $findPotentialCompositeChanges$ in order to determine the composite changes that could potentially be detected. In the following, we need to check for which composite change in $potC$ the conditions are true (line 16). When such a change is encountered, we can add the change to the detected composite changes $composite_changes$ and remove from $basic_changes$ all the basic changes contained in its Σ set(line 18), as we know (per Theorem 3) that no other detectable change could be associated with the basic change under question. At the end of the algorithm, the composite and remaining basic changes (the ones that have not been subsumed) are returned. In our example, the composite change *Generalize_Domain(participants, Onset, Event)* will be returned since the conditions

for this operation are true. The basic changes *Delete_Domain(participants,Onset)* and *Add_Domain(participants, Event)* won't be returned because they are subsumed by the aforementioned composite change.

Algorithm 3: Change Detection Algorithm

```
1: basic_changes = ∅
2: composite_changes = ∅
3: for all low-level changes t do
4:    potC := findPotentialBasicChanges(t,Δ)
5:    for all b ∈ potC do
6:       if φ(b) = true then
7:          basic_changes := basic_changes ∪ {b}
8:          Δ1 := (Δ1 \ δ1(b)), Δ2 := (Δ2 \ δ2(b))
9:          break
10:      end if
11:   end for
12: end for
13: for all basic changes b ∈ basic_changes do
14:   potC := findPotentialCompositeChanges(b,basic_changes)
15:   for all c ∈ potC do
16:      if φ(c) = true then
17:         composite_changes := composite_changes ∪ {b}
18:         basic_changes := basic_changes \ Σ(c)
19:         break
20:      end if
21:   end for
22: end for
23: return basic_changes ∪ composite_changes
```

The next theorem shows that the presented algorithm is sound and complete with respect to $\mathcal{L}$:

Theorem 11. *A change $c \in \mathcal{L}$ will be returned by Algorithm 3 with input V_1, V_2 iff c is detectable between V_1, V_2.*

Proof. Set $C = \{c \in \mathcal{L} | c : detectable\}$, $A = \{c \in \mathcal{L} | c$ was returned by Algorithm 3 $\}$. Consider a change $c \in A$.
If c is a basic change, then by $findPotentialBasicChanges$ we know that $\delta_1(c) \subseteq \Delta_1, \delta_2(c) \subseteq \Delta_2$. Moreover, by line 6 of the main algorithm, we also know that $\phi(c)$ is true. So c is initially detectable. Since c is contained in the output of the algorithm we know by line 18 that it is not a subsumed basic change of any composite change. So, c is detectable as well and $c \in C$.
If c is a composite change, then by $findPotentialCompositeChanges$ we know that $\forall b \in \Sigma(c) : detectable(b) = true$. Moreover, by line 16 of the main algorithm, we also know that $\phi(c)$ is true. So c is initially detectable and detectable, so $c \in C$.
We conclude that $A \subseteq C$.

Now consider a change $c \in C$.
Suppose initially that c is a basic change. Since c is detectable, $\delta_1(c) \subseteq \Delta_1$ and $\delta_2(c) \subseteq \Delta_2$, so the handle of c will be in Δ. Due to Theorem 3, the handle of c will not be removed from Δ (in line 7), because there is no other detectable change that contains the same low-level change in its set of respective required added or deleted triples. So, the algorithm will eventually read the handle (in the for loop of line 2) and call $findPotentialBasicChanges$ with this triple. Since c is a detectable change, it will be returned by $findPotentialBasicChanges$, its conditions will be true and it won't be contained in any composite's change Σ set so it will also be returned by Algorithm 3.
If, on the other hand, c is a composite change, the same argumentation holds, the only difference being that the handle of c is an initially detectable basic change that has been previously detected. From Theorem 10 since c is detectable all subsumed basic changes of c are initially detectable. Moreover, Theorem 3 guarantees that the handle of c will not be removed from $basic_changes$ (in line 18), because there is no other detectable change that contains the same basic changes in its set of subsumed basic changes. Finally, as explained above, c will be checked for its conditions being true in line 16 and c will be detected.
Therefore, $C \subseteq A$ and the proof is complete.

The following theorem shows the complexity of the change detection algorithm; the result assumes that V_1, V_2 are sorted, so that some searches can be made faster:

Theorem 12. *The complexity of Algorithm 3 for input V_1, V_2 is $O(max\{N_1, N_2, N^2\})$, where N_i is the size (in triples) of V_i $(i = 1, 2)$ and N is the size of $\Delta(V_1, V_2)$.*

Proof. To begin with, let us calculate the cost of computing $\Delta(V_1, V_2)$. This can be made using a variation of the Merge algorithm, which will output the non-merged items. The Merge algorithm takes $O(N_1+N_2)$ which is equal to $O(max\{N_1, N_2\})$. Note that, using the Merge algorithm, the output (i.e., Δ) will be also sorted, a result that will be used in the following.
The look-up table used by $findPotentialBasicChanges$ has a constant size, so it takes $O(1)$ time to search it. For each matching low-level change (left column in Tables 4, 5, one has to make a full search of the Δ for the required low-level changes (middle column). Since we need to search for a constant number of low-level changes, the required time is $O(N)$ (in the worst-case), using hash tables. This search will determine the potential basic changes to be put in $potC$, per the right column of Tables 4, 5. Since the table is of constant size, the size of $potC$ returned will be $O(1)$ as well; therefore, computing $potC$ (line 4) takes $O(N)$ in total.

The same reasoning holds for the complexity of $findPotentialCompositeChanges$ where for the matching basic change in the left column one has to search the detected basic changes for the required basic changes in the middle column. The best way to do this search is to make a sequential scan of the entire $basic_changes$ each time for the basic change patterns under question. This costs $O(N)$ time again (in the worst case) since the number of detected basic changes cannot exceed the size of Δ. For each potentially detectable change in $potC$, we need to determine whether its conditions are true. The time required for this depends on the change considered. For basic changes (e.g., *Delete_Domain*), it takes $O(1)$ number of checks, whereas for com-

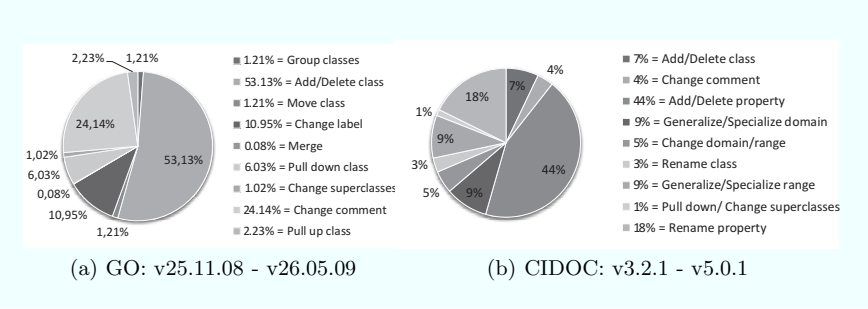
posite changes the cost is either $O(M)$ (e.g., *Delete_Class*) or $O(M^2)$ (e.g., *Reclassify_Individual_Higher*), where M is the number of basic changes in Σ of the respective composite change. Note that each individual check can be done in $O(1)$, a result which can be achieved using sophisticated labeling algorithms, as described in [3].
Now let us consider the worst-case scenario. The for loop (line 13) iterates over the basic changes. Let us consider the i-th iteration: for the selected change, we need $O(N)$ time for line 14, plus $O(1)$ iterations of $O(M_i^2)$ cost (lines 15-21), where M_i is the total size of $basicChng$ for the composite change considered. Then, the total cost (for the entire algorithm) is: $O(\sum_{i=1,\ldots,N}(N + M_i^2))$. However, note that: $\sum_{i=1,\ldots,N}(N + M_i^2) = N^2 + \sum_{i=1,\ldots,N} M_i^2 \leq N^2 + (\sum_{i=1,\ldots,N} M_i)^2$. The sum in the last equation cannot exceed the size of Δ by more than a constant factor (i.e., it is $O(N)$). Therefore, the complexity of the algorithm is $O(N^2)$.
We conclude that the total cost of computing the high-level changes is $O(max\{N_1, N_2, N^2\})$.

