A

High-Level Change Detection in RDF(S) KBs

Vicky Papavasileiou, University of Crete and FORTH-ICS, Greece
Giorgos Flouris, FORTH-ICS, Greece
Irini Fundulaki, FORTH-ICS, Greece
Dimitris Kotzinos, TEI of Serres and FORTH-ICS, Greece
Vassilis Christophides, University of Crete and FORTH-ICS, Greece

With the increasing use of Web 2.0 to create, disseminate and consume large volumes of data, more and more information is published and becomes available for potential data consumers, i.e., applications/services, individual users and communities, outside their production site. The most representative example of this trend is Linked Open Data (LOD), a set of interlinked data and knowledge bases. The main challenge in this context is data governance within loosely-coordinated organizations that are publishing added-value interlinked data on the Web, bringing together issues related to data management and data quality, in order to support the full lifecycle of data production, consumption and management. In this paper, we are interested in curation issues for RDF(S) data, which is the default data model for LOD. In particular, we are addressing change management for RDF(S) data maintained by large communities (scientists, librarians, etc.) which act as curators to ensure high quality of data. Such curated Knowledge Bases (KBs) are constantly evolving for various reasons, such as the inclusion of new experimental evidence or observations, or the correction of erroneous conceptualizations. Managing such changes poses several research problems, including the problem of detecting the changes (delta) between versions of the same KB developed and maintained by different groups of curators, a crucial task for assisting them in understanding the involved changes. This becomes all the more important, as curated KBs are interconnected (through copying or referencing) and thus changes need to be propagated from one KB to another either within or across communities. This paper addresses this problem by proposing a change language which allows the formulation of *concise* and *intuitive* deltas. The language is expressive enough to *describe unambiguously* any possible change encountered in curated KBs expressed in RDF(S), and can be *efficiently and deterministically detected* in an automated way. Moreover, we devise a change detection *algorithm* which is sound and complete with respect to the aforementioned language, and study appropriate *semantics for executing the deltas* expressed in our language in order to move backwards and forwards in a multi-version repository, using only the corresponding deltas. Finally, we evaluate through experiments the effectiveness and efficiency of our algorithms using real ontologies from the cultural, bioinformatics and entertainment domains.

Categories and Subject Descriptors:
H.2.8 [**Information Systems**]: Database Management—*Database Applications*;

This work was partially supported by the EU-funded project DIACHRON.
Author's addresses: V. Papavassileiou, University of Crete; G. Flouris and I. Fundulaki, FORTH-ICS; D. Kotzinos, TEI of Serres; V. Christophides FORTH-ICS.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

H.2.3 [**Information Systems**]: Database Management—*Languages*;
I.2.4 [**Computing Methodologies**]: Artificial Intelligence—*Knowledge Representation Formalisms and Methods*

General Terms: Languages, Management

Additional Key Words and Phrases: Change Detection, Change Management, Delta, Diff, High-Level Changes, RDF(S), Deterministic Change Detection

ACM Reference Format:
Papavasileiou, V., Flouris, G., Fundulaki, I., Kotzinos, D., and Christophides, V. 2012. High-Level Change Detection in RDF(S) KBs. ACM Trans. Datab. Syst. V, N, Article A (January YYYY), 66 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Web 2.0 has completely changed the way we create, disseminate and consume large volumes of information. More and more corporate, government, or even user-generated data are published and become available for potential data consumers outside their production site, i.e., applications/services, individual users and communities. As a matter of fact, various RDF Knowledge Bases (KBs) from Wikipedia[1], US Census[2], CIA World Factbook[3], open government sites in the USA and the UK[4], museums (e.g., British Museum[5]) and memory institutions (e.g., Europeanna[6]), bibliographic (e.g., DBLP[7]), news (e.g., BBC[8]) and entertainment[9] sources, have been created and published online on the so-called Web of Data. In addition, numerous vocabularies and conceptual schemas in e-science are published nowadays as Semantic Web (SW) [Berners-Lee et al. 2001; Shadbolt et al. 2006; Bizer et al. 2009] ontologies (in RDF(S) [McBride et al. 2004], [Brickley and Guha 2004] or OWL [Horrocks et al. 2003]), most notably in life sciences[10] and environmental or earth sciences[11]. The publication of such data aim at facilitating community annotation and interlinkage of both scientific and scholarly data of interest. Moreover, Linked Open Data (LOD)[12] form essentially a global space of shared data on the Web which can be exploited by a variety of applications and tools, using adequate mechanisms [Schmedding 2011; Hartig et al. 2009].

The main challenge in this emerging space is *data governance* within loosely-coordinated organizations that are publishing added-value interlinked data on the Web. This challenge essentially brings together *data management* along with *data quality* issues and asks for techniques for supporting the *full lifecycle* of data produced and consumed on the Web of Data: from data extraction, transformation and integration, to monitoring, quality assessment and repair, until long-term preservation. In this paper we are interested in curation issues in the lifecycle of data published in RDF(S), which is the *lingua franca* of LOD. In particular, we are addressing *change management* for RDF(S) data maintained by large communities (e.g., scientists, librarians etc.) which act as curators to ensure the high quality of LOD. Such *curated KBs* [Buneman 2008] are constantly evolving to reflect an updated community under-

[1] dbpedia.org
[2] rdfabout.com/demo/census
[3] ckan.net/dataset/cia-world-factbook
[4] usa.gov/Topics/Reference_Shelf/Data.shtml, data.gov.uk
[5] collection.britishmuseum.org
[6] version1.europeana.eu/web/lod/data
[7] ckan.net/dataset/l3s-dblp
[8] www.bbc.co.uk/ontologies
[9] my.opera.com/community/sparql
[10] www.geneontology.org, www.biopax.org, www.nlm.nih.gov/research/umls, www.co-ode.org/galen
[11] sweet.jpl.nasa.gov/ontology
[12] linkeddata.org

standing of the domain or phenomena under investigation. They also evolve as new experimental evidence and observations are acquired, due to revisions in the initially intended usage of the curated KBs (e.g., narrowing or broadening their application scope) or even due to corrections of erroneous conceptualizations actually employed. Curated KBs are interconnected (through copying or referencing) and thus changes need to be propagated from one KB to another either within or across communities.

Clearly, there is a need for tools assisting curators in understanding and managing the changes of evolving KBs. In particular, managing the differences (*deltas*) of KBs has been proved to play a crucial role in various curation tasks such as the synchronization of autonomously developed KB versions [Cloran and Irwin 2005], or the visualization of the evolution history of a KB [Noy et al. 2006]. Change detection has attracted in the past the interest of both industry and academia and has been studied, along with the progress on data modeling, in object-oriented ([Skarra and Zdonik 1986], [Banerjee et al. 1987], [Nguyen and Rieu 1989], [Peters and Ozsu 1997]), hierarchical ([Chawathe et al. 1996], [Chawathe and Garcia-Molina 1997], [Lerner 2000], [Curino et al. 2008]) and semi-structured/XML data ([Cobena et al. 2001], [Marian et al. 2001]).

Note that RDF(S) graphs have labels on nodes and edges, in contrast to node-labeled XML trees and flat relations of the relational data model. Also, the emergence of properties as first-class citizens of the RDF(S) model and the use of hierarchies on them makes the RDF(S) data model different from object-oriented data models. As a result, none of the works on change detection in such models can be directly employed to automatically detect changes of RDF(S) KBs in order to assist curators in the time-consuming and error-prone task of documenting and exchanging deltas between RDF(S) KBs.

Motivated by this fact, in this work we address the problem of *change detection* in RDF(S) KBs. A change detection tool is essentially based on a *language of changes*, which describes the meaning of the different change operations that the underlying algorithm understands and detects. An important requirement for change detection tools is their ability to produce deltas that can be interpreted both by humans and machines. The first requirement (*human-interpretability*) implies that the corresponding language of changes should contain changes that are *intuitive*, *concise*, *close to the perception* of curators and which *capture as accurately as possible* the intent of a performed change [Klein 2004; Stojanovic 2004]. The second requirement (*machine-interpretability*) calls for a language which can describe *any possible change* that a curator could perform, in a *unique* and *deterministic* manner, so that each (arbitrarily complex) change corresponds to (is associated with) exactly one delta (set of changes) from the language; in addition, each change should have *well-defined detection semantics*, to allow the development of a corresponding detection algorithm, and each delta should have *well-defined application semantics*, allowing its subsequent execution (application) on a version.

In its simplest form, a language of changes consists of only two *low-level* operations, *Add(x)* and *Delete(x)*, which determine individual constructs that were added or deleted [Volkel et al. 2005; Zeginis et al. 2011]. In [Klein 2004; Noy and Musen 2002; Palma et al. 2007; Plessers and De Troyer 2005; Rogozan and Paquette 2005; Stojanovic et al. 2002], *high-level* change operations are employed to describe more complex changes, as, e.g., a complex modification in the subsumption hierarchy. A high-level language is preferable to a low-level one in terms of human interpretability, because it is more intuitive, concise and closer to human intuition, thereby capturing the semantics of a change more accurately [Klein 2004; Stojanovic 2004].

However, detecting *high-level* change operations incurs a number of issues related to machine-interpretability. As the detectable changes get more complicated, so does the

definition of their semantics, and the corresponding detection algorithm. In particular, the detection may be inefficient and/or based on matchers [Euzenat and Shvaiko 2007] or other heuristic-based techniques [Klein 2004] that make it difficult to provide any formal guarantees on the detection process. Another problem stems from the fact that it is impossible to define a complete list of high-level changes [Klein 2004], so there is no agreed set of operations that one could be based on. Finally, high-level changes should be intuitive and should capture the curators' perception on the change; however, identifying the changes that have these properties is quite challenging as this is often subjective to the curators and can be determined only through experiments in real settings.

As already mentioned, a framework for detecting high-level changes would be incomplete if the application of deltas were not supported. This is necessary especially in the case of community-driven (distributed) curated KBs, where local and remote KBs are interrelated; in this case, curators can exchange concise deltas (rather than entire versions) and apply them to inspect the effects of remote changes to their local KBs. Conciseness implies that the detected deltas should contain the smallest possible number of reported changes. Furthermore, the change application semantics should be consistent with the detection semantics, i.e., a detected delta between a pair of versions when applied upon the first version should result in the second version (thus allowing moving *forwards* or *backwards* from one version to another).