In practice, our algorithm will rarely exhibit the quadratic worst-case complexity described in Theorem 12. There are several reasons for that. First of all, the complexity of searching through Δ (in $findPotentialBasicChanges$) was calculated to be $O(N)$; for most changes, this will be $O(1)$ on average, due to the use of hash tables. Secondly, evaluating the conditions (line 6 for basic changes, line 16 for composite changes) varies from constant to quadratic over M_i, depending on the type of changes in $potC$. Furthermore, even though M_i may, in the worst case, be comparable to N, this will rarely be the case; therefore, even operations that exhibit quadratic complexity over M_i, will rarely exhibit quadratic complexity over N. The above arguments appear more emphatically for basic changes, as the cost of evaluating the conditions of any basic change is $O(1)$. The above observations will be verified by the results of our experiments (Section 5).

4 Operations Based on Heuristics

The detection semantics of the changes described so far used no heuristics or other approximation techniques, and were based on the implicit assumption that no terminological changes occurred between the RDF/S KBs. However, as described in Section 2, this is not always true. In Figure 1 for example, a matcher could identify that classes $Existing$ and $Persistent$ correspond to the same entity, so a *Rename_Class* operation should be detected (rather than the addition of a class and the deletion of another). Operations that capture terminological changes are different from the changes discussed so far, because they can only be detected using *matchers* [6], which employ various sophisticated, heuristic-based techniques for identifying elements with different names that correspond to the same real world entity.

For evaluation purposes, we implemented a simple matcher that associates elements based on the similarity of their "neighborhoods" that is: The sets of nodes and links that are pointing from/to the elements under question. If the similarity exceeds a certain threshold, then a matching is reported. More specifically, if an element in V_1 is matched with an element in V_2, we detect a *Rename* operation, whereas if it is matched with a set of elements in V_2, we detect a *Split* operation. On the other hand, if a set of elements in V_1 are matched with an element in V_2, a *Merge* operation is detected.



(a) GO: v25.11.08 - v26.05.09 (b) CIDOC: v3.2.1 - v5.0.1

Fig. 2. Overview of Composite and Heuristic Changes

Another case where matchers are necessary appears when an object is associated with a different comment in V_1, V_2, which could be either because the old comment was deleted and a new one was added, or because the old comment was edited. In this case, we use the Levenshtein [11] string distance metric which compares the similarity of the respective comments and determines whether a pair of *Delete_Comment-Add_Comment*, or a single *Change_Comment*, should be returned (similarly for labels).

It should be noted that once the matchings are calculated and the corresponding detected operations are reported as above, we continue with the normal, non-approximate change detection process (as described in Section 3). This means in practice that the detection of heuristic changes takes precedence over composite changes in the same way that the detection of composite changes takes precedence over basic changes. For example, in Figure 1, we would *not* report a *Change_Domain(started_on,Existing,Persistent)*, because $Existing$ and $Persistent$ are identified as the same class.

Note that the focus of this paper is *not* on developing a sophisticated matcher, but on change detection. Thus, our system was designed to be modular, in the sense that any custom-made or off-the-shelf matcher could be used to calculate the required matchings instead of the one described above. In addition, the user may choose to circumvent the matching process altogether by not defining a matcher. In this respect, the matching process can be viewed as an optional, pre-processing phase to the actual change detection algorithm, so it is an extension of the basic framework described in Section 3.

5 Experimental Evaluation

The evaluation of our approach was based on experiments performed on two well-established ontologies from the cultural (CIDOC [4]) and biological (GO [7]) domains. It aims at showing the intuitiveness and conciseness of the changes contained in $\mathcal{L}$ (Figure 2 and Table 7) as well as verifying that the performance of the implemented algorithm conforms to the average-case analysis of Section 3.2 (Table 8).

CIDOC consists of nearly 80 classes and 250 properties(no instances). For our experiments, we used the versions of CIDOC encoded in RDF, namely v3.2.1 (dated 02.2002), v3.3.2 (dated 10.2002), v3.4.9 (dated 12.2003), v4.2 (dated 06.2005) and

v5.0.1 (dated 04.2009), which are available in [4]. The detected changes apply mostly on properties, and many involve the heuristic change *Rename* (see Figure 2). For the detection of the heuristic changes a special-purpose matcher was developed, that exploited CIDOC's naming policy which attaches a unique, change preserving ID in the used names; this way, the accuracy (precision and recall) of the matchings for CIDOC was 100%. Every CIDOC version is accompanied by release notes describing in natural language the differences with the previous version. A very interesting observation is that the changes detected by our algorithm uncovered a number of typos, omissions and other mistakes in these release notes, which were verified by one of CIDOC's editors. These findings highlight the need for automated change detection algorithms, as they show that even the most careful manual recording process may introduce mistakes in the description of the changes between versions.

Our second case study involved the Gene Ontology (GO) [7], whose size and update rate make it one of the largest and most representative data sets for ontology evolution. The GO project has developed three ontologies that describe gene products in terms of their associated biological processes, cellular components and molecular functions in a species-independent manner. GO is composed of circa 28000 classes, that are all instances of one meta-class, and 1350 property instances of property $obsolete$ which is, sometimes, used by the GO editors to mark classes as obsolete as an alternative to deleting them. Although GO is provided in RDF/XML format the subsumption relationships between classes are represented by user-defined properties instead of the standard subClassOf property, so we used versions of GO released by the UniProt Consortium [1], which use RDFS semantics. Even though GO is updated on a daily basis, UniProt releases a new version every month and only the latest version is available for download[2]. During the time of our experiments we were able to retrieve 5 versions of GO (dated 25.11.08, 16.12.08, 24.03.09, 05.05.09 and 26.05.09). The detected heuristic changes (*Merge*) were very few (0.08% of the total) as shown in Figure 2; the string matcher, on the other hand, detected several *Change_Comment* and *Change_Label* operations. The rest of the changes were mostly additions and deletions of classes, as well as changes in the hierarchy. The detected basic changes (not pictured) included, among others, additions of property instances. Even though we weren't able to find any recent official documentation regarding the changes on GO, the changes reported by certain studies (e.g. [25]) show that the detected operations capture the intuition of the editors.

Table 7 shows the number of changes detected between different pairs of CIDOC and GO versions. The columns report the compared versions, their sizes (number of triples), the number of added/deleted triples (in Δ) and the number of changes recognized by our algorithm by disabling or enabling the detection of composite and heuristic changes. The number of detected basic changes is almost the same as the size of Δ, showing that deltas consisting entirely of basic changes are not concise. On the other hand, the number of reported changes is significantly reduced when composite and heuristic changes are considered; this reduction ranges from 44% to 78% for CIDOC and from 59% to 74% for GO.

Table 8 reports the running time of the detection algorithm. The times were measured on a Linux machine equipped with a Pentium 4 processor running at 3.4GHz

[2] ftp://ftp.uniprot.org/pub/databases/uniprot_datafiles_by_format/rdf/

Versions	$V1$	$V2$	Δ	Basic	Basic + Composite + Heuristic
CIDOC					
v3.2.1 - v3.3.2	952	1081	870	834	202+120+39 = 361
v3.3.2 - v3.4.9	1081	1110	287	285	13+15+34 = 62
v3.4.9 - v4.2	1110	1254	571	538	287+6+10 = 303
v4.2 - v5.0.1	1254	1318	339	327	44+51+52=147
GO					
v25.11.08 - v16.12.08	183430	184704	2979	2260	326+296+307 = 929
v16.12.08 - v24.03.09	184704	188268	7312	5053	745+706+440 = 1891
v24.03.09 - v05.05.09	188268	190097	3108	2322	359+362+97 = 818
v05.05.09 - v26.05.09	190097	191417	2663	1983	265+312+147 = 724

Table 7. Evaluation Results

and 1.5GB of main memory. The execution times for the detection of basic changes were, in general, linear to the input, instead of quadratic, verifying our average-case analysis in Section 3. With respect to composite changes, the execution time reveals some interesting anomalies. For example, comparing the results for versions v3.3.2-v3.4.9 and v3.4.9-v4.2 (for CIDOC) we see a reduction in the running time, despite the increase of the input size (cf. Table 7). This is due to the very small number of detected composite changes for the second input (see Table 7). Also, when comparing the results of v16.12.08-v24.03.09 to v25.11.08-v16.12.08 (GO) we see that the running time increases in a sub-linear fashion with respect to the input. This can be explained by considering the types of detected composite changes, which reveals that for versions v16.12.08-v24.03.09 the changes whose complexity for evaluating the conditions is quadratic are 4.5% of the total, whereas for v25.11.08-v16.12.08 such changes constitute 15% of the total. The slow execution times related to heuristic changes is due to the overhead caused by the employed matcher.