To the best of our knowledge, none of the current detection tools for RDF(S) fulfill all the above requirements. Most of them ([Klein 2004; Noy and Musen 2002; Palma et al. 2007; Plessers and De Troyer 2005; Rogozan and Paquette 2005; Stojanovic et al. 2002]) focus exclusively on human understandability by producing high-level deltas which are not equipped with an appropriate application semantics, nor give any formal guarantees on their expected behavior and properties. Such guarantees are currently provided only by works studying low-level change operations (e.g., [Zeginis et al. 2011]); however, as explained above, low-level deltas are less concise than high-level ones, and do not capture well the editor's intentions and perception of the changes, thereby producing less intuitive deltas. In this paper we attempt to fill this gap by introducing a high-level language of changes and its formal detection and application semantics, as well as a corresponding change detection algorithm, which satisfy the above needs for the case of RDF(S) KBs. More specifically, the main contributions of our work are:

— A set of *desired features* related to the detection and application semantics of a language of changes. These features are related to both human and machine interpretability. A language of changes that satisfies them is guaranteed to (a) be intuitive and capture as accurately as possible the perception and intent of editors regarding the performed changes, (b) be able to handle (describe) any possible change in a unique manner, and, (c) have well-defined formal and consistent detection and application semantics.
— A *formal language of changes* for RDF(S) KBs (including both its detection and application semantics), which considers operations at both ontology (i.e., schema) and instance (i.e., data) levels, as well as operations employing heuristic matching techniques. This language was carefully designed in order to satisfy the aforementioned desired features for a language of changes.
— *Efficient and effective change detection and application algorithms* which are *sound* and *complete* with respect to the proposed language.
— The *experimental evaluation* of the effectiveness and efficiency of the proposed algorithms in real settings using three well-established ontologies from the cultural (CIDOC [CIDOC 2010]), biological (Gene Ontology [Hill et al. 2008]) and entertain-

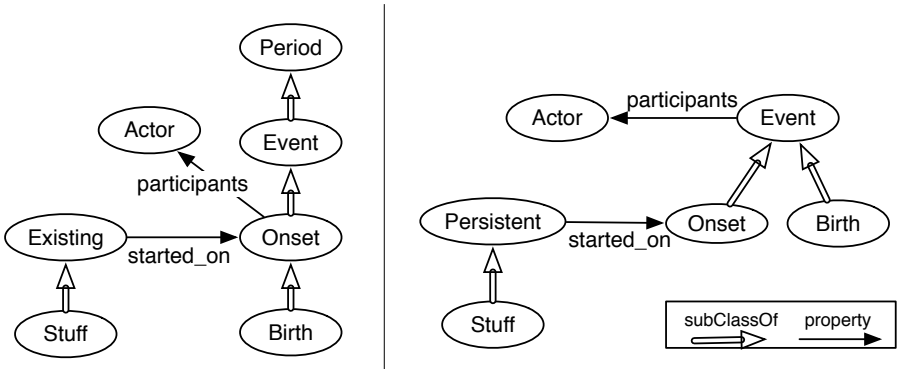
ment (Music Ontology [Raimond et al. 2010]) domains, which exhibit different structural characteristics and evolution patterns.

The paper is organized as follows: Section 2 presents a motivating example that will be used for illustration purposes throughout the paper and establishes the main requirements for our change detection framework. In Section 3, we provide some basic definitions upon which this work is based. Section 4 contains a brief and informal description of the proposed language of changes. Sections 5, 6 present respectively the detection and application semantics of the changes in the language, and show that it satisfies the required properties. Section 7 describes the proposed detection and application algorithms, shows their correctness with respect to the underlying language and establishes their computational complexity. In Section 8 we discuss our experimental evaluation on the selected ontologies and Section 9 discusses related work. We conclude in Section 10. Two appendices are also included, the first containing the proofs for all the presented results, and the second giving the formal definition of the changes in the proposed language of changes.

2. MOTIVATING EXAMPLE

Figure 1 depicts two versions of a toy ontology inspired by CIDOC [CIDOC 2010], which will be used as a running example throughout this paper. Table I illustrates the corresponding added and deleted triples (i.e., the *low-level delta*) as well as the high-level changes (*high-level delta*) that we are able to detect using the proposed framework of high-level changes. We can easily observe that even though the low-level delta contains all the changes that have been performed between the two versions, it is not really helpful for curators to understand and manipulate the detected changes as it essentially describes changes at a syntactic level. Our work exploits 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 I). Note that the high-level changes in the third column describe changes at the level of RDF(S) constructs (classes, properties, individuals), as well as in the possible relationships among them (subsumption, instantiation, etc.); this is closer to how humans usually understand a change.

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



(a) V_1 (b) V_2

Fig. 1. Motivating Example

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

Low-Level Delta		**High-Level Delta**
Added Triples	**Deleted Triples**	**Detected Changes**
(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)	

change *Generalize_Domain(participants,Onset,Event)* combines them into one. A similar case appears in the change of the position of "Birth" in the subsumption hierarchy, which our framework reports as a *Pull_up_Class*, and in the deletion of class "Period", where the low-level changes representing the deletion of all edges originating from, or ending in, the deleted class are combined in a single high-level change. Regarding the latter, only a subclass relation is deleted (with "Event"), whereas other relations, such as superclasses, supertypes, subtypes, comments and labels are absent and not deleted (denoted by empty sets in the parameters of *Delete_Class* in Table I). In total, in the example of Figure 1 only 4 high-level changes will be reported instead of 12 low-level ones. Reporting more concise deltas is a desirable feature, because it allows efficient exchange and storage of versions, as well as more intuitive visualization and explanation of the delta. Hence, high-level languages are more appealing.

Apart from being more concise, the reported high-level changes are also more intuitive. For example, the *Generalize_Domain* operation also informs us that the new domain is a superclass of the old. This captures more closely the perception of the curators regarding the change and thus facilitates a better analysis and understanding of the performed changes: if we know only that a domain changed we cannot presume anything about the validity of the existing instances, but if we know that the domain changed to a superclass we can assume, according to the RDF(S) specification, that the property is instantiated properly [Klein 2004].

Another interesting example is the change *Rename_Class(Existing,Persistent)*, 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_Class* (as well as other operations, such as merges or splits) is based on the fact that the *neighborhoods* of the elements "Existing" and "Persistent" (i.e., the triples involving the elements under question) are identical in the two versions; thus, we can safely assume that the two URIs correspond to the same real-world notion, so a renaming took place (rather than a class addition/deletion). Such an identification can only be made using heuristic techniques employed by matchers [Euzenat and Shvaiko 2007]. Unlike [Klein 2004; Noy and Musen 2002], we believe that the usage of heuristics should be restricted exclusively to the matching process and should not be an integral part of the detection process itself, to avoid imprecise and uncertain detection results. Note also that the *Rename_Class* operation is associated with some additional low-level changes that were caused by the renaming and do not correspond to real changes (see Table I).

Another interesting observation is that each of the low-level changes (in Table I) is associated with one, and only one, high-level change, thus creating a unique partition of low-level changes into high-level ones. This is an important property, because it leads to a unique set of high-level changes describing the evolution of a KB version to another, in an *unambiguous* manner. In other words, the changes between any given

pair of versions should be describable by the given language using *one, and only one*, set of high-level changes, to allow the detection algorithm to process any input and return a (single) corresponding high-level delta in a *deterministic* manner. The latter requirement calls for a careful definition of the change operations and is not directly related to the detection algorithm *per se*. In a nutshell, we claim that the detection algorithm should be based on the defined language of changes, rather than the other way round.

The aforementioned requirements for the language establish the need for a well-defined detection process. However, one of the main expected uses of the deltas is the reduction of the amount of data exchanged through the network when, e.g., parts of highly dynamic KBs need to be copied by different groups of curators in their own KBs; this can be done by exchanging the deltas, rather than the versions themselves. In order for this approach to work, one should know how to apply (execute) a given delta, and should also have guarantees that the application of the delta upon the old version would give the new one.

To this end, each operation in the language should be equipped with appropriate application semantics. Moreover, detection and application semantics should agree in the sense that, given two KBs V_1, V_2, the application (upon V_1) of the delta computed between them should give V_2; this way, one can determine the new version given the old one and the delta. In addition, in order to efficiently support multi-version repositories, every change in the language must have a unique reverse with the appropriate semantics; this would guarantee that, by keeping only the newest version in a repository and the set of changes that led to it, we will be able to reproduce all previous versions.

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 to achieve concise and intuitive deltas. On the other, fine-grained operations are necessary to capture subtle differences between a pair of versions. The existence of both fine 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 sections, we will formally describe a change language and a detection algorithm that avoids these problems and provably satisfies the above properties, while being efficient. It is worth noticing that some of the requirements (e.g., conciseness, intuitiveness) can be verified only through extensive experimentation in real settings as there is no way to formally prove the "usefulness" of a change language for curators. For this reason, in Section 8, we rely on real KBs (from different domains) and compare the deltas reported by our tool with the corresponding release notes manually provided by the editors when available.

3. FORMAL DEFINITIONS

3.1. RDF(S) KBs

The popularity of the RDF data model [McBride et al. 2004] is due to the flexible and extensible representation of information using *triples* of the form (*subject*, *predicate*, *object*). Assuming two disjoint and infinite sets $\mathbf{U}$, $\mathbf{L}$, denoting resources and literals respectively, $\mathcal{T} = \mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$ denotes the set of all triples. An ***RDF Graph*** V is then defined as a finite set of triples, i.e., $V \subseteq \mathcal{T}$.

Note that the definition of $\mathcal{T}$ ignores blank nodes (also called unnamed resources), which are defined in the RDF specification as "existential variables" in the same way that has been used in mathematical logic. Blank nodes are omitted for several reasons. First, because their actual usage in practice does not always follow their intended

Table II. RDFS Structural Inference Rules

Transitivity of class subsumption	Transitivity of property subsumption
$\frac{(C_1, \mathsf{subClassOf}, C_2), (C_2, \mathsf{subClassOf}, C_3)}{(C_1, \mathsf{subClassOf}, C_3)}$	$\frac{(P_1, \mathsf{subPropertyOf}, P_2), (P_2, \mathsf{subPropertyOf}, P_3)}{(P_1, \mathsf{subPropertyOf}, P_3)}$
Transitivity of class instantiation	**Transitivity of property instantiation**
$\frac{(x, \mathsf{type}, C_1), (C_1, \mathsf{subClassOf}, C_2)}{(x, \mathsf{type}, C_2)}$	$\frac{(P_1, \mathsf{subPropertyOf}, P_2), (x_1, P_1, x_2)}{(x_1, P_2, x_2)}$

Table III. RDFS Typing Inference Rules

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

semantics [Arenas et al. 2010]. Second, because the majority of the defined changes in our language concern RDFS (schema) entities, which are named resources (blank nodes at the schema layer may appear only in composite class and property definitions allowed by more expressive formalisms such as OWL). Third, because unnamed resources appear mainly at a "pure" RDF instance layer and are captured by the employed matchers.

RDFS [Brickley and Guha 2004] builds on the RDF language and introduces resource *types*, such as classes and properties. In this work we rely on a refined type system for RDF Graphs [Serfiotis et al. 2005] which identifies 5 different types, namely *individuals* (corresponding to data objects or instances), *schema classes* (or classes for brevity – corresponding to sets of individuals), *properties* (corresponding to relations between classes), *metaclasses* (corresponding to sets of classes) and *metaproperties* (corresponding to sets of properties). This typing system allows us to focus more on the nodes of RDF Graphs, rather than on their edges (i.e., triples), for defining intuitive high-level changes which are closer to curators' perception. The type of a given resource is determined by the triples it participates in, and by its relation with the special resources resource, class, property [Serfiotis et al. 2005]. Note that the typing system introduced in [Serfiotis et al. 2005] is not the same with the instantiation used in RDF(S) (through the type property); for example, an individual that is instantiated under two different classes through the type property is still of type "individual".

It should be stressed that RDFS [Brickley and Guha 2004] is also equipped with *inference rules*, namely *structural inference* (through the transitivity of subsumption relations – see Table II) and *type inference* (through the typing system, e.g., if p is a property, the triple $(p, \mathsf{type}, \mathsf{property})$ can be inferred – see Table III).

The notion of validity has been used in various contexts to overrule certain triple combinations, thereby imposing application-specific constraints (e.g., functional properties) or semantics (e.g., acyclicity in subsumptions) in RDF Graphs [Serfiotis et al. 2005; Motik et al. 2007; Lausen et al. 2008; Tao et al. 2010]. The validity constraints that we consider in this work were initially presented in [Serfiotis et al. 2005], and are required by a wide range of curated KBs. They concern the type uniqueness, i.e., that each resource has a unique type, the acyclicity of subClassOf and subPropertyOf relations, the uniqueness of properties' domains and ranges, that the domain and range of

a property should be subclasses of the corresponding domain/range of the superproperty, and that the subject and object of some property instance should be instances of the domain and range of the property respectively.

Some of the above restrictions are necessary for defining our operations and guarantee their properties (e.g., type uniqueness, domain/range uniqueness, correct classification of property instances), whereas others (acyclicity) are incorporated for efficiency purposes (cf. also the proof of Theorem 7.3 in Appendix A).

Given an RDF Graph satisfying the above validity constraints, the *closure* of V, denoted by $Cl(V)$, consists of all triples that are either explicitly stated in V, or can be inferred using *both* kinds of inference rules (Tables II, III). An ***RDF(S) Knowledge Base*** (or ***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 (Table III). We denote by $\mathbf{L}(V)$ the set of literals that appear in V. We denote by $\mathbf{U}(V)$ the set of *custom URIs* that appear in V, i.e., all the URIs appearing in V except from the built-in RDF(S) URIs, such as type or class.

3.2. Low-Level Deltas and Mappings

A *low-level change* is the addition or deletion of a triple from an RDF(S) KB. For a pair of RDF(S) KBs (V_1, V_2), the *low-level delta* between V_1, V_2, denoted by $\Delta(V_1, V_2)$ (or simply Δ) is defined as follows: $\Delta(V_1, V_2) = \langle V_2 \setminus V_1,\ V_1 \setminus V_2 \rangle$. For brevity, we use the notation Δ^+, Δ^- for $V_2 \setminus V_1$, $V_1 \setminus V_2$ respectively, denoting the triples t for which $Add(t)/Delete(t)$ was executed upon V_1 to get V_2.

Note that the computation of Δ is syntactical (rather than semantical) and based only on the explicit triples of V_1, V_2 [Zeginis et al. 2011]. Thus, if V_1, V_2 contain *redundant triples*, i.e., triples that are inferable from other triples using the structural inference rules of Table II (e.g., when $(A$, subClassOf, $B)$, $(B$, subClassOf, $C)$, $(A$, subClassOf, $C) \in V_1$), then these triples would affect Δ, and, consequently, the computation of the high-level delta (which is based on Δ). This feature is necessary because curators often want to be informed even about syntactical changes, so the reported delta should contain such triples as well. On the other hand, redundant triples do not carry any additional semantic information, so in many scenarios, the system should not report changes about them. If we are only interested in semantical changes then we should use a pre-processing phase that would remove any redundant triples from V_1, V_2 before computing Δ; that would make both the low-level and the high-level delta report only semantical changes.

The low-level delta alone may not be enough to fully capture the editor's perception of the change: sometimes we need to consider semantic information that remained unchanged (see, e.g., the change of the domain of "participants" in Figure 1 and the subsequent analysis presented in Section 2), and/or information on node renaming that a matcher provides (see, e.g., the change of the name of class "Existing" to "Persistent", where the matcher identified the two classes to be the same based on their neighborhoods, despite having different names).

The former will be handled through the introduction of *conditions* in the definition of high-level changes (see Definition 3.2 in the next subsection). To handle the latter (terminological changes), we introduce *sets of mappings*, which identify nodes (URIs) in the two versions that correspond to the "same" real-world entity. A mapping essentially associates a URI in V_1 to a new (different) URI in V_2, with which it has been found to "match". To capture the cases of merges and splits, we also allow mappings to associate a single URI to a set of URIs (and vice-versa). Thus, a mapping can have one of the following general forms:

— $\{u\} \rightsquigarrow \{u'\}$ (corresponds to renaming)

— $\{u\} \rightsquigarrow \{u_1, \ldots, u_n\}$ (corresponds to *split*)
— $\{u_1, \ldots, u_n\} \rightsquigarrow \{u\}$ (corresponds to *merge*)

The latter two can be further discriminated depending on whether the URI u appears in both the left and right part of the mapping. In the case of split, this means that one of the resources resulting from the split retains the URI of the split resource, whereas in the case of merge, this means that the resource resulting from the merge retains the URI of one of the merged resources. We call these cases *split into existing* and *merge into existing* respectively. In addition, we consider literal mappings to identify operations like *Change_Comment*, as opposed to a pair of *Add_Comment*, *Delete_Comment* operations. Note that literal mappings can only be one-to-one (i.e., of the form $\{u\} \rightsquigarrow \{u'\}$).

The above are more formally stated in the following definition:

Definition 3.1. A *mapping* μ between two RDF(S) KBs V_1, V_2 is an association, denoted by $S_1 \rightsquigarrow S_2$, which can take one of the following six forms:

(1) *Renaming:* $\{u\} \rightsquigarrow \{u'\}$, where $u \in \mathbf{U}(V_1) \setminus \mathbf{U}(V_2)$, $u' \in \mathbf{U}(V_2) \setminus \mathbf{U}(V_1)$
(2) *Split:* $\{u\} \rightsquigarrow \{u_1, \ldots, u_n\}$ for $n > 1$, where $u \in \mathbf{U}(V_1) \setminus \mathbf{U}(V_2)$, $u_1, \ldots, u_n \in \mathbf{U}(V_2) \setminus \mathbf{U}(V_1)$
(3) *Split into existing:* $\{u\} \rightsquigarrow \{u, u_1, \ldots, u_n\}$ for $n \geq 1$, where $u \in \mathbf{U}(V_1) \cap \mathbf{U}(V_2)$, $u_1, \ldots, u_n \in \mathbf{U}(V_2) \setminus \mathbf{U}(V_1)$
(4) *Merge:* $\{u_1, \ldots, u_n\} \rightsquigarrow \{u\}$ for $n > 1$, where $u_1, \ldots, u_n \in \mathbf{U}(V_1) \setminus \mathbf{U}(V_2)$, $u \in \mathbf{U}(V_2) \setminus \mathbf{U}(V_1)$
(5) *Merge into existing:* $\{u, u_1, \ldots, u_n\} \rightsquigarrow \{u\}$ for $n \geq 1$, where $u \in \mathbf{U}(V_1) \cap \mathbf{U}(V_2)$, $u_1, \ldots, u_n \in \mathbf{U}(V_1) \setminus \mathbf{U}(V_2)$
(6) *Literal mapping:* $\{u\} \rightsquigarrow \{u'\}$, where $u \in \mathbf{L}(V_1)$, $u' \in \mathbf{L}(V_2)$, $u \neq u'$

A *valid set of mappings* $\mathcal{M}$ is a set of mappings such that $\mu \neq \mu'$ implies $S_1 \cap S_1' = \emptyset$ and $S_2 \cap S_2' = \emptyset$ for any $\mu = S_1 \rightsquigarrow S_2, \mu' = S_1' \rightsquigarrow S_2'$, $\mu, \mu' \in \mathcal{M}$.

Note that the requirements associated with mappings take special care to discriminate between URIs that appear in both V_1, V_2 as opposed to those that appear in either, but not both of V_1, V_2. For example, a URI cannot be renamed into a URI that already existed in V_1. Valid sets of mappings are those for which no URI or literal appears in more than one mapping in the set. This guarantees that there will be no ambiguity as to where a certain URI is mapped. Valid mappings are used as input in our detection process (along with V_1, V_2).

Given a mapping $\mu = S_1 \rightsquigarrow S_2$, and two triples, t_1, t_2, we say that t_1 *maps to* t_2 through μ, denoted by $t_1 \rightsquigarrow_\mu t_2$ (or simply $t_1 \rightsquigarrow t_2$ when μ is obvious), iff t_2 is obtained by replacing in t_1 the URI(s) appearing in S_1 with URI(s) from S_2.

Mappings are computed by *matchers* [Euzenat and Shvaiko 2007; Ferrara et al. 2011; Flouris et al. 2008], which employ various sophisticated, heuristic-based techniques that consider the neighborhood of RDF Graph nodes and/or application-specific information to identify nodes with different names that correspond to the same real world entity. For instance, in the example of Figure 1, the detection of the heuristic change *Rename_Class(Existing, Persistent)* is based on the fact that the classes "Existing" and "Persistent" have the same neighborhood, so they are considered to correspond to the same entity. This fact is expressed by a mapping {Existing} $\rightsquigarrow$ {Persistent}.

3.3. High-Level Deltas

A *high-level change* c (e.g., *Generalize_Domain(participants,Onset,Event)*) is composed of the *operation* (e.g., *Generalize_Domain*) and the *parameters of the operation* (e.g.,

participants,Onset,Event). As explained above, the detection of a change is based on (a) the existence of certain triples in the low-level delta, (b) the existence of a certain mapping, and, (c) the satisfaction of certain contextual conditions in the two versions enabling its detection. Formally, a *high-level change* (or simply *change*) is defined as follows:

Definition 3.2. A *change* c is a quadruple $\langle \delta^+(c), \delta^-(c), \mathcal{M}(c), \phi(c) \rangle$, where

— $\delta^+(c) \subseteq \mathcal{T}$ (or simply δ^+) is the set of *required added triples* and corresponds to the triples that should be in the new version but not in the old one in order for c to be detected.
— $\delta^-(c) \subseteq \mathcal{T}$ (or simply δ^-) is the set of *required deleted triples* and corresponds to the triples that should be in the old version but not in the new one in order for c to be detected.
— $\mathcal{M}(c)$ (or simply $\mathcal{M}$) is the set of *required mappings* and corresponds to the mappings that should exist between the two versions in order for c to be detected.
— $\phi(c)$ (or simply ϕ) is the set of *required conditions* and corresponds to the conditions that should be true in order for c to be detected. A condition is a logical formula that could contain mappings, equalities/inequalities (e.g., $u = u'$ for two URIs u, u') and inclusion relations of the form $t \in V$, $t \notin V$, $t \in Cl(V)$, $t \notin Cl(V)$ (where $t \in \mathcal{T}$, V is either the new or the old version and $Cl(V)$ is the closure of V according to the structural and typing inference rules).

A *high-level delta* C is a set of high-level changes. We restrict our attention to changes c for which $\delta^+(c) \cup \delta^-(c) \neq \emptyset$ and $\delta^+(c) \cap \delta^-(c) = \emptyset$. The first condition guarantees that, in order for c to be detected, at least *something* must be in Δ^+ or Δ^- (i.e., something has actually changed between V_1 and V_2). The second condition guarantees that no change would require (for its detection) the addition and deletion of the same triple to happen at the same time.

4. THE PROPOSED LANGUAGE OF CHANGES

In this work we propose a specific *language of changes* (denoted by $\mathcal{L}$) which consists of 132 changes. $\mathcal{L}$ is defined at the level of RDF(S) constructs, i.e., it contains changes capturing the modifications (e.g., addition, deletion, renaming, move in the hierarchy, change of domain/range etc.) that the various constructs (classes, properties etc.) of an RDF(S) KB can undergo (see also Table I). This is based on how humans tend to understand and describe changes (e.g., in release notes of curators [CIDOC 2010]); for example, it is more natural to say that a property P was deleted, than saying that the triple $(P, \mathsf{type}, \mathsf{property})$ was deleted.

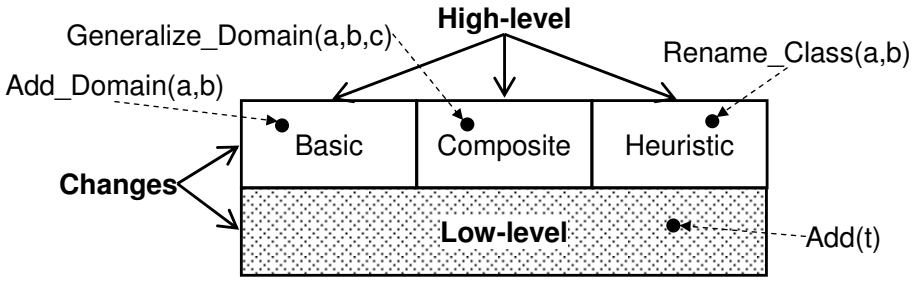


Fig. 2. A Classification of Changes

As discussed in Section 1, both fine-grained and coarse-grained changes are necessary in order to enable determinism, conciseness and intuitiveness. For this reason, we follow a common approach in the literature [Klein 2004; Plessers and De Troyer 2005; Stojanovic 2004] and define both *basic* (i.e., fine-grained changes on individual RDF Graph nodes or edges) and *composite* high-level changes (coarse-grained changes affecting several nodes and/or edges). Note that basic and composite changes do not have required mappings, i.e., $\mathcal{M} = \emptyset$. Unlike other related works (e.g., [Plessers et al. 2007]), we consider also (as a separate category) *heuristic* changes whose detection conditions require the existence of a mapping, i.e., $\mathcal{M} \neq \emptyset$. The changes' classification is depicted in Figure 2.