Versions	Δ	Basic Changes	Composite Changes	Heuristic Changes
CIDOC				
v3.2.1 - v3.3.2	95.91 ms	13.53 ms	3.35 ms	26.19 ms
v3.3.2 - v3.4.9	91.45 ms	3.94 ms	1.01 ms	5.54 ms
v3.4.9 - v4.2	95.75 ms	8.05 ms	0.26 ms	9.68 ms
v4.2 - v5.0.1	120.58 ms	5.50 ms	2.12 ms	861.77 ms
GO				
v25.11.08 - v16.12.08	35.214 s	133.79 ms	28.60 ms	45.195 s
v16.12.08 - v24.03.09	36.610 s	249.66 ms	39.65 ms	345.419 s
v24.03.09 - v05.05.09	36.684 s	146.20 ms	23.99 ms	38.006 s
v05.05.09 - v26.05.09	36.712 s	131.22 ms	24.45 ms	40.067 s

Table 8. Running Time

6 Related Work

In the literature, there are certain change detection algorithms that report low-level deltas (e.g., [23,24]), where a representation of changes based on the added and deleted triples is used. On the other hand, several approaches use high-level changes [15,17,22] which, like in our case, are usually distinguished in basic and composite.

In [10,14,15,18,19,21] authors describe several change operations and the intuition behind them. However, a formal definition of the semantics of such changes ([10,14,15,19]) or of the corresponding detection process ([15]) is usually missing, thus not allowing us to guarantee useful properties such as one low-level change triggering the detection of only one high-level change.

Authors in [10,14] describe an algorithm for detecting changes. This algorithm is implemented in PromptDiff, which is an extension of Protégé [8]. PromptDiff is a fixed-point algorithm that incorporates heuristic-based matchers in order to detect the changes that occurred between two versions. Therefore, the entire change detection process is heuristic-based, thereby introducing an uncertainty in the detection results: the evaluation reported by the authors showed that their algorithm had a recall of 96% and a precision of 93%. In our case, such metrics are not relevant, as our detection process does not use heuristics and any false positives or negatives will be artifacts of the matching process, not of the detection algorithm itself.

In [18] the Change Definition Language (CDL) is proposed as a means to define a language of high-level changes. In CDL, a change is defined and detected using temporal queries over a version log that contains recordings of the applied low-level changes. The version log is updated when a change occurs which overrules the use of this approach in non-curated or distributed environments. In our work, version logs are not necessary for the detection, as the low-level delta can be produced also *a posteriori*. Note also that in [18], changes that require heuristics for their detection (such as *Rename*) are completely ignored. This reduces the usefulness of the proposed language.

In our framework, changes that require heuristics are considered separately. This way, we can support operations that require heuristics, while maintaining determinism for the operations that don't need them. In addition, we have the option to ignore such changes, which may be useful for applications that require perfect precision and recall.

7 Conclusion and Future Work

The need for dynamic ontologies makes the automatic identification of deltas between versions increasingly important for several reasons (storing and communication efficiency, visualization of differences etc). Unfortunately, it is often difficult or impossible for curators or editors to accurately record such deltas without the use of automated tools; this was also evidenced by the mistakes found in the release notes of CIDOC.

In this paper, we addressed the problem of automatically identifying deltas. We proposed a formal framework and used it for defining a language of high-level changes for both schema and data, $\mathcal{L}$, and an algorithm that correctly detects changes from $\mathcal{L}$. We proved that $\mathcal{L}$ satisfies several intuitive properties (*Completeness*, *Non-ambiguity*, *Reversibility*). Note that the existence of other languages satisfying these properties is not

ruled out. However, if the intuitiveness of the changes is not taken into account, the languages will end up being artificial and without practical use in real-world scenarios. Thus, the intuitiveness of the changes that $\mathcal{L}$ contains was a critical factor in our design and experimental evidence on the usefulness of $\mathcal{L}$ was provided. The detection algorithm itself was shown to be quite efficient, namely of quadratic worst-case complexity (even though, in practice, it seems to exhibit linear average-case complexity). The approach can be extended to more expressive ontology languages but the details depend on the semantics of the language and must be determined. As future work, we plan to extend $\mathcal{L}$ by considering *complex changes*, which aggregate several composite changes together. Moreover, we plan to conduct empirical studies involving real users.

Acknowledgements

This work was partially supported by the EU project KP-Lab (FP6-2004-IST-4). A shorter version of this report has been published in ISWC-09 [16].

References

1. A. Bairoch, R. Apweiler, C.H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, et al. The Universal Protein Resource (UniProt). *Nucleic Acids Research*, 2005.
2. D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. www.w3.org/TR/2004/REC-rdf-schema-20040210, 2004.
3. V. Christophides, G. Karvounarakis, D. Plexousakis, M. Scholl, and S. Tourtounis. Optimizing taxonomic semantic web queries using labeling schemes. *Web Semantics: Science, Services and Agents on the WWW*, 2004.
4. CIDOC-CRM. http://cidoc.ics.forth.gr/official_release_cidoc.html.
5. R. Cloran and B. Irwin. Transmitting rdf graph deltas for a cheaper semantic web. In *Proc. of SATNAC*, 2005.
6. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag, 2007.
7. D.P. Hill, B. Smith, M.S. McAndrews-Hill, and J.A. Blake. Gene ontology annotations: What they mean and where they come from. *BMC Bioinformatics*, 2008.
8. http://protege.stanford.edu. Protege Project, 2002.
9. http://www.hozo.jp/. Hozo.
10. M. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Vrije Univ., 2004.
11. VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics-Doklady*, volume 10, 1966.
12. B. McBride, F. Manola, and E. Miller. Rdf primer. www.w3.org/TR/rdf-primer, 2004.
13. N.F. Noy, A. Chugh, W. Liu, and M.A. Musen. A Framework for Ontology Evolution in Collaborative Environments. In *Proc. of ISWC*, 2006.
14. N.F. Noy and M.A. Musen. PromptDiff: A Fixed-Point Algorithm for Comparing Ontology Versions. In *Proc. of AAAI*, 2002.
15. A. Palma, P. Haase, Y. Wang, and M. dAquin. D1.3.1 propagation models and strategies. Technical report, NeOn Deliverable D1.3.1, 2007.
16. V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. On Detecting High-Level Changes in RDF/S KBs. In *Proc. of ISWC*, 2009.
17. P. Plessers and O. De Troyer. Ontology Change Detection Using a Version Log. In *Proc. of ISWC*, 2005.

18. P. Plessers, O. De Troyer, and S. Casteleyn. Understanding Ontology Evolution: A Change Detection Approach. *Web Semantics: Science, Services and Agents on the WWW*, 2007.
19. D. Rogozan and G. Paquette. Managing Ontology Changes on the Semantic Web. In *Proc. of IEEE/WIC/ACM on Web Intelligence*, 2005.
20. G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *Proc. of ISWC*, 2005.
21. L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, Univ. of Karlsruhe, 2004.
22. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-Driven Ontology Evolution Management. In *Proc. of EKAW. Ontologies and the Semantic Web*, 2002.
23. M. Volkel, W. Winkler, Y. Sure, S. Ryszard Kruk, and M. Synak. Semversion: A versioning system for rdf and ontologies. In *Proc. of ESWC*, 2005.
24. D. Zeginis, Y. Tzitzikas, and V. Christophides. On the Foundations of Computing Deltas Between RDF Models. In *Proc. of ISWC+ ASWC*, 2007.
25. A.V. Zhdanova. Community-Driven Ontology Evolution: Gene Ontology Case Study. In *Proc. of BIS*, 2008.