Apart from their intuitive character, the changes in $\mathcal{L}$ were carefully selected (and defined) so as to achieve the required characteristics of human and machine interpretability. As a result, $\mathcal{L}$ results in concise and intuitive deltas, which are close to human perception (see Section 8); moreover, $\mathcal{L}$ has well-defined, formal and consistent detection and application semantics and satisfies a number of formal properties, such as the ability to describe any possible change in a unique and deterministic manner (see Sections 5, 6). Note that $\mathcal{L}$ is, to our knowledge, the first language of changes that achieves these goals. However, $\mathcal{L}$ is not the only language with these characteristics; it is a subject of future work to determine whether a better one can be defined.

Below we informally describe the changes that are included in $\mathcal{L}$. The interested reader is referred to Appendix B for the full list of these changes, along with their formal definition.

4.1. Basic Changes

Regarding basic changes, our main objective was to guarantee the completeness of the language at a fine-grained level, i.e., to guarantee that each possible fine-grained change that can be performed on an RDF(S) KB is captured by a basic change. To do so, we considered all possible additions/deletions of both URIs and relationships between such URIs that can appear in an RDF(S) KB, leading to a total of 54 basic change operations.

Additions and Deletions of URIs. URIs can be added or deleted; addition refers to the introduction of a new URI that did not previously appear in the RDF(S) KB, whereas deletion refers to the elimination of all its occurrences from the RDF(S) KB. Recall that in this work we identify 5 different types of URIs, so, in order for the operation to additionally encode the type information of the added/deleted URI, we introduce one "add" and one "delete" operation per type. This leads to the following 10 operations: *Add_Type_Class*, *Delete_Type_Class*, *Add_Type_Metaclass*, *Delete_Type_Metaclass*, *Add_Type_Metaproperty*, *Delete_Type_Metaproperty*, *Add_Type_Property*, *Delete_Type_Property*, *Add_Type_Individual* and *Delete_Type_Individual*.

Retyping of URIs. Another important category of changes deals with resources that change their type. To support all possible retypings, we defined 20 such *Retype_...* operations, one for each pair of types (see Appendix B for the full list).

Subsumption Relationships. Changes in subsumption relationships are described by specialized basic changes. They are distinguished depending on the type of the involved URIs (classes, properties, metaclasses, metaproperties) and on whether we have an addition or deletion. This leads to a total of 8 changes: *Add_Superclass*, *Delete_Superclass*, *Add_SuperMetaclass*, *Delete_SuperMetaclass*, *Add_SuperMetaproperty*, *Delete_SuperMetaproperty*, *Add_Superproperty* and *Delete_Superproperty*.

Instantiation Relationships. Another interesting relationship between URIs described by basic changes concerns the addition and deletion of instantiation relationships between classes/metaclasses (*Add_Type_To_Class*, *Delete_Type_From_Class*), between properties/metaproperties (*Add_Type_To_Property*, *Delete_Type_From_Property*), between individuals/classes (*Add_Type_To_Individual*, *Delete_Type_From_Individual*) and between property instances/properties (*Add_Property_Instance*, *Delete_Property_Instance*). This category of changes contains 8 operations.

Domains and Ranges. The addition and deletion of domain or range for a property is also described by basic changes, leading to 4 different operations: *Add_Domain*, *Delete_Domain*, *Add_Range*, *Delete_Range*.

Comments and Labels. Finally, to describe the addition and deletion of comments and labels for URIs we defined another set of 4 operations: *Add_Comment*, *Delete_Comment*, *Add_Label*, *Delete_Label*.

4.2. Composite Changes

Composite changes are meant to capture coarse-grained changes that are useful in practice and often used by curators. They correspond to the simultaneous addition/deletion of several triples, and, as we will show later, they can be equivalently represented using a set of basic changes. They were defined based on the idea that certain fine-grained changes usually appear together; for example, when a class A is deleted, all triples that associate A with other URIs (via subsumptions, instantiations etc.) must be removed as well. Thus, a single *Delete_Class* operation could be used to describe this change, rather than a set of *Delete_Type_Class*, *Delete_Superclass* etc. operations. Note that the basic change *Delete_Type_Class(A)* only deals with the deletion of the triples that define A and does not deal with the other triples involving A. Based on this idea, we defined several composite change operations, namely 51, for describing such common, coarse-grained changes.

Additions and Deletions of URIs. The composite changes dealing with the addition/deletion of resources are actually "enhanced" versions of the corresponding basic changes (e.g., *Delete_Class* is an "enhanced" version of *Delete_Type_Class*). As with basic changes, we have one pair of changes for each type, leading to a set of 10 operations: *Add_Class*, *Delete_Class*, *Add_Metaclass*, *Delete_Metaclass*, *Add_Metaproperty*, *Delete_Metaproperty*, *Add_Property*, *Delete_Property*, *Add_Individual* and *Delete_Individual*.

Subsumption Relationships. The composite changes dealing with subsumptions are much more complicated than their basic counterparts. Let's consider a class A which has a different set of parents in V_1, V_2. Let us denote by $\mathcal{B}_1$ the set of parents of A in V_1 that are not its parents in V_2 and by $\mathcal{B}_2$ the set of parents of A in V_2 that are not its parents in V_1. Then, we distinguish the following cases:

— If all classes in $\mathcal{B}_1$ are subclasses of the classes in $\mathcal{B}_2$, then the class was moved upwards in the hierarchy, and its change is described using *Pull_up_Class*.
— If all classes in $\mathcal{B}_1$ are superclasses of the classes in $\mathcal{B}_2$, then the class was moved downwards in the hierarchy, and its change is described using *Pull_down_Class*.
— If none of the classes in $\mathcal{B}_1, \mathcal{B}_2$ are connected with a subsumption relationship, then the class was moved "horizontally" in the subsumption hierarchy, and its change is described using *Move_Class*.
— Finally, to capture the cases where none of the above three cases applies (e.g., when some of the classes in $\mathcal{B}_1$ are subclasses of some, but not all, of the classes in $\mathcal{B}_2$,

or when the classes in $\mathcal{B}_1, \mathcal{B}_2$ have also undergone some kind of relocation in the hierarchy themselves), we use the generic *Change_Superclasses* operation.

It should be noted that in all the above cases we assume that $\mathcal{B}_1, \mathcal{B}_2$ are non-empty. If $\mathcal{B}_1$ is empty (but $\mathcal{B}_2$ is not), then A has some new parents without losing any of its old ones; the change is described by *Group_Classes*. Similarly, if $\mathcal{B}_2$ is empty but $\mathcal{B}_1$ is not, then A lost some of its parents without getting any new ones, and the operation *Ungroup_Classes* is used to describe this change.

Finally, note that the above operations require (in their conditions, ϕ) that all the involved resources (i.e., A, as well as the resources in $\mathcal{B}_1, \mathcal{B}_2$) exist in both versions, and are classes. In a different case, these operations are not applicable, and the change is described using other operations (composite operations dealing with the addition/deletion of resources and/or basic changes, depending on the case).

The above 6 composite operations are used to describe hierarchy changes in classes. The same ideas are used to define operations describing changes for the metaclass, metaproperty and property hierarchies. Thus, composite changes for subsumption relationships contain a total of 24 changes.

Instantiation Relationships. Similar ideas are used for the classification hierarchies. First, we consider all the classes that the individual used to be instantiated under, and the ones that is newly instantiated under. If they are all connected with the same kind of subsumption relationship (i.e., if all the new ones are subclasses/superclasses of the old ones), then we report a *Reclassify_Individual_Higher* or *Reclassify_Individual_Lower* change; if no such pattern appears, then we have a generic *Reclassify_Individual* operation to cover this case. The same idea applies for classes (which are instantiated under metaclasses) and properties (which are instantiated under metaproperties), giving a total of 9 operations.

Domains and Ranges. The same approach is followed to report a change of the domain of some property: the domain could be generalized or specialized (operations *Generalize_Domain*, *Specialize_Domain* respectively), or the new domain may be unrelated to the old one (operation *Change_Domain*). The same applies for property ranges, except that for property ranges it also makes sense to consider the special case where the range of the property is changed in such a way that an object property becomes a datatype property (or vice-versa); to support those cases, we have defined the operations *Change_To_Datatype_Property* and *Change_To_Object_Property* (in addition to the operations *Generalize_Range*, *Specialize_Range* and *Change_Range*). The total number of operations in this category is 8.

4.3. Heuristic Changes

Finally, we have defined 27 heuristic changes, which are meant to capture terminological changes. Heuristic changes are based on the existence of a mapping, and the form of the mapping essentially classifies heuristic changes into four major categories.

Renamings (One-to-one URI Mappings). When the mapping is of the form $\{u\} \rightsquigarrow \{u'\}$, and u, u' are URIs, then we deal with a *renaming*. As usual, renamings are categorized depending on the type of the renamed resource, so we have 5 different operations: *Rename_Class*, *Rename_Metaclass*, *Rename_Metaproperty*, *Rename_Property* and *Rename_Individual*.

Merges (Many-to-one URI Mappings). The operations referring to merges are based on (URI) mappings of the form $\{u_1, \ldots, u_n\} \rightsquigarrow \{u\}$. As before, we discriminate merges depending on the type of the merged resources (individuals, classes etc.). An additional dimension is imposed by the fact that the merged resources could merge into an exist-

ing resource (i.e., the URI of one of the merged resources persists), or they could merge into a new resource (i.e., the resource resulting from the merge gets a new URI). These two cases are discriminated by checking whether $u \in \{u_1, \ldots, u_n\}$ or not. Thus, in total, we have 10 different merge operations: *Merge_Classes*, *Merge_Classes_Into_Existing*, *Merge_Properties*, *Merge_Properties_Into_Existing* etc.

Splits (One-to-many URI Mappings). Split operations are similar, except from the fact that the (URI) mappings are of the form $\{u\} \rightsquigarrow \{u_1, \ldots, u_n\}$. As before, we have 10 different split operations depending on the type of the split resource and on whether the URI of the original resource persists in the new version.

Comments and Labels (Literal Mappings). Finally, the mapping may refer to literals, in which case we have 2 heuristic changes, namely *Change_Comment* and *Change_Label*, which are used to describe cases where a comment or a label associated with a resource changes. They both use mappings of the form $\{u\} \rightsquigarrow \{u'\}$ where u, u' are different literals.

5. CHANGE DETECTION SEMANTICS

5.1. Consumption

As explained in Section 2, our approach is based on partitioning (or "grouping") of low-level changes into high-level ones. Thus, once a change has been detected, we can safely eliminate from the low-level delta the triples that are involved in its detection, because they have already been "used" during the detection process. The elimination of triples from the low-level delta should be seen as a consequence of the detection of a specific change and will be called *consumption*.

The consumed triples of a change c usually include just the triples required for its detection ($\delta^+(c), \delta^-(c)$), as, e.g., in the case of *Generalize_Domain(participants, Onset,Event)* in the motivating example (Figure 1). However, this is not true when mappings are involved. For instance, the addition of (Stuff,subClassOf,Persistent) that appears in the low-level delta of the motivating example (Table I) is not a "real" change, but an artifact of the renaming. Thus, it would make sense to be associated with (and consumed by) the heuristic change *Rename_Class(Existing,Persistent)*. Therefore, *Rename_Class(Existing,Persistent)* should consume the triples that are required for its detection ((Persistent,type,class) from Δ^+ and (Existing,type,class) from Δ^-), as well as the triples that correspond to the artifacts of the terminological change: (Stuff,subClassOf,Persistent) and (started_on,domain,Persistent) from Δ^+, and (Stuff,subClassOf,Existing) and (started_on,domain,Existing) from Δ^-.

The low-level changes that should be consumed due to the mapping could be defined as all the triples that contain URIs involved in the mapping. That would give the correct result in the above example: since the mapping is of the form $\{\text{Existing}\} \rightsquigarrow \{\text{Persistent}\}$, every triple containing "Existing"/"Persistent" in Δ^-/Δ^+ respectively should be consumed, because we know that such low-level changes do not correspond to "real" changes, but are caused by the renaming. The problem with this naive approach is that other "real" changes may actually affect these triples as well.

This can be seen in the example of Figure 3, where we assume that the mapping $\{A\} \rightsquigarrow \{A_1, A_2, A_3\}$ was produced by the matcher. This mapping indicates that A was split into A_1, A_2, A_3, so the change *Split_Class(*$A, \{A_1, A_2, A_3\}$*)* should be detected. Moreover, note that even though A used to be a subclass of F, the nodes that replaced A (A_1, A_2, A_3) are not all subclasses of F. Using the naive approach, the detection of *Split_Class* should cause the consumption of all triples in Δ containing A, A_1, A_2 or A_3, i.e., the entire delta (see the left part of Table IV); we would get the same result if A_2, A_3 were actually subclasses of F in the new version. Thus, the naive approach

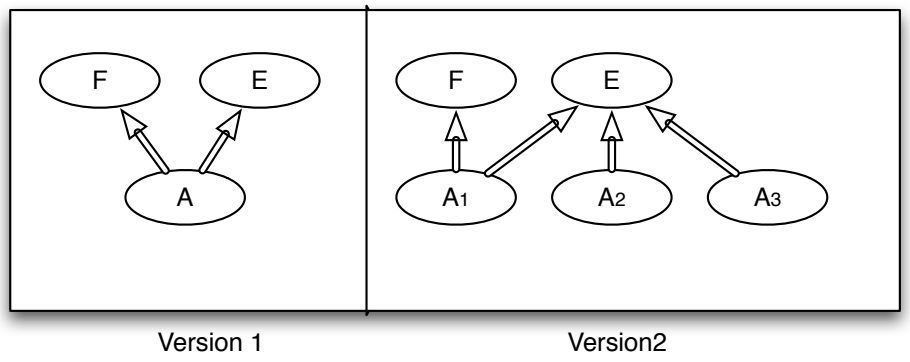


Fig. 3. Splitting a Class While Changing the Hierarchy

hides the fact that part of the neighborhood was not transferred intact in the new version and does not allow us to detect and describe changes in the neighborhood.

To avoid this problem, we use the following general principle: if a part of the neighborhood of the mapped URI (e.g., a subclass relationship that involves it) is transferred intact during the evolution (i.e., it is deleted and subsequently added with the new name), then the addition/deletion is due to the mapping and should be associated with (and consumed by) the heuristic change. On the other hand, the parts of the neighborhood that are not transferred intact (i.e., they are not identical in the two versions) correspond to changes that happened in addition to the heuristic change, so the low-level changes that describe them should not be consumed by the heuristic change, in order to allow other high-level changes to consume them and describe the neighborhood change.

So, take a triple $t \in V_1$ that was deleted ($t \in \Delta^-$). Triple t should be consumed iff all its associated triples, through μ, were added/deleted as appropriate. This means that all triples t' such that $t \rightsquigarrow t'$ must have been added ($t' \in \Delta^+$) and all triples t'' that are mapped to each of these t' (i.e., $t'' \rightsquigarrow t'$) must have been deleted ($t'' \in \Delta^-$); all these triples (t, t', t'') must be consumed together (from Δ^+ or Δ^-, as appropriate). A similar process should be followed if $t \in V_2$. Thus, we can define the *low-level changes consumed by the mapping* μ of a change c as a pair of sets of triples $\langle \xi^+_M(\mu, c, V_1, V_2), \xi^-_M(\mu, c, V_1, V_2) \rangle$ such that:

— $t \in \xi^+_M(\mu, c, V_1, V_2)$ iff $t \in \Delta^+$, $t \notin \delta^+(c)$ and $t' \in \Delta^-$, $t' \notin \delta^-(c)$ for all t' such that $t' \rightsquigarrow t$ and $t'' \in \Delta^+$, $t'' \notin \delta^+(c)$ for all t'' such that $t' \rightsquigarrow t''$.
— $t \in \xi^-_M(\mu, c, V_1, V_2)$ iff $t \in \Delta^-$, $t \notin \delta^-(c)$ and $t' \in \Delta^+$, $t' \notin \delta^+(c)$ for all t' such that $t \rightsquigarrow t'$ and $t'' \in \Delta^-$, $t'' \notin \delta^-(c)$ for all t'' such that $t'' \rightsquigarrow t'$.

Table IV. Low-level Delta and Consumed Triples for *Split_Class* for the Example in Figure 3

Low-Level Delta	
Added Triples	**Deleted Triples**
$(A_1$, type, class)	$(A$, type, class)
$(A_2$, type, class)	$(A$, subClassOf, $E)$
$(A_3$, type, class)	$(A$, subClassOf, $F)$
$(A_1$, subClassOf, $E)$	-
$(A_2$, subClassOf, $E)$	-
$(A_3$, subClassOf, $E)$	-
$(A_1$, subClassOf, $F)$	-

Consumption for *Split_Class*(A,$\{A_1, A_2, A_3\}$)	
Added Triples	**Deleted Triples**
$(A_1$, type, class)	$(A$, type, class)
$(A_2$, type, class)	
$(A_3$, type, class)	
$(A_1$, subClassOf, $E)$	$(A$, subClassOf, $E)$
$(A_2$, subClassOf, $E)$	
$(A_3$, subClassOf, $E)$	

The following theorem shows that a low-level change is consumed by a mapping iff all its associated (through the mapping) changes are also consumed[13]:

THEOREM 5.1. *If* $t_1 \rightsquigarrow t_2$ *then* $t_1 \in \xi^-_M(\mu, c, V_1, V_2)$ *iff* $t_2 \in \xi^+_M(\mu, c, V_1, V_2)$.

The consumed triples of a heuristic change c should be those in $\delta^+(c), \delta^-(c)$, plus those in $\langle \xi^+_M(\mu, c, V_1, V_2), \xi^-_M(\mu, c, V_1, V_2)\rangle$ for the mappings $\mu \in \mathcal{M}(c)$:

Definition 5.2. The *consumed triples of* c *with respect to* V_1, V_2, denoted by $\Xi(c, V_1, V_2) = \langle \xi^+(c, V_1, V_2), \xi^-(c, V_1, V_2)\rangle$ are defined as:

$$\xi^+(c, V_1, V_2) = \delta^+(c) \cup \Big(\bigcup_{\mu \in \mathcal{M}(c)} \xi^+_M(\mu, c, V_1, V_2)\Big),$$
$$\xi^-(c, V_1, V_2) = \delta^-(c) \cup \Big(\bigcup_{\mu \in \mathcal{M}(c)} \xi^-_M(\mu, c, V_1, V_2)\Big).$$

Note that for basic and composite changes (where $\mathcal{M}(c) = \emptyset$), the consumed triples are just those in $\delta^+(c), \delta^-(c)$. The right part of Table IV shows the low-level changes that should be consumed in the example of Figure 3 by *Split_Class*. In particular, (A_1, subClassOf, E) $\in \Delta^+$ is consumed, along with all its associated triples ((A, subClassOf, E) $\in \Delta^-$, (A_2, subClassOf, E) $\in \Delta^+$, (A_3, subClassOf, E) $\in \Delta^+$). On the other hand, (A_1, subClassOf, F) $\in \Delta^+$ is not consumed: even though (A, subClassOf, F) $\in \Delta^-$, the fact that (A_2, subClassOf, F) $\notin \Delta^+$ prevents us from including both (A, subClassOf, F) and (A_1, subClassOf, F) in ξ^-_M, ξ^+_M respectively. Thus, the low-level changes that are not consumed can be used to detect *Delete_Superclass(*A, F*)* and *Add_Superclass(*A_1, F*)*, which describe the change in the neighborhood.

5.2. Detectability of Changes

The next step is to determine when a change is *detectable*. The detection semantics of a change c is given by its definition: in particular, a change c should be detected if its required added/deleted triples are in Δ, its required mappings exist in the set of mappings and the required conditions are true. However, under the above definition, certain changes could be simultaneously detectable. For instance, in the motivating example (Figure 1 and Table I), both *Delete_Domain(participants,Onset)*, and *Generalize_Domain(participants,Onset,Event)* (cf. Appendix B), would be detectable. This fact would jeopardize determinism, because whichever operation is considered first would consume its corresponding triples, making the other non-detectable. To avoid this problem, we define priorities: heuristic changes are detected first, followed by composite changes, whereas basic changes are detected last. These priorities allow us to sort out what to detect in case of changes whose set of consumed changes overlap. Formally:

Definition 5.3. Two changes c, c' are called *overlapping with respect to* V_1, V_2 iff $\xi^+(c, V_1, V_2) \cap \xi^+(c', V_1, V_2) \neq \emptyset$ or $\xi^-(c, V_1, V_2) \cap \xi^-(c', V_1, V_2) \neq \emptyset$.

Definition 5.4. Consider two RDF(S) KBs V_1, V_2, their respective $\Delta = \langle \Delta^+, \Delta^-\rangle$, a valid set of mappings $\mathcal{M}$ between V_1, V_2 and a change c. Then:

— If c is a heuristic change then c is *detectable* iff $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\mathcal{M}(c) \subseteq \mathcal{M}$ and $\phi(c)$ is true.
— If c is a composite change then c is *detectable* iff $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and there is no detectable heuristic change c' that overlaps with c.
— If c is a basic change then c is *detectable* iff $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and there is no detectable heuristic or composite change c' that overlaps with c.

[13]Formal proofs for this and subsequent theorems are given in Appendix A.

In our running example, *Change_Domain(participants,Onset,Event)* is not detectable because its conditions are not true. On the other hand, *Generalize_Domain (participants,Onset,Event)* is detectable. Finally, the basic change *Delete_Domain (participants,Onset)* is not detectable, because of the overlap with the detectable composite change *Generalize_Domain(participants,Onset,Event)* (they both consume the low-level change (participants,domain,Onset) $\in \Delta^-$).

5.3. Results on Detection Semantics

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

THEOREM 5.5. *Any low-level change in the delta between two RDF(S) KBs is consumed by some detectable high-level change.*

The *unambiguity* of $\mathcal{L}$ is guaranteed by the following theorem, which states that we cannot find two detectable changes that overlap:

THEOREM 5.6. *The set of detectable changes for any pair of RDF(S) KBs does not contain overlapping changes.*

The combination of these two theorems shows the deterministic nature of the detection process, as they imply that the set of low-level changes can be uniquely partitioned into disjoint subsets, each corresponding to one high-level change.

6. APPLICATION SEMANTICS

6.1. Executing Changes

So far, we were only concerned with the detection of changes, but the application semantics, determining the result of applying some change(s) upon an RDF(S) KB, is equally important, because it allows curators to store or exchange deltas, rather than versions, to describe a change. The application and detection semantics should be consistent, in the sense that, given two RDF(S) KBs V_1, V_2, the application (upon V_1) of the delta computed between them should give V_2. To achieve this effect, a change's application should add/delete to/from V_1 its consumed triples. The problem with this statement is that the computation of Ξ needs the low-level delta which is unknown before the application of a change.

In order to tackle this issue, we will use the same intuitions as in the discussion on consumption. In particular, the effects of applying a change c should be the changes in $\delta^+(c), \delta^-(c)$, plus the changes caused by the required mapping, i.e., the changes causing the "transition" of the neighborhood under the new name(s); high-level changes that are simultaneously applied and modify part of the neighborhood should be taken into account, essentially "blocking" the heuristic change from having any effects on this part of the neighborhood.