A Basic changes

Changes on nodes

Change	***Add_Type_Class(a)***	***Delete_Type_Class(a)***
Intuition	Add object a of type class	Delete object a of type class
Arguments	a = The added object	a = The deleted object
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$
C^{-1}	*Delete_Type_Class(a)*	*Add_Type_Class(a)*

Change	***Add_Type_Metaclass(a)***	***Delete_Type_Metaclass(a)***
Intuition	Add object a of type metaclass	Delete object a of type metaclass
Arguments	a = The added object	a = The deleted object
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$
C^{-1}	*Delete_Type_Metaclass(a)*	*Add_Type_Metaclass(a)*

Change	***Add_Type_Metaproperty(a)***	***Delete_Type_Metaproperty(a)***
Intuition	Add object a of type metaproperty	Delete object a of type metaproperty
Arguments	a = The added object	a = The deleted object
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$
C^{-1}	*Delete_Type_Metaproperty(a)*	*Add_Type_Metaproperty(a)*

Change	***Add_Type_Property(a)***	***Delete_Type_Property(a)***
Intuition	Add object a of type property	Delete object a of type property
Arguments	a = The added object	a = The deleted object
δ_1	$(a, \mathsf{type}, \mathsf{property})$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{type}, \mathsf{property})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$
C^{-1}	*Delete_Type_Property(a)*	*Add_Type_Property(a)*

Change	***Add_Type_Individual(a)***	***Delete_Type_Individual(a)***
Intuition	Add object a of type individual	Delete object a of type individual
Arguments	a = The added object	a = The deleted object
δ_1	$(a, \mathsf{type}, \mathsf{resource})$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{type}, \mathsf{resource})$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2)$
C^{-1}	*Delete_Type_Individual(a)*	*Add_Type_Individual(a)*

Changes on links

Change	***Add_Superclass(a,b)***	***Delete_Superclass(a,b)***
Intuition	Parent b of class a is added	Parent b of class a is deleted
Arguments	a = The class b = The new parent	a = The class b = The old parent
δ_1	$(a, \mathsf{subClassOf}, b)$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{subClassOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $b \neq \mathsf{resource}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $b \neq \mathsf{resource}$
C^{-1}	*Delete_Superclass(a,b)*	*Add_Superclass(a,b)*

Change	***Add_SuperMetaclass(a,b)***	***Delete_SuperMetaclass(a,b)***
Intuition	Parent b of metaclass a is added	Parent b of metaclass a is deleted
Arguments	a = The metaclass b = The new parent	a = The metaclass b = The old parent
δ_1	$(a, \mathsf{subClassOf}, b)$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{subClassOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V2) \wedge$ $b \neq \mathsf{class}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V1) \wedge$ $b \neq \mathsf{class}$
C^{-1}	*Delete_SuperMetaclass(a,b)*	*Add_SuperMetaclass(a,b)*

Change	***Add_SuperMetaproperty(a,b)***	***Delete_SuperMetaproperty(a,b)***
Intuition	Parent b of metaproperty a is added	Parent b of metaproperty a is deleted
Arguments	a = The metaproperty b = The new parent	a = The metaproperty b = The old parent
δ_1	$(a, \mathsf{subClassOf}, b)$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{subClassOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V2) \wedge$ $b \neq \mathsf{property}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V1) \wedge$ $b \neq \mathsf{property}$
C^{-1}	*Delete_SuperMetaproperty(a,b)*	*Add_SuperMetaproperty(a,b)*

Change	***Add_Superproperty*(a,b)**	***Delete_Superproperty*(a,b)**
Intuition	Parent b of property a is added	Parent b of property a is deleted
Arguments	a = The metaproperty b = The new parent	a = The metaproperty b = The old parent
δ_1	$(a, \mathsf{subPropertyOf}, b)$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{subPropertyOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V2)$	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$-
C^{-1}	*Delete_SuperProperty(a,b)*	*Add_SuperProperty(a,b)*

Change	***Add_Type_To_Class*(a,b)**	***Delete_Type_From_Class*(a,b)**
Intuition	Type b of class a is added	Type b of class a is deleted
Arguments	a = The class b = The new type	a = The class b = The old type
δ_1	(a, type, b)	$\emptyset$
δ_2	$\emptyset$	(a, type, b)
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $b \neq \mathsf{class}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $b \neq \mathsf{class}$
C^{-1}	*Delete_Type_From_Class(a,b)*	*Add_Type_To_Class(a,b)*

Change	***Add_Type_To_Metaclass*(a,b)**	***Delete_Type_From_Metaclass*(a,b)**
Intuition	Type b of metaclass a is added	Type b of metaclass a is deleted
Arguments	a = The metaclass b = The new type	a = The metaclass b = The old type
δ_1	(a, type, b)	$\emptyset$
δ_2	$\emptyset$	(a, type, b)
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V2) \wedge$ $b \neq \mathsf{class}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V1) \wedge$ $b \neq \mathsf{class}$
C^{-1}	*Delete_Type_From_Metaclass(a,b)*	*Add_Type_To_Metaclass(a,b)*

Change	***Add_Type_To_Metaproperty***(a,b)	***Delete_Type_From_Metaproperty***(a,b)
Intuition	Type b of metaproperty a is added	Type b of metaproperty a is deleted
Arguments	a = The metaproperty b = The new type	a = The metaproperty b = The old type
δ_1	(a, type, b)	$\emptyset$
δ_2	$\emptyset$	(a, type, b)
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V2) \wedge$ $b \neq \mathsf{class}$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V1) \wedge$ $b \neq \mathsf{class}$
C^{-1}	*Delete_Type_From_Metaproperty*(a,b)	*Add_Type_To_Metaproperty*(a,b)

Change	***Add_Type_To_Property***(a,b)	***Delete_Type_From_Property***(a,b)
Intuition	Add a new type to property a	Delete an old type of property a
Arguments	a = The property to which a new type is added b = The new type of property a	a = The property from which an old type is deleted b = The deleted type of property a
δ_1	(a, type, b)	$\emptyset$
δ_2	$\emptyset$	(a, type, b)
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $b \neq \mathsf{property}$	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $b \neq \mathsf{property}$
C^{-1}	*Delete_Type_From_Property*(a,b)	*Add_Type_To_Property*(a,b)

Change	***Add_Type_To_Individual***(a,b)	***Delete_Type_From_Individual***(a,b)
Intuition	Add a new type to individual a	Delete an old type of individual a
Arguments	a = The individual to which a new type is added b = The new type of individual a	a = The individual from which an old type is deleted b = The deleted type of individual a
δ_1	(a, type, b)	$\emptyset$
δ_2	$\emptyset$	(a, type, b)
ϕ	$(a, \mathsf{type}, \mathsf{resource}) \in Cl(V2) \wedge$ $b \neq \mathsf{resource}$	$(a, \mathsf{type}, \mathsf{resource}) \in Cl(V1) \wedge$ $b \neq \mathsf{resource}$
C^{-1}	*Delete_Type_From_Individual*(a,b)	*Add_Type_To_Individual*(a,b)

Change	***Add_Comment(a,b)***	***Delete_Comment(a,b)***
Intuition	Comment b of object a is added	Comment b of object a is deleted
Arguments	a = The object b = The new comment	a = The object b = The old comment
δ_1	$(a, \mathsf{comment}, b)$	$\emptyset$
δ_2	$\emptyset$	$(a, \mathsf{comment}, b)$
ϕ	$((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V2))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V2)) \vee$ $((a, \mathsf{type}, \mathsf{class}) \in Cl(V2))$ $\vee ((a, \mathsf{type}, \mathsf{property}) \in Cl(V2))$	$((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V1))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V1)) \vee$ $((a, \mathsf{type}, \mathsf{class}) \in Cl(V1))$ $\vee ((a, \mathsf{type}, \mathsf{property}) \in Cl(V1))$
C^{-1}	*Delete_Comment(a,b)*	*Add_Comment(a,b)*

Change	***Add_Label(a,b)***	***Delete_Label(a,b)***
Intuition	Label b of object a is added	Label b of object a is deleted
Arguments	a = The object b = The new comment	a = The object b = The old comment
δ_1	(a, label, b)	$\emptyset$
δ_2	$\emptyset$	(a, label, b)
ϕ	$((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V2))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V2)) \vee$ $((a, \mathsf{type}, \mathsf{class}) \in Cl(V2))$ $\vee ((a, \mathsf{type}, \mathsf{property}) \in Cl(V2))$	$((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \in Cl(V1))$ $\vee \ ((a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \in Cl(V1)) \vee$ $((a, \mathsf{type}, \mathsf{class}) \in Cl(V1))$ $\vee ((a, \mathsf{type}, \mathsf{property}) \in Cl(V1))$
C^{-1}	*Delete_Label(a,b)*	*Add_Label(a,b)*

Change	***Add_Property_Instance(a_1,a_2,b)***	***Delete_Property_Instance(a_1,a_2,b)***
Intuition	Add property instance of property b	
Arguments	a_1 = The domain a_2 = The range b = The predicate of the property	a_1 = The domain a_2 = The range b = The predicate of the property
δ_1	(a_1, b, a_2)	$\emptyset$
δ_2	$\emptyset$	(a_1, b, a_2)
ϕ	$(b, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $b \neq \mathsf{subClassOf} \wedge$ $b \neq \mathsf{subPropertyOf} \wedge$ $b \neq \mathsf{type} \wedge$ $b \neq \mathsf{comment} \wedge$ $b \neq \mathsf{label} \wedge$ $b \neq \mathsf{domain} \wedge$ $b \neq \mathsf{range}$	$(b, \mathsf{type}, \mathsf{property}) \in Cl(V1) \ b \neq$ $\mathsf{subClassOf} \wedge$ $b \neq \mathsf{subPropertyOf} \wedge$ $b \neq \mathsf{type} \wedge$ $b \neq \mathsf{comment} \wedge$ $b \neq \mathsf{label} \wedge$ $b \neq \mathsf{domain} \wedge$ $b \neq \mathsf{range}$
C^{-1}	*Delete_Property_Instance(a_1,a_2,b)*	*Add_Property_Instance(a_1,a_2,b)*