Considering the example of Figure 1, the application of *Generalize_Domain (participants,Onset,Event)* should cause the addition of (participants,domain,Event) and the deletion of (participants,domain,Onset) (the triples in $\delta^+(c), \delta^-(c)$). The application of *Rename_Class(Existing,Persistent)* should add the triples in $\delta^+(c)$ ((Persistent,type,class)), delete the triples in $\delta^-(c)$ ((Existing,type,class)), as well as add/delete the triples that would cause "Existing" to be replaced by "Persistent" (i.e., add (Stuff,subClassOf,Persistent), (started_on,domain,Persistent), and delete (Stuff,subClassOf,Existing), (started_on,domain,Existing)).

On the other hand, in the example of Figure 3, if we applied, on V_1, *Split_Class(*A*,* $\{A_1, A_2, A_3\}$*)* along with the rest of the changes (*Delete_Superclass(*A, F*)*, *Add_*

Table V. Applied Triples for *Split_Class* for the Example of Figure 3

Applied Triples	
Split_Class(A, $\{A_1, A_2, A_3\}$***)***	
α^+	α^-
(A_1, type, class)	(A, type, class)
(A_2, type, class)	(A, subClassOf, E)
(A_3, type, class)	-
(A_1, subClassOf, E)	-
(A_2, subClassOf, E)	-
(A_3, subClassOf, E)	-

Table VI. Applied Triples for *Split_Class* Alone

Applied Triples (Naive)	
Split_Class(A, $\{A_1, A_2, A_3\}$***)***	
α^+	α^-
(A_1, type, class)	(A, type, class)
(A_2, type, class)	(A, subClassOf, E)
(A_3, type, class)	(A, subClassOf, F)
(A_1, subClassOf, E)	-
(A_2, subClassOf, E)	-
(A_3, subClassOf, E)	-
(A_1, subClassOf, F)	-
(A_2, subClassOf, F)	-
(A_3, subClassOf, F)	-

Superclass(A_1, F*)*, then the latter should affect the application of *Split_Class(A,* $\{A_1, A_2, A_3\}$*)* as shown in Table V (these are the same as its consumed triples shown in Table IV). In effect, the existence of *Delete_Superclass(*A, F*)*, *Add_Superclass (*A_1*,F)* indicates that this part of the neighborhood was changed by other operations, so the application of *Split_Class* should ignore this part. Note that if *Split_Class(A,* $\{A_1, A_2, A_3\}$*)* was applied alone, its application effect would also include the deletion of (A, subClassOf, F) and the addition of (A_i, subClassOf, F), $i = 1, 2, 3$ (see Table VI). This shows that the application effects of a change depend on the set of changes applied as a whole, rather than just the change itself.

Formally, for a set of changes C and a version V, the *low-level changes caused by the mapping* μ is a pair of sets of triples $\langle \alpha^+_M(\mu, C, V), \alpha^-_M(\mu, C, V) \rangle$ such that:

— $t \in \alpha^+_M(\mu, C, V)$ iff $t \notin V$, and for all t', $t' \rightsquigarrow t$ implies $t' \in V$ and there is no $c \in C$ such that $t' \in \delta^-(c)$ and for all t'', $t' \rightsquigarrow t''$ implies $t'' \notin V$ and there is no $c \in C$ such that $t'' \in \delta^+(c)$.
— $t \in \alpha^-_M(\mu, C, V)$ iff $t \in V$, and for all t', $t \rightsquigarrow t'$ implies $t' \notin V$ and there is no $c \in C$ such that $t' \in \delta^+(c)$ and for all t'', $t'' \rightsquigarrow t'$ implies $t'' \in V$ and there is no $c \in C$ such that $t'' \in \delta^-(c)$.

As with the detection semantics, the following theorem shows that a low-level change is caused by a mapping μ iff all its associated changes (through μ) are also caused by μ:

THEOREM 6.1. *If* $t_1 \rightsquigarrow t_2$ *then* $t_1 \in \alpha^-_M(\mu, C, V)$ *iff* $t_2 \in \alpha^+_M(\mu, C, V)$.

Given a set of changes C, the triples to apply should be those in $\delta^+(c), \delta^-(c)$, plus those in $\langle \alpha^+_M(\mu, C, V), \alpha^-_M(\mu, C, V) \rangle$ for the mappings $\mu \in \mathcal{M}(c)$, for all $c \in C$:

Definition 6.2. The *applied triples of a set of changes* C *upon* V, denoted by $\alpha(C, V) = \langle \alpha^+(C, V), \alpha^-(C, V) \rangle$ are defined as:

$\alpha^+(C,V) = \bigcup_{c\in C}\left(\delta^+(c) \cup \left(\bigcup_{\mu\in\mathcal{M}(c)} \alpha^+_M(\mu,C,V)\right)\right)$,
$\alpha^-(C,V) = \bigcup_{c\in C}\left(\delta^-(c) \cup \left(\bigcup_{\mu\in\mathcal{M}(c)} \alpha^-_M(\mu,C,V)\right)\right)$.
The *application of* C *upon* V, denoted by $V \bullet C$ is defined as $V \cup \alpha^+(C,V) \setminus \alpha^-(C,V)$.

The following theorem shows that the requirement of having consistent application and detection semantics has been fulfilled:

THEOREM 6.3. *Consider two RDF(S) KBs* V_1, V_2 *and their corresponding set of detectable changes* C*. Then* $V_1 \bullet C = V_2$.

6.2. Bulk and Sequential Application

Note that Definition 6.2 refers to *bulk* change application, i.e., in one step, in the sense that all the additions and deletions dictated by the changes are accumulated and applied in a single transaction. Alternatively, one could consider *sequential* change application, where we select one change from C, apply it (independently from the others) to V, then select the next one to apply and so on. This way, changes are not accumulated, but applied independently. Thus, if $C = \{c_1, \ldots, c_n\}$ and decide to apply the changes in that order, then the sequential application of C upon V would be: $(\ldots((V \bullet \{c_1\}) \bullet \{c_2\})\ldots) \bullet \{c_n\}$.

It is easy to see that sequential application does not allow us to take into account interactions between the changes in C (like the interaction between *Split_Class(*A, $\{A_1, A_2, A_3\}$*)* and *Delete_Superclass(*A, F*)*, *Add_Superclass(*A_1, F*)* in the example of Figure 3); therefore, the result of the two application types may differ, and sequential application is not consistent with the detection semantics. In addition, sequential application may be sensitive to the order of application. Nonetheless, there are cases where sequential and bulk application give the same results.

To study this phenomenon, we define the notions of *conflicting* and *independent* changes. We say that c_1 *conflicts with* c_2 iff $\delta^+(c_1) \cap \delta^-(c_2) \neq \emptyset$ or $\delta^+(c_2) \cap \delta^-(c_1) \neq \emptyset$. Intuitively, if c_1, c_2 are conflicting then the application/detection of c_1 requires the addition of a triple whose deletion is required by c_2 (or vice-versa). Thus, when sequentially applying c_1, c_2 (in any order), said triple will be added and subsequently deleted (or vice-versa). For example, *Delete_Domain(participants,Event)* and *Change_Domain(participants,Onset,Event)* are conflicting, because the application of the former deletes (participants,domain,Event) whereas the latter adds the same triple. A set $C \subseteq \mathcal{L}$ is called *conflicting* iff C contains any pair of conflicting changes. We can show that changes detected between versions are not conflicting:

THEOREM 6.4. *The set of detectable high-level changes of any pair of RDF(S) KBs is not conflicting.*

The second related notion, independence, has to do with the fact that the computation of $\alpha^+_M(\mu,C,V), \alpha^-_M(\mu,C,V)$ is generally, but not always, affected by the changes $c \in C$ (cf. the examples in Figures 1, 3). Formally, c_1, c_2 are *independent* with respect to V iff $\alpha^+_M(\mu,\{c_1,c_2\},V) = \alpha^+_M(\mu,\{c_i\},V)$ and $\alpha^-_M(\mu,\{c_1,c_2\},V) = \alpha^-_M(\mu,\{c_i\},V)$, for $i = 1,2$ and for all $\mu \in \mathcal{M}(c_i)$. A set of changes C is called *independent* iff c_1, c_2 are independent for all $c_1, c_2 \in C$. Note that for any basic or composite change c, $\mathcal{M}(c) = \emptyset$, so a set of changes consisting only of basic and composite changes is independent.

It is trivial to see that changes that are not independent may give different results in sequential application, depending on the application order. The same is true for conflicting changes. These are the only two cases where application order matters. This is shown in Theorem 6.5 below, which states that, for non-conflicting and independent sets of changes, bulk and sequential application (under any order) give the same result.

Using Theorem 6.4, we can also deduce that detected changes between two versions can be applied sequentially (in any order) iff they are independent (Corollary 6.6).

THEOREM 6.5. *Consider an RDF(S) KB V and a set of changes $C = \{c_1, \ldots, c_n\}$, such that C is non-conflicting and independent. Then, $(\ldots((V \bullet \{c_{\pi(1)}\}) \bullet \{c_{\pi(2)}\}) \bullet \ldots) \bullet \{c_{\pi(n)}\} = V \bullet C$ for any permutation π over the set of indices $\{1, \ldots, n\}$.*

COROLLARY 6.6. *Consider two RDF(S) KBs V_1, V_2 and the corresponding set of detectable changes C. If C is independent then $(\ldots((V_1 \bullet \{c_1\}) \bullet \{c_2\}) \bullet \ldots) \bullet \{c_n\} = V_1 \bullet C = V_2$ for any given order $c_1, c_2, \ldots, c_n$ of elements in C.*

Even in cases where sequential application does not give the same results as the bulk one, it may be the case that a set of changes C can be decomposed in two or more smaller sets (e.g., C_1, C_2) such that C_1, C_2 can be applied separately and give the same result as the bulk application of C. This eventually allows us to separate the changes in C into several groups, such that each change is independent of and non-conflicting with the changes in other groups, and perform a "partially sequential" application.

COROLLARY 6.7. *Consider an RDF(S) KB V and two sets of changes C_1, C_2 such that for all $c_1 \in C_1$, $c_2 \in C_2$, c_1, c_2 are non-conflicting and independent. Then $(V \bullet C_1) \bullet C_2 = (V \bullet C_2) \bullet C_1 = V \bullet (C_1 \cup C_2)$.*

6.3. Traversing the History of Versions

Consider now the case where several subsequent versions of a KB $V_1, \ldots, V_n$ are preserved in a multi-version repository. Using deltas, we can decide to store only the first (oldest) version (V_1), along with the high-level deltas C_i that lead from V_i to V_{i+1}. Then a version V_k can be restored by applying $C_1, C_2, \ldots, C_{k-1}$, in this order, allowing us to move forwards in the history of versions. Alternatively, we may decide to store only the last (most recent) version (V_n) and the deltas that led to it. Then, to restore V_{n-1}, we should be able to generate the *reverse* of C_{n-1}, i.e., a set C'_{n-1} such that $V_n \bullet C'_{n-1} = V_{n-1}$. Note that this implies $(V_{n-1} \bullet C_{n-1}) \bullet C'_{n-1} = V_{n-1}$, i.e., C'_{n-1} "cancels" the effects of C_{n-1}. More generally, to revert to any previous version V_k, we need to find the reverse of C_i, $i = k, \ldots, n-1$ (say C'_i) and apply them in the reverse order: $C'_{n-1}, \ldots, C'_k$. This allows us to move *backwards* in versions.

In the sequel, we will clarify the notion of reversibility and verify that we can find reverses for all deltas in our language; thus, we can move both backwards and forwards in version history, given any single version and the deltas that led to/from it. We start by considering reversibility of single changes.