Change	***Add_Domain(a,b)***	***Delete_Domain(a,b)***
Intuition	Domain b of property a is added	Domain b of property a is deleted
Arguments	a = The property b = The domain	a = The property b = The domain
δ_1	(a, domain, b)	$\emptyset$
δ_2	$\emptyset$	(a, domain, b)
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V2)$	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1)$
C^{-1}	*Delete_Domain(a,b)*	*Add_Domain(a,b)*

Change	***Add_Range(a,b)***	***Delete_Range(a,b)***
Intuition	Range b of property a is added	Range b of property a is deleted
Arguments	a = The property b = The range	a = The property b = The range
δ_1	(a, range, b)	$\emptyset$
δ_2	$\emptyset$	(a, range, b)
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V2)$	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1)$
C^{-1}	*Delete_Range(a,b)*	*Add_Range(a,b)*

Regarding retyping

Change	***Retype_Class_To_Metaclass(a)***	***Retype_Metaclass_To_Class(a)***
Intuition	Retype class a to a metaclass	Retype metaclass a to a class
Arguments	a = The class	a = The metaclass
δ_1	$(a, \mathsf{subClassOf}, \mathsf{class})$	$(a, \mathsf{subClassOf}, \mathsf{resource})$
δ_2	$(a, \mathsf{subClassOf}, \mathsf{resource})$	$(a, \mathsf{subClassOf}, \mathsf{class})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2)$
C^{-1}	*Retype_Metaclass_To_Class(a)*	*Retype_Class_To_Metaclass(a)*

Change	***Retype_Class_To_Metaproperty(a)***	***Retype_Metaproperty_To_Class(a)***
Intuition	Retype class a to a metaproperty	Retype metaproperty a to a class
Arguments	a = The class	a = The metaproperty
δ_1	$(a, \mathsf{subClassOf}, \mathsf{property})$	$(a, \mathsf{subClassOf}, \mathsf{resource})$
δ_2	$(a, \mathsf{subClassOf}, \mathsf{resource})$	$(a, \mathsf{subClassOf}, \mathsf{property})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2)$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2)$
C^{-1}	*Retype_Metaproperty_To_Class(a)*	*Retype_Class_To_Metaproperty(a)*

Change	***Retype_Class_To_Individual(a)***	***Retype_Individual_To_Class(a)***
Intuition	Retype class a to an individual	Retype individual a to a class
Arguments	a = The class	a = The individual
δ_1	$(a, \mathsf{type}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
δ_2	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{resource})$
ϕ	–	–
C^{-1}	*Retype_Individual_To_Class(a)*	*Retype_Class_To_Individual(a)*

Change	***Retype_Class_To_Property(a)***	***Retype_Property_To_Class(a)***
Intuition	Retype class a to a property	Retype property a to a class
Arguments	a = The class	a = The property
δ_1	$(a, \mathsf{type}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
δ_2	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{property})$
ϕ	–	–
C^{-1}	*Retype_Property_To_Class(a)*	*Retype_Class_To_Property(a)*

Change	***Retype_Property_To_Metaproperty(a)***	***Retype_Metaproperty_To_Property(a)***
Intuition	Retype property a to a metaproperty	Retype metaproperty a to a property
Arguments	a = The retyped object	
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{property})$
δ_2	$(a, \mathsf{type}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$
ϕ	–	–
C^{-1}	*Retype_Metaproperty_To_Property(a)*	*Retype_Property_To_Metaproperty(a)*

Change	***Retype_Property_To_Metaclass(a)***	***Retype_Metaclass_To_Property(a)***
Intuition	Retype property a to a metaclass	Retype metaclass a to a property
Arguments	a = The retyped object	
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$	$(a, \mathsf{type}, \mathsf{property})$
δ_2	$(a, \mathsf{type}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$
ϕ	–	–
C^{-1}	*Retype_Metaclass_To_Property(a)*	*Retype_Property_To_Metaclass(a)*

Change	***Retype_Property_To_Individual(a)***	***Retype_Individual_To_Property(a)***
Intuition	Retype property a to an individual	Retype individual a to a property
Arguments	a = The retyped object	
δ_1	$(a, \mathsf{type}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{property})$
δ_2	$(a, \mathsf{type}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{resource})$
ϕ	–	–
C^{-1}	*Retype_Individual_To_Property(a)*	*Retype_Property_To_Individual(a)*

Change	***Retype_Individual_To_Metaproperty(a)***	***Retype_Metaproperty_To_Individual(a)***
Intuition	Retype individual a to a metaproperty	Retype metaproperty a to an individual
Arguments	a = The retyped object	
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{resource})$
δ_2	$(a, \mathsf{type}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{property})$
ϕ	–	–
C^{-1}	*Retype_Metaproperty_To_Individual(a)*	*Retype_Individual_To_Metaproperty(a)*

Change	***Retype_Individual_To_Metaclass(a)***	***Retype_Metaclass_To_Individual(a)***
Intuition	Retype individual a to a metaclass	Retype metaclass a to an individual
Arguments	a = The retyped object	
δ_1	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$	$(a, \mathsf{type}, \mathsf{resource})$
δ_2	$(a, \mathsf{type}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{class})$
ϕ	–	–
C^{-1}	*Retype_Metaclass_To_Individual(a)*	*Retype_Individual_To_Metaclass(a)*

Change	***Retype_Metaproperty_To_Metaclass(a)***	***Retype_Metaclass_To_Metaproperty(a)***
Intuition	Retype metaproperty a to a metaclass	Retype metaclass a to a metaproperty
Arguments	a = The metaproperty	a = The metaclass
δ_1	$(a, \mathsf{subClassOf}, \mathsf{class})$	$(a, \mathsf{subClassOf}, \mathsf{property})$
δ_2	$(a, \mathsf{subClassOf}, \mathsf{property})$	$(a, \mathsf{subClassOf}, \mathsf{class})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2)$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2)$
C^{-1}	*Retype_Metaclass_To_Metaproperty(a)*	*Retype_Metaproperty_To_Metaclass(a)*

B Composite Changes

Change	***Add_Class***$(a, P_1, P_2, P_3, P_4, P_5, P_6)$	***Delete_Class***$(a, P_1, P_2, P_3, P_4, P_5, P_6)$
Intuition	Add class a with its neighborhood links	Delete class a with its neighborhood links
Arguments	P_1 = set of new parent classes of a, P_2 = set of classes that have as parent a, P_3 = set of new metaclasses of a, P_4 = set of new individuals that are type of a, P_5 = set of new comments of a, P_6 = set of new labels of a	P_1 = set of old parent classes of a, P_2 = set of classes that had as parent a, P_3 = set of old metaclasses of a, P_4 = set of individuals that were type of a, P_5 = set of old comments of a, P_6 = set of old labels of a
δ_1	$\forall p \in P_1 : (a, \mathsf{subClassOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subClassOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p \in P_4 : (a, \mathsf{type}, p)$, $\forall p \in P_5 : (a, \mathsf{comment}, p)$, $\forall p \in P_6 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$\emptyset$
δ_2	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{subClassOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subClassOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p \in P_4 : (a, \mathsf{type}, p)$, $\forall p \in P_5 : (a, \mathsf{comment}, p)$, $\forall p \in P_6 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$ $\forall p \in P_2$: $(p, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $\forall p \in P_4$: $(p, \mathsf{type}, \mathsf{resource}) \in Cl(V1)$	$(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$ $\forall p \in P_2$: $(p, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall p \in P_4$: $(p, \mathsf{type}, \mathsf{resource}) \in Cl(V2)$
$Subsumed$ $Basic\ changes$	*Add_Type_Class(a),* $\forall p \in P_1$: *Add_Superclass(a,p),* $\forall p \in P_2$: *Add_Superclass(p,a),* $\forall p \in P_3$: *Add_Type_To_Class(a,p),* $\forall p \in P_4$: *Add_Type_To_Class(p,a),* $\forall p \in P_5$: *Add_Comment(a,p),* $\forall p \in P_6$: *Add_Label(a,p)*	*Delete_Type_Class(a),* $\forall p \in P_1$: *Delete_Superclass(a,p),* $\forall p \in P_2$: *Delete_Superclass(p,a),* $\forall p \in P_3$: *Delete_Type_From_Class(a,p),* $\forall p \in P_4$: *Delete_Type_From_Class(p,a),* $\forall p \in P_5$: *Delete_Comment(a,p),* $\forall p \in P_6$: *Delete_Label(a,p)*
C^{-1}	Delete_Class$(a, P_1, P_2, P_3, P_4, P_5, P_6)$	Add_Class$(a, P_1, P_2, P_3, P_4, P_5, P_6)$

Table 9. The changes *Add_Metaclass* and *Add_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ being in δ_1 respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$. The changes *Delete_Metaclass* and *Delete_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ being in δ_2 respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$

Change	**Group_Classes(A,b)**	**Ungroup_Classes(A,b)**
Intuition	Group classes in A under b	Ungroup classes in A
Arguments	A = set of classes that have as new parent b, b = new parent class b	A = set of classes that had as parent b, b = old parent class b
δ_1	$\forall a \in A : (a, \mathsf{subClassOf}, b)$	$\forall a \in A : (a, \mathsf{subClassOf}, b)$
δ_2	$\emptyset$	$\forall a \in A$: *Delete_Superclass(a,b)*
ϕ	$\forall a \in A : (a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $\forall a \in A : (a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(b, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subClassOf}, x) \in Cl(V1)$ $\Rightarrow (a, \mathsf{subClassOf}, x) \in Cl(V2)) \wedge$ $\forall a \in A : (a, \mathsf{subClassOf}, b) \notin Cl(V1)$	$\forall a \in A : (a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $\forall a \in A : (a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(b, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subClassOf}, x) \in Cl(V1)$ $\Rightarrow (a, \mathsf{subClassOf}, x) \in Cl(V2)) \wedge$ $\forall a \in A : (a, \mathsf{subClassOf}, b) \notin Cl(V2)$
$Subsumed\ Basic\ changes$	$\forall a \in A$: *Add_Superclass(a,b)*	$\forall a \in A$: *Delete_Superclass(a,b)*
C^{-1}	Ungroup_Classes(A,b)	Group_Classes(A,b)

Table 10. The changes *Group_Metalasses* and *Group_Metaproperties* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$ and $(b, \mathsf{subClassOf}, \mathsf{resource})$. The changes *Ungroup_Metalasses* and *Ungroup_Metaproperties* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$ and $(b, \mathsf{subClassOf}, \mathsf{resource})$.

Change	***Pull_up_Class***(a,B,C)	**Pull_down_Class(a,**B,C**)**
Intuition	Move class a to a higher position in the subsumption hierarchy	Move a class to a lower position in the subsumption hierarchy
Arguments	B = set of old parents of a, C = set of new parents of a	B = set of old parents of a, C = set of new parents of a
δ_1	$\forall c \in C : (a, \mathsf{subClassOf}, c)$	$\forall c \in C : (a, \mathsf{subClassOf}, c)$
δ_2	$\forall b \in B : (a, \mathsf{subClassOf}, b)$	$\forall b \in B : (a, \mathsf{subClassOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B : (b, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall c \in C : (c, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(c, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subClassOf}, c)$ $\in Cl(V1) \wedge (b, \mathsf{subClassOf}, c) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subClassOf}, b)$ $\in Cl(V1) \wedge (c, \mathsf{subClassOf}, b) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Superclass(a,c),* $\forall b \in B$: *Delete_Superclass(a,b)*	$\forall c \in C$: *Add_Superclass(a,c),* $\forall b \in B$: *Delete_Superclass(a,b)*
C^{-1}	Pull_down_Class(a,C,B)	Pull_up_Class(a,C,B)

Table 11. The changes *Pull_up_Metaclass* and *Pull_up_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$. The changes *Pull_down_Metaclass* and *Pull_down_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$

Change	**Move_Class(a,**B,C**)**	**Change_Superclasses(a,**B,C**)**
Intuition	Move a class to a different subsumption hierarchy	Change the parents of class a
Arguments	B = set of old parents of a, C = set of new parents of a	B = set of old parents of a, C = set of new parents of a
δ_1	$\forall c \in C : (a, \mathsf{subClassOf}, c)$	$\forall c \in C : (a, \mathsf{subClassOf}, c)$
δ_2	$\forall b \in B : (a, \mathsf{subClassOf}, b)$	$\forall b \in B : (a, \mathsf{subClassOf}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subClassOf}, b)$ $\notin Cl(V1) \wedge (c, \mathsf{subClassOf}, b) \notin Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subClassOf}, c)$ $\notin Cl(V1) \wedge (b, \mathsf{subClassOf}, c) \notin Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $(\neg Cond$(*Pull_up_Class(a,B,C)* $\wedge$ $\neg Cond$(*Pull_down_Class(a,B,C)* $\wedge$ $\neg Cond$(*Move_Class(a,B,C)*) $\wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Superclass(a,c),* $\forall b \in B$: *Delete_Superclass(a,b)*	$\forall c \in C$: *Add_Superclass(a,c),* $\forall b \in B$: *Delete_Superclass(a,b)*
C^{-1}	Move_Class(a,C,B)	Change_Superclasses(a,C,B)

Table 12. The changes *Move_Metaclass* and *Move_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$. The changes *Change_SuperMetaclass* and *Change_SuperMetaproperties* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$.

Change	**Reclassify_Class_Higher(a,B,C)**	**Reclassify_Class_lower(a,B,C)**
Intuition	Reclassify a class to a higher position in the subsumption hierarchy	Reclassify a class to a lower position in the subsumption hierarchy
Arguments	B = set of old types of a, C = set of new types of a	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subClassOf}, c)$ $\in Cl(V1) \wedge (b, \mathsf{subClassOf}, c) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subClassOf}, b)$ $\in Cl(V1) \wedge (c, \mathsf{subClassOf}, b) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Class(a,c),* $\forall b \in B$: *Delete_Type_From_Class(a,b)*	$\forall c \in C$: *Add_Type_To_Class(a,c),* $\forall b \in B$: *Delete_Type_From_Class(a,b)*
C^{-1}	Reclassify_Class_lower(a,C,B)	Reclassify_Class_higher(a,C,B)

Table 13. The changes *Reclassify_Metaclass_Higher* and *Reclassify_Metaproperty_Higher* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$. The changes *Reclassify_Metaclass_Lower* and *Reclassify_Metaproperty_Lower* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$

Change	**Reclassify_Class(a,B,C)**
Intuition	Reclassify a class
Arguments	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \in Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ ($\neg Cond$(*Reclassify_Class_higher(a,B,C)* $\wedge$ $\neg Cond$(*Reclassify_Class_lower(a,B,C)*) $\wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Class(a,c),* $\forall b \in B$: *Delete_Type_From_Class(a,b)*
C^{-1}	Reclassify_Class(a,C,B)

Table 14. The changes *Reclassify_Metaclass* and *Reclassify_Metaproperty* are defined similarily with the exception of $(a, \mathsf{subClassOf}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{class})$, $(c, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$, $(b, \mathsf{subClassOf}, \mathsf{property})$, $(c, \mathsf{subClassOf}, \mathsf{property})$ holding for ϕ respectively instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$, $(c, \mathsf{subClassOf}, \mathsf{resource})$.