Definition 6.8. A change c_1 is called the *reverse* of c_2 iff $\delta^+(c_1) = \delta^-(c_2)$, $\delta^-(c_1) = \delta^+(c_2)$ and $\langle S_1, S_2 \rangle \in \mathcal{M}(c_1)$ iff $\langle S_2, S_1 \rangle \in \mathcal{M}(c_2)$.

THEOREM 6.9. *Every change in $\mathcal{L}$ has a unique reverse.*

THEOREM 6.10. *If c_2 is the reverse of c_1, then c_1 is the reverse of c_2 and c_1, c_2 are conflicting.*

For example, the reverse of *Change_Domain(participants,Onset,Event)* is *Change_Domain(participants,Event,Onset)*, and vice-versa. We will denote by c^{-1} the reverse of $c \in \mathcal{L}$. Theorem 6.11 shows that the consumed triples of c, c^{-1} are related:

THEOREM 6.11. *For any two RDF(S) KBs V_1, V_2 and change c, the following are true: $\xi^+(c^{-1}, V_2, V_1) = \xi^-(c, V_1, V_2)$ and $\xi^-(c^{-1}, V_2, V_1) = \xi^+(c, V_1, V_2)$.*

THEOREM 6.12. *Consider two RDF(S) KBs V_1, V_2 and their corresponding set of detectable changes C. Set $C^{-1} = \{c^{-1} \mid c \in C\}$. Then C^{-1} is non-conflicting and $V_2 \bullet C^{-1} = V_1$.*

Theorem 6.12 shows that a set of changes performed upon an RDF(S) KB can be canceled by applying the reverse upon the result. This allows for both "undoing" unwanted changes, and reproducing older versions of an RDF(S) KB based on the newest version and the deltas that led to it.

7. ALGORITHMS

7.1. Detection Algorithm

Algorithm 1 presents our core detection algorithm. Detection can take place in any order (due to our deterministic semantics), but Algorithm 1 considers heuristic changes at a first step (lines 4-6) and then deals with basic and composite changes (lines 7-26); details on those steps can be found in the subsections below.

Note that the symbol $\Sigma(c)$ used in line 22 of Algorithm 1 refers to the basic changes that are *subsumed* by the composite change c, i.e., those basic changes whose detection requirements (δ^+, δ^-, ϕ) are more specific than the corresponding requirements for c; more details on this concept and its importance for the change detection algorithm are given in Subsection 7.3 below.

ALGORITHM 1: ChangeDetectionAlgorithm$(V_1, V_2, \mathcal{M})$

```
1: basic_changes := ∅; composite_changes := ∅; heuristic_changes := ∅
2: Δ+ := V2 \ V1; Δ− := V1 \ V2
3: Δ := ⟨Δ+, Δ−⟩
4: if M ≠ ∅ then
5:    heuristic_changes := findHeuristicChanges(Δ, M)
6: end if
7: for all low-level changes t in Δ+, Δ− do
8:    potBC := findPotentialChanges(t, Δ, true)
9:    for all b ∈ potBC do
10:      if φ(b) = true then
11:         basic_changes := basic_changes ∪ {b}
12:         Δ+ := (Δ+ \ δ+(b)), Δ− := (Δ− \ δ−(b))
13:         break
14:      end if
15:   end for
16: end for
17: for all basic changes b ∈ basic_changes do
18:    potCC := findPotentialChanges(b, basic_changes, false)
19:    for all c ∈ potCC do
20:       if φ(c) = true then
21:          composite_changes := composite_changes ∪ {c}
22:          basic_changes := basic_changes \ Σ(c)
23:          break
24:       end if
25:    end for
26: end for
27: return basic_changes ∪ composite_changes ∪ heuristic_changes
```

7.2. Detection of Heuristic changes

In order to detect heuristic changes we rely on a mapping $\mathcal{M}$ which is an input to the algorithm. This mapping is produced by a matcher [Euzenat and Shvaiko 2007; Ferrara et al. 2011; Flouris et al. 2008], which is used to identify terminological changes. Producing the mapping is not an integral part of our detection algorithm, but constitutes an optional, pre-processing phase, which is out of the scope of our work. Any custom-made or off-the-shelf matcher could be used for producing the mappings[14], or the curator may choose to exclude heuristic changes from the detection process altogether by omitting the matching process and setting $\mathcal{M} = \emptyset$.

ALGORITHM 2: findHeuristicChanges(Δ,$\mathcal{M}$)

```
1: heuristic := ∅
2: for all μ = {A_1, ..., A_n} ⇝ {B_1, ..., B_m} ∈ M do
3:   if n = 1 then
4:     if m = 1 then
5:       if A_1 and B_1 are classes then
6:         c := Rename_Class(A_1, B_1)
7:       end if
8:       if A_1 and B_1 are labels then
9:         c := Change_Label(A_1, B_1)
10:      end if
11:    else {if m > 1}
12:      if A_1 and B_i(i = 1, ..., m) are classes then
13:        if A_1 ∈ {B_1, ..., B_m} then
14:          c := Split_Class_into_existing(A_1, {B_1, ..., B_m})
15:        else
16:          c := Split_Class(A_1, {B_1, ..., B_m})
17:        end if
18:      end if
19:    end if
20:  else {if n > 1}
21:    if A_i(i = 1, ..., n) and B_1 are classes then
22:      if B_1 ∈ {A_1, ..., A_n} then
23:        c := Merge_Classes_into_existing({A_1, ..., A_n}, B_1)
24:      else
25:        c := Merge_Classes({A_1, ..., A_n}, B_1)
26:      end if
27:    end if
28:  end if
29:  ⟨toConsume+, toConsume−⟩ := findMapConsumeTriples(Δ, μ, c)
30:  toConsume+ := toConsume+ ∪ δ+(c), toConsume− := toConsume− ∪ δ−(c)
31:  heuristic := heuristic ∪ {c}
32:  Δ+ := Δ+ \ toConsume+, Δ− := Δ− \ toConsume−
33: end for
34: return heuristic
```

Given a mapping $\mathcal{M}$, the detection algorithm for heuristic changes is shown in Algorithm 2. The idea is to use a series of checks (lines 2-5, 8, 12-13, 21-22) to identify the form of each $\mu \in \mathcal{M}$ (see Definition 3.1), as well as the type of the URIs it involves (classes, metaclasses etc.). The results of these checks determine the detectable heuristic change. For simplicity, in Algorithm 2 we only present the tests for URIs/literals

[14] See [Ferrara et al. 2011] and `www.ontologymatching.org` for details on state-of-the-art matchers.

which are of type class or label, but the algorithm can easily be generalized to apply to all types by adding "IF" blocks similar to those in lines 5-7, 12-19 and 21-27 for properties, individuals, metaclasses and metaproperties, and a block similar to the block in 8-10 for comments.

Note that we need not check whether the triples in δ^+, δ^- are in Δ: by the mapping we can be certain that the required added/deleted triples for each heuristic change will be in Δ. The last steps of the algorithm are the addition of the detectable heuristic change to the set of detectable heuristic changes $heuristic$ and the consumption of the corresponding triples (lines 29-32). To compute the consumed triples, we rely on another routine, called $findMapConsumeTriples$ (see Algorithm 3), which essentially computes ξ^+_M, ξ^-_M.

ALGORITHM 3: findMapConsumeTriples(Δ,μ,c)

1: $toConsume^+ := \emptyset$, $toConsume^- := \emptyset$
2: Assume that $\mu = \{A_1, \ldots, A_n\} \rightsquigarrow \{B_1, \ldots, B_m\}$
3: **if** $n = 1$ **then**
4: **for all** triples $t \in \Delta^- \setminus \delta^-(c)$ containing A_1 **do**
5: $T := \{t'$: triples obtained by replacing A_1 with B_i $(t \rightsquigarrow t')\}$
6: **if** $T \subseteq \Delta^+$ and $T \cap \delta^+(c) = \emptyset$ **then**
7: $toConsume^+ := toConsume^+ \cup T$, $toConsume^- := toConsume^- \cup \{t\}$
8: **end if**
9: **end for**
10: **else**
11: **for all** triples $t \in \Delta^+ \setminus \delta^+(c)$ containing B_1 **do**
12: $T := \{t'$: triples obtained by replacing B_1 with A_i $(t' \rightsquigarrow t)\}$
13: **if** $T \subseteq \Delta^-$ and $T \cap \delta^-(c) = \emptyset$ **then**
14: $toConsume^+ := toConsume^+ \cup \{t\}$, $toConsume^- := toConsume^- \cup T$
15: **end if**
16: **end for**
17: **end if**
18: $toConsume := \langle toConsume^+, toConsume^- \rangle$
19: **return** $toConsume$

More specifically, $findMapConsumeTriples$ takes in its input the delta (Δ), some mapping (μ) and the change (c) that corresponds to it. Depending on the form of the mapping (one-to-one, one-to-many, many-to-one), the algorithm scans Δ^- (or Δ^+) for "relevant" triples, i.e., triples containing a URI involved in the mapping (lines 4 and 11). For each such triple, the algorithm checks whether all the corresponding triples (through μ) exist in Δ^+ (or Δ^-), and, if so, the corresponding triples are added in the variables $toConsume^+, toConsume^-$ (lines 5-8 and 12-15). The pair $\langle toConsume^+, toConsume^- \rangle$, which essentially contains the triples in ξ^+_M, ξ^-_M, is returned.

7.3. Detection of Basic and Composite Changes

For the detection of basic and composite changes, we will exploit one particular property of our language, namely that each composite change *subsumes* certain basic changes. Take for example the composite change c =*Generalize_Domain(a,*b_1*,*b_2*)* and the basic change c' =*Delete_Domain(a,*b_1*)*. As can be seen by the definition of these changes in Appendix B, the low-level changes required for the detection of c are a superset of those required for c' (i.e., $\delta^+(c') \subseteq \delta^+(c), \delta^-(c') \subseteq \delta^-(c)$) and the conditions of c ($\phi(c)$) are more restrictive than those of c' ($\phi(c')$). Thus, given that $\mathcal{M}(c) = \mathcal{M}(c') = \emptyset$, whenever c is detectable, c' would have been detectable too, had it not been for the

prioritization of composite over basic changes. This is a more general phenomenon, i.e., for every composite change c there are some basic changes c'_i which are *subsumed* by c (i.e., $\delta^+(c'_i) \subseteq \delta^+(c), \delta^-(c'_i) \subseteq \delta^-(c)$ and $\phi(c)$ is more restrictive than $\phi(c'_i)$); the latter are denoted by $\Sigma(c)$ (or simply Σ when c is obvious from the context). The tables at Appendix B, show the basic changes that are subsumed by each composite change. Using Σ, we can establish an interesting property of our language, namely that for any detectable composite change, e.g., c, the low-level changes consumed by c are exactly the same as those consumed by all changes in $\Sigma(c)$ combined:

THEOREM 7.1. *For any composite change c the following are true:* $\cup_{b\in\Sigma(c)}\delta^+(b) = \delta^+(c)$ *and* $\cup_{b\in\Sigma(c)}\delta^-(b) = \delta^-(c)$.

Theorem 7.1 essentially allows the association of groups of basic changes with composite ones (via subsumption), in the same way as groups of low-level changes are associated with (partitioned into) high-level ones. Thus, we can break the detection process of basic and composite changes in two parts. In the first pass, we ignore composite changes and compute the detectable basic changes (lines 7-16 in Algorithm 1). Note that some of those basic changes would not have been detectable if composite changes were considered, because composite changes take priority; such changes are identified in the second step (lines 17-26). Using Theorem 7.1, we know that in order for a composite change to be detectable, all its subsumed basic changes must have been detected in the first pass. Therefore, we can detect composite changes based on the basic changes identified in the first pass, and remove the corresponding basic changes from our list of detected changes. The same technique that is used to "group" low-level changes to form basic ones, is used also to "group" basic changes to form composite ones.

To speed up this "grouping", we note that each low-level change can be potentially associated to (consumed by) a small and predefined set of basic changes. For instance, the low-level change (participants,domain,Onset) $\in \Delta^-$ cannot trigger the detection of any change of the form *Delete_Superclass(a,b)*, because no low-level change of the form (participants,domain,Onset) appears in the required deleted triples of *Delete_Superclass(a,b)*. Similarly, each basic change is subsumed by a small and predefined set of composite changes. Thus, in both cases, a first filtering can be made even at design time (e.g., through a lookup table).

This filtering is performed in method $findPotentialChanges$ (Algorithm 4). This method is called for a specific change $change$, which can be either a low-level change (to be consumed by a basic one) or a basic change (to be subsumed by a composite one). The two cases are discriminated using the flag $changeIsBasic$. Let's concentrate first on the filtering of basic changes (where $changeIsBasic = true$). In that case, the parameter $set\_of\_changes$ contains the changes in Δ.

As mentioned above, we can easily identify the basic changes that can consume said low-level change at design-time. For each such change bc, we identify the low-level changes in $\delta^+(bc) \cup \delta^-(bc)$ (line 4) and check whether they are all contained in $set\_of\_changes$ (lines 9-14). If this is not true, then bc cannot be detectable, so it is not included in pot; otherwise, bc is added in pot (line 16), and will be detectable if and only if (a) $\phi(bc)$ is true, and, (b) bc is not subsumed by some detectable composite change. These two checks are performed in Algorithm 1 and are not discussed here.

In the case where $changeIsBasic$ is false (i.e., when we filter composite changes), the parameter $set\_of\_changes$ contains the basic changes that have been identified as detectable (barring their possible subsumption by some detectable composite change). The same steps are performed, except that now we check whether all the basic changes in $\Sigma(bc)$ are in $set\_of\_changes$ (line 6).

ALGORITHM 4: findPotentialChanges($change, set\_of\_changes, changeIsBasic$)

```
1: pot := ∅
2: for all changes bc whose detection is possibly triggered by change do
3:    if changeIsBasic = true then
4:       toCheck := set of low-level changes in δ+(bc) ∪ δ−(bc)
5:    else
6:       toCheck := set of basic changes in Σ(bc)
7:    end if
8:    flag := true
9:    for all change0 ∈ toCheck do
10:      if change0 does not appear in set_of_changes then
11:         flag := false
12:         break
13:      end if
14:   end for
15:   if flag = true then
16:      pot := pot ∪ {bc}
17:   end if
18: end for
19: return pot
```

Now let's go back to Algorithm 1 to see how these are used. The first step for the detection of basic changes is to pick a low-level change (i.e., a triple in Δ) in line 7 of Algorithm 1. Then, we perform a first "filtering" of the basic changes that need to be considered using $findPotentialChanges$; note that here, $findPotentialChanges$ is called using as parameters a low-level change t, the low-level delta Δ and $true$ to indicate that we want to filter basic changes.

As already mentioned, the basic changes returned by $findPotentialChanges$ are not necessarily detectable; each of them represents one basic change whose detection *could* be triggered by the existence of the low-level change selected in line 7. So we loop over all changes in $potBC$ (line 9) and check whether their conditions are true (line 10). If the check succeeds for any such change, no further checks need to be made, as we know (by Theorem 5.6) that no other detectable change could be associated with the low-level change under question. For this reason, we add the change to the set of detectable basic changes $basic\_changes$, we remove from Δ (in line 12) all the low-level changes that were associated with said change (so they are no longer relevant for future detections), and stop the process of checking (line 13). Note that the changes in $basic\_changes$ may still be overridden by detectable composite changes, identified later in the algorithm (second pass); nevertheless, the consumption process can proceed normally, thanks to Theorem 7.1. Once we have iterated over the whole Δ, the set $basic\_changes$ will contain all detectable basic changes, plus some basic changes that would have been detectable if we didn't consider (and prioritize) composite changes.

Then, we proceed to the second step of Algorithm 1 (lines 17-26), where composite changes are detected. As in the case of basic changes, we find those composite changes c whose set of subsumed changes Σ consists of basic changes that have been detected in the first pass using $findPotentialChanges$ (line 18); note that this time the parameters of $findPotentialChanges$ are a basic change b, the set of previously detected basic changes $basic\_changes$ and $false$, to indicate that we look to filter composite changes. Then, in line 20 of Algorithm 1, the corresponding conditions of each identified composite change are checked to determine whether it is detectable or not, and, in line 22, the basic changes in Σ are removed from $basic\_changes$, as the composite change takes priority over them. Given that the required added/deleted triples of changes in Σ

combined are the same as the required added/deleted triples of c (by Theorem 7.1), this would give the correct results. At the end of the process, the composite and remaining basic changes (the ones that have not been subsumed) are returned along with the detected heuristic changes (line 27).

7.4. Detection Algorithm: Correctness and Complexity

The following theorems establish the soundness and completeness of the presented algorithm with respect to $\mathcal{L}$, as well as its computational complexity:

THEOREM 7.2. *A change c will be returned by Algorithm 1 with input $V_1, V_2, \mathcal{M}$ iff c is detectable.*

THEOREM 7.3. *The complexity of Algorithm 1 is $O(N^2)$, where N is the total size of the input ($V_1, V_2, \mathcal{M}$).*

The reason for the above complexity is that the evaluation of the conditions (ϕ) of certain composite changes (e.g., *Pull_up_Class*(a, B, C)) in line 20 of Algorithm 1 requires time quadratic over the size of the operation's parameters (i.e., B, C in our example). In the worst case scenario, B, C could contain a number of classes analogous to the sizes of the input versions (V_1, V_2); this results in the aforementioned complexity. However, in practice, said size will be much smaller. In addition, not all operations require a quadratic time for evaluating their conditions. As a result, in practice, the complexity is dominated by the preprocessing cost of sorting the triples in the two input versions ($O(N \cdot \log N)$), which is necessary for finding Δ and for speeding up certain searches on V_1, V_2. The above analysis will be verified by our experimental evaluation in Section 8, which will show that the processing time of computing the low-level changes (which includes sorting the triples in V_1, V_2) is much larger than the time required to compute the high-level ones. It should be also noted that the time required to compute the mapping is not considered in Theorem 7.3, because, as mentioned before, the matching process is external to the detection algorithm, which assumes the mapping to be given as input.

7.5. Application Algorithm

An algorithm for applying a set of changes upon an RDF(S) KB is also of interest. Such an algorithm is quite simple and shown in Algorithm 5. Given a set of changes C and a version V, the algorithm computes $\alpha^+(C, V), \alpha^-(C, V)$ and applies it to V (line 13). The correctness of the algorithm can be easily established, once we note that T^- will contain all the triples t'' for which there is a triple t' such that $t \rightsquigarrow t'$, $t'' \rightsquigarrow t'$. The worst-case scenario for the algorithm's complexity appears when a split operator with the mapping $u \rightsquigarrow \{u_1, \ldots, u_n\}$ appears in C, and V contains many triples of the form (u, P_i, u) for some individual u and properties $P_1, \ldots, P_m$. In this scenario, $O(m \cdot n^2)$ new triples will be added; given that m can be $O(N_V)$ and n can be $O(N_C)$ (where N_V, N_C are the sizes of V, C respectively), the application of C upon V could result in the addition of as many as $O(N_V \cdot N_C^2)$ triples. Some searches that line 7 requires for each of the applied triples lead to the complexity of Theorem 7.5. Formally:

THEOREM 7.4. *The output of Algorithm 5 with input C, V is $V \bullet C$.*

THEOREM 7.5. *The complexity of Algorithm 5 for input C, V is $O(N_V \cdot N_C^2 \cdot (\log N_C + \log N_V))$, where N_C, N_V are the sizes of C, V respectively.*

Note that, as expected, Theorem 7.5 shows that sequential application (where N_C is smaller) is more efficient than bulk application.

ALGORITHM 5: Apply(C,V)

1: $apply^+ := \bigcup_{c \in C} \delta^+(c)$, $apply^- := \bigcup_{c \in C} \delta^-(c)$, $applyM^+ := \emptyset$, $applyM^- := \emptyset$
2: **for all** $c \in C$ **do**
3: **for all** $\{A_1, \ldots, A_n\} \rightsquigarrow \{B_1, \ldots, B_m\}$ in $\mathcal{M}(c)$ **do**
4: **for all** triples $t \in V$ containing A_i **do**
5: $T^+ := \{t'$: triples obtained by replacing A_i with B_j in t $(t \rightsquigarrow t')\}$
6: $T^- := \{t''$: triples obtained by replacing A_i with A_j in t $(t'' \rightsquigarrow t')\}$
7: **if** $T^+ \cap apply^+ = \emptyset$ and $T^- \cap apply^- = \emptyset$ and $T^- \subseteq V$ **then**
8: $applyM^+ := applyM^+ \cup T^+$, $applyM^- := applyM^- \cup T^-$
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: **return** $((V \setminus apply^-) \setminus applyM^-) \cup apply^+ \cup applyM^+$

8. EXPERIMENTAL EVALUATION

8.1. Evaluation Aims

The evaluation of our approach is based on experiments performed on three well-established ontologies from the cultural (CIDOC Conceptual Reference Model – CIDOC-CRM [CIDOC 2010]), biological (Gene Ontology – GO [Hill et al. 2008]) and musical (Music Ontology – MO [Raimond et al. 2010]) domains.

For the purposes of evaluation, and in order to produce the mappings required as input for our detection algorithm, we implemented a simple matcher which associates resources (URIs) based on the similarity of their neighborhoods. If the similarity exceeds a certain threshold then a mapping is reported. A similar string matcher was implemented for literals: when, for example, a resource is associated with a different comment in V_1, V_2, this could be either because the old comment was deleted and a new one was added, or because the old comment was edited. To discriminate between the two cases, we used the Levenshtein string distance metric [Levenshtein 1966] which compares the similarity of the respective comments and determines whether a pair of *Delete_Comment-Add_Comment* or a single *Change_Comment* operation should be returned (similarly for labels). In the following, we will refer to this matcher as the *neighborhood matcher*.

The aims of our evaluation are the following:

— To show the *intuitiveness* of $\mathcal{L}$. In particular, we will show, first, that the defined changes are being detected in real-world cases (see Tables VII, IX, X, and Figure 4), so they occur frequently in real-world scenarios, and, second, that the reported changes are close to the editors' perception (see our analysis on the results of CIDOC and GO ontologies); this implies that the changes commonly employed by editors are captured by our framework. This analysis will also show that even the most careful manual recording of changes may lead to errors and omissions, as, e.g., in the case of CIDOC (see Table VIII).
— To evaluate the *effect of the matcher* on the detected changes. The matcher is an important factor, because the quality of the reported mapping affects the quality of the detection result. To perform this evaluation, we exploited CIDOC's naming policy for URIs, which attaches a unique, change preserving ID in the used names; this allowed us to develop a special-purpose matcher whose accuracy (precision and recall) was 100%. We compared the results of the detection process for CIDOC using the accurate matcher, as opposed to the results obtained using the neighborhood matcher (whose mappings are inaccurate). This analysis showed that, even though

the reported mappings affect directly only the detected heuristic changes, they also have an important cascading effect on the detected basic/composite changes. This is due to the fact that different heuristic changes would consume different changes from Δ, leaving a different set of low-level changes to be used for the detection of basic and composite changes.

— To show the *conciseness* of composite and heuristic changes (Table XII), i.e., to quantify the effect of using composite and heuristic changes on the delta size.

— To measure the *performance* of the detection process in real settings (Table XIII) and identify the main factors that affect performance in practice.

Note that, out of the three selected RDF(S) KBs (CIDOC, GO, MO), only MO contains changes related to the data level, and only GO contains changes related to the meta-level. In fact, the use of the meta-level is rather limited in real-world ontologies, and, even in ontologies containing meta-level constructs, the meta-level is fairly static in practice. On the other hand