Change	**Add_Property(a,P_1,P_2,P_3,P_4,p_5,p_6,P_7,P_8)**	**Delete_Property(a,P_1,P_2,P_3,P_4,p_5,p_6,P_7,P_8)**
Intuition	Add property a with its neighborhood links	Delete property a with its neighborhood links
Arguments	P_1 = set of new parent properties of a, P_2 = set of properties that have as parent a, P_3 = set of new metaproperties of a, P_4 = set of new property instances of a, p_5 = the new domain of a, p_6 = the new range of a, P_7 = set of new comments of a, P_8 = set of new labels of a	P_1 = set of old parent classes of a, P_2 = set of classes that had as parent a, P_3 = set of old metaclasses of a, P_4 = set of old property instances of a, p_5 = the old domain of a, p_6 = the old range of a, P_7 = set of old comments of a, P_8 = set of old labels of a
δ_1	$\forall p \in P_1 : (a, \mathsf{subPropertyOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subPropertyOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p_1, p_2 \in P_4 : (p_1, a, p_2)$, $(a, \mathsf{domain}, p_5)$, (a, range, p_6), $\forall p \in P_7 : (a, \mathsf{comment}, p)$, $\forall p \in P_8 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{property})$	$\emptyset$
δ_2	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{subPropertyOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subPropertyOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p_1, p_2 \in P_4 : (p_1, a, p_2)$, $\forall p \in P_7 : (a, \mathsf{comment}, p)$, $\forall p \in P_8 : (a, \mathsf{label}, p)$, $(a, \mathsf{domain}, p_5)$, (a, range, p_6), $(a, \mathsf{type}, \mathsf{property})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V1)$ $\forall p \in P_2$: $(p, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $\forall p_1, p_2 \in P_4$: $(p_1, a, p_2) \in V1$	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \notin Cl(V2)$
$Subsumed$ $Basic\ changes$	*Add_Type_Property(a),* $\forall p \in P_1$: *Add_Superproperty(a,p),* $\forall p \in P_2$:*Add_Superproperty(p,a),* $\forall p \in P_3$: *Add_Type_To_Property(a,p),* $\forall p_1, p_2 \in P_4$: *Add_Property_Instance(p_1,p_2,a),* *Add_Domain(a,p_5),* *Add_Range(a,p_6),* $\forall p \in P_7$: *Add_Comment(a,p),* $\forall p \in P_8$: *Add_Label(a,p)*	*Delete_Type_Property(a),* $\forall p \in P_1$: *Delete_Superproperty(a,p),* $\forall p \in P_2$: *Delete_Superproperty(p,a),* $\forall p \in P_3$: *Delete_Type_From_Property(a,p),* $\forall p_1, p_2 \in P_4$:*Delete_Property_Instance(p_1,p_2,a),* *Delete_Domain(a,p_5),* *Delete_Range(a,p_6),* $\forall p \in P_7$: *Delete_Comment(a,p),* $\forall p \in P_8$: *Delete_Label(a,p)*
C^{-1}	Delete_Property(a,P_1,P_2,P_3,P_4,p_5,p_6,P_7,P_8)	Add_Property(a,P_1,P_2,P_3,P_4,p_5,p_6,P_7,P_8)

Change	**Group_Properties_Under(A,b)**	**Ungroup_Properties_Under(A,b)**
Intuition	Group properties in A under b	Ungroup properties in A under b
Arguments	A = set of properties that have as new parent b, b = new parent property b	A = set of properties that had as parent b, b = the old parent property b
δ_1	$\forall a \in A : (a, \mathsf{subPropertyOf}, b)$	$\emptyset$
δ_2	$\emptyset$	$\forall a \in A : (a, \mathsf{subPropertyOf}, b)$
ϕ	$\forall a \in A : (a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $\forall a \in A : (a, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$, $\forall a \in A, \forall x : (a, \mathsf{subPropertyOf}, x) \in Cl(V1) \Rightarrow (a, \mathsf{subPropertyOf}, x) \in Cl(V2)) \wedge$ $\forall a \in A : (a, \mathsf{subPropertyOf}, b) \notin Cl(V1)$	$\forall a \in A : (a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $\forall a \in A : (a, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(b, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$, $\forall a \in A, \forall x : (a, \mathsf{subPropertyOf}, x) \in Cl(V1) \Rightarrow (a, \mathsf{subPropertyOf}, x) \in Cl(V2)) \wedge$ $\forall a \in A : (a, \mathsf{subPropertyOf}, b) \notin Cl(V2)$
$Subsumed$ $Basic\ changes$	$\forall a \in A$: *Add_Superproperty(a,b)*	$\forall a \in A$: *Delete_Superproperty(a,b)*
C^{-1}	Ungroup_Properties_Under(A,b)	Group_Properties_Under$A$,b)

Change	***Pull_up_Property***(a,B,C)	***Pull_down_Property***(a,B,C)
Intuition	Move property a to a higher position in the subsumption hierarchy	Move property a to a lower position in the subsumption hierarchy
Arguments	B = set of old parents of a, C = set of new parents of a	B = set of old parents of a, C = set of new parents of a
δ_1	$\forall c \in C : (a, \mathsf{subPropertyOf}, c)$	$\forall c \in C : (a, \mathsf{subPropertyOf}, c)$
δ_2	$\forall b \in B : (a, \mathsf{subPropertyOf}, b)$	$\forall b \in B : (a, \mathsf{subPropertyOf}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subPropertyOf}, c) \in Cl(V1) \wedge (b, \mathsf{subPropertyOf}, c) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subPropertyOf}, b) \in Cl(V1) \wedge (c, \mathsf{subPropertyOf}, b) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Superproperty(a,c),* $\forall b \in B$: *Delete_Superproperty(a,b)*	$\forall c \in C$: *Add_Superproperty(a,c),* $\forall b \in B$: *Delete_Superproperty(a,b)*
C^{-1}	Pull_down_Property(a,C,B)	Pull_up_Property(a,C,B)

Change	***Move_Property***(a,B,C)	***Change_Superproperties***(a,B,C)
Intuition	Move property a to a different subsumption hierarchy	Change the parents of property a
Arguments	B = set of old parents of a, C = set of new parents of a	B = set of old parents of a, C = set of new parents of a
δ_1	$\forall c \in C : (a, \mathsf{subClassOf}, c)$	$\forall c \in C : (a, \mathsf{subClassOf}, c)$
δ_2	$\forall b \in B : (a, \mathsf{subClassOf}, b)$	$\forall b \in B : (a, \mathsf{subClassOf}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subPropertyOf}, b) \notin Cl(V1) \wedge (c, \mathsf{subPropertyOf}, b) \notin Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subPropertyOf}, c) \notin Cl(V1) \wedge (b, \mathsf{subPropertyOf}, c) \notin Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(\neg Cond$(*Pull_up_Property(a,B,C)* $\wedge$ $\neg Cond$(*Pull_down_Property(a,B,C)* $\wedge$ $\neg Cond$(*Move_Property(a,B,C)*$) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Superproperty(a,c),* $\forall b \in B$: *Delete_Superproperty(a,b)*	$\forall c \in C$: *Add_Superproperty(a,c),* $\forall b \in B$: *Delete_Superproperty(a,b)*
C^{-1}	Move_Property(a,C,B)	Change_Superproperties(a,C,B)

Change	***Reclassify_Property_higher***(a,B,C)	***Reclassify_Property_lower***(a,B,C)
Intuition	Reclassify property a to a higher position in the subsumption hierarchy	Reclassify property a to a lower position in the subsumption hierarchy
Arguments	B = set of old types of a, C = set of new types of a	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subPropertyOf}, c) \in Cl(V1) \wedge (b, \mathsf{subPropertyOf}, c) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subPropertyOf}, b) \in Cl(V1) \wedge (c, \mathsf{subPropertyOf}, b) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Property(a,c),* $\forall b \in B$: *Delete_Type_From_Property(a,b)*	$\forall c \in C$: *Add_Type_To_Property(a,c),* $\forall b \in B$: *Delete_Type_From_Property(a,b)*
C^{-1}	Reclassify_Property_lower(a,C,B)	Reclassify_Property_higher(a,C,B)

Change	***Reclassify_Property(*a*,*B*,*C*)***
Intuition	Reclassify property a
Arguments	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ ($\neg Cond$(*Reclassify_Property_higher(*a*,*B*,*C*)*) $\wedge$ $\neg Cond$(*Reclassify_Property_lower(*a*,*B*,*C*)*)) $\wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Property(a,c),* $\forall b \in B$: *Delete_Type_From_Property(a,b)*
C^{-1}	Reclassify_Property(a,C,B)

Change	***Change_To_Datatype_Property(*a*,*b*,*c*)***	***Change_To_Object_Property(*a*,*b*,*c*)***
Intuition	Change the range of property a to a datatype	Change the range of property a to an object
Arguments	b = old range of a, c = new range of a	b = old range of a, c = new range of a
δ_1	$\forall c \in C : (a, \mathsf{range}, c)$	$\forall c \in C : (a, \mathsf{range}, c)$
δ_2	$\forall b \in B : (a, \mathsf{range}, b)$	$\forall b \in B : (a, \mathsf{range}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{range}, b) \in V1 \wedge$ $(b, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(b, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{range}, c) \in V2 \wedge$ $(c, \mathsf{type}, rdfs : Literal) \in V2$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{range}, b) \in V1 \wedge$ $(c, \mathsf{type}, \mathsf{class}) \in Cl(V1) \wedge$ $(c, \mathsf{subClassOf}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{range}, c) \in V2 \wedge$ $(b, \mathsf{type}, rdfs : Literal) \in V2$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Range(a,c),* $\forall b \in B$: *Delete_Range(a,b)*	$\forall c \in C$: *Add_Range(a,c),* $\forall b \in B$: *Delete_Range(a,b)*
C^{-1}	Change_To_Object_Property(a,c,b)	Change_To_Datatype_Property(a,c,b)

Change	***Specialize_Range***(a,b,c)	***Generalize_Range***(a,b,c)
Intuition	Change the range of property a to a subClass of it	Change the range of property a to a superClass of it
Arguments	b = old range of a, c = new range of a	b = old range of a, c = new range of a
δ_1	$\forall c \in C : (a, \mathsf{range}, c)$	$\forall c \in C : (a, \mathsf{range}, c)$
δ_2	$\forall b \in B : (a, \mathsf{range}, b)$	$\forall b \in B : (a, \mathsf{range}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{range}, b) \in V1 \wedge$ $(a, \mathsf{range}, c) \in V2 \wedge$ $(c, \mathsf{subClassOf}, b) \in Cl(V1) \wedge$ $(c, \mathsf{subClassOf}, b) \in Cl(V2)$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{range}, b) \in V1 \wedge$ $(a, \mathsf{range}, c) \in V2 \wedge$ $(b, \mathsf{subClassOf}, c) \in Cl(V1) \wedge$ $(b, \mathsf{subClassOf}, c) \in Cl(V2)$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Range(a,c)*, $\forall b \in B$: *Delete_Range(a,b)*	$\forall c \in C$: *Add_Range(a,c)*, $\forall b \in B$: *Delete_Range(a,b)*
C^{-1}	Generalize_Range(a,c,b)	Specialize_Range(a,c,b)

Change	***Change_Range***(a,b,c)
Intuition	Change the range of property a.
Arguments	b = old range of a, c = new range of a
δ_1	$\forall c \in C : (a, \mathsf{range}, c)$
δ_2	$\forall b \in B : (a, \mathsf{range}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{range}, b) \in V1 \wedge$ $(a, \mathsf{range}, c) \in V2 \wedge$ $\neg Cond($*Change_To_Datatype_Property(a,b,c)*$) \wedge$ $\neg Cond($*Change_To_Object_Property(a,b,c)*$) \wedge$ $\neg Cond($*Specialize_Range(a,b,c)*$) \wedge$ $\neg Cond($*Generalize_Range(a,b,c)*$)$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Range(a,c)*, $\forall b \in B$: *Delete_Range(a,b)*
C^{-1}	Change_Range(a,c,b)

Change	***Specialize_Domain***(a,b,c)	***Generalize_Domain***(a,b,c)
Intuition	Change the domain of property a to a subClass of it	Change the domain of property a to a superClass of it
Arguments	b = old domain of a, c = new domain of a	b = old domain of a, c = new domain of a
δ_1	$\forall c \in C : (a, \mathsf{domain}, c)$	$\forall c \in C : (a, \mathsf{domain}, c)$
δ_2	$\forall b \in B : (a, \mathsf{domain}, b)$	$\forall b \in B : (a, \mathsf{domain}, b)$
ϕ	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{domain}, b) \in V1 \wedge$ $(a, \mathsf{domain}, c) \in V2 \wedge$ $(c, \mathsf{subClassOf}, b) \in Cl(V1) \wedge$ $(c, \mathsf{subClassOf}, b) \in Cl(V2)$	$(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(\mathsf{a}, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{domain}, b) \in V1 \wedge$ $(a, \mathsf{domain}, c) \in V2 \wedge$ $(b, \mathsf{subClassOf}, c) \in Cl(V1) \wedge$ $(b, \mathsf{subClassOf}, c) \in Cl(V2)$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Domain(a,c)*, $\forall b \in B$: *Delete_Domain(a,b)*	$\forall c \in C$: *Add_Domain(a,c)*, $\forall b \in B$: *Delete_Domain(a,b)*
C^{-1}	Specialize_Domain(a,c,b)	Generalize_Domain(a,c,b)

Change	***Change_Domain(a,b,c)***
Intuition	Change the domain of property a.
Arguments	b = old domain of a, c = new domain of a
δ_1	$\forall c \in C : (a, \mathsf{domain}, c)$
δ_2	$\forall b \in B : (a, \mathsf{domain}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{property}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \in Cl(V2) \wedge$ $(a, \mathsf{domain}, b) \in V1 \wedge$ $(a, \mathsf{domain}, c) \in V2 \wedge$ $\neg Cond$(*Specialize_Domain(a,b,c)*)$\wedge$ $\neg Cond$(*Generalize_Domain(a,b,c)*)
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Domain(a,c)*, $\forall b \in B$: *Delete_Domain(a,b)*
C^{-1}	Change_Domain(a,c,b)

Change	**Add_Individual(a,P_1,P_2,P_3)**	**Delete_Individual(a,P_1,P_2,P_3)**
Intuition	Add individual a with its neighborhood links	Delete individual a with its neighborhood links
Arguments	P_1 = set of new classes of a, P_2 = set of new comments of a, P_3 = set of new labels of a	P_1 = set of old classes of a, P_2 = set of old comments of a, P_3 = set of old labels of a
δ_1	$\forall p \in P_1 : (a, \mathsf{type}, p)$, $\forall p \in P_2 : (a, \mathsf{comment}, p)$, $\forall p \in P_3 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{resource})$	$\emptyset$
δ_2	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{type}, p)$, $\forall p \in P_2 : (a, \mathsf{comment}, p)$, $\forall p \in P_3 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{resource})$
ϕ	$(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V1) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V1)$	$(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{resource}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{property}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{type}, \mathsf{class}) \notin Cl(V2) \wedge$ $(a, \mathsf{subClassOf}, \mathsf{property}) \notin Cl(V2)$
$Subsumed$ $Basic\ changes$	*Add_Type_Property(a)*, $\forall p \in P_1$: *Add_Type_To_Individual(a,p)*, $\forall p \in P_2$: *Add_Comment(a,p)*, $\forall p \in P_3$: *Add_Label(a,p)*	*Delete_Type_Property(a)*, $\forall p \in P_1$: *Delete_Type_From_Individual(a,p)*, $\forall p \in P_2$: *Delete_Comment(a,p)*, $\forall p \in P_3$: *Delete_Label(a,p)*
C^{-1}	Delete_Individual(a,P_1,P_2,P_3)	Add_Individual(a,P_1,P_2,P_3)

Change	**Reclassify_Individual(a,B,C)**
Intuition	Reclassify an individual
Arguments	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \in Cl(V2) \wedge$ ($\neg Cond$(*Reclassify_Individual_higher(a,B,C)* $\wedge$ $\neg Cond$(*Reclassify_Individual_lower(a,B,C)*) $\wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Individual(a,c)*, $\forall b \in B$: *Delete_Type_From_Individual(a,b)*
C^{-1}	Reclassify_Individual(a,C,B)

Change	**Reclassify_Individual_higher(a,B,C)**	**Reclassify_Individual_lower(a,B,C)**
Intuition	Reclassify an individual under a class that is at a higher position in the subsumption hierarchy	Reclassify an individual under a class that is at a lower position in the subsumption hierarchy
Arguments	B = set of old types of a, C = set of new types of a	B = set of old types of of a, C = set of new types of of a
δ_1	$\forall c \in C : (a, \mathsf{type}, c)$	$\forall c \in C : (a, \mathsf{type}, c)$
δ_2	$\forall b \in B : (a, \mathsf{type}, b)$	$\forall b \in B : (a, \mathsf{type}, b)$
ϕ	$(a, \mathsf{type}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (b, \mathsf{subClassOf}, c) \in$ $Cl(V1) \wedge (b, \mathsf{subClassOf}, c) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$	$(a, \mathsf{type}, \mathsf{resource}) \in Cl(V1) \wedge$ $(a, \mathsf{type}, \mathsf{resource}) \in Cl(V2) \wedge$ $\forall b \in B, \forall c \in C : (c, \mathsf{subClassOf}, b) \in$ $Cl(V1) \wedge (c, \mathsf{subClassOf}, b) \in Cl(V2) \wedge$ $B \neq \emptyset \wedge$ $C \neq \emptyset$
$Subsumed$ $Basic\ changes$	$\forall c \in C$: *Add_Type_To_Individual(a,c),* $\forall b \in B$: *Delete_Type_From_Individual(a,b)*	$\forall c \in C$: *Add_Type_To_Individual(a,c),* $\forall b \in B$: *Delete_Type_From_Individual(a,b)*
C^{-1}	Reclassify_Individual_lower(a,C,B)	Reclassify_Individual_higher(a,C,B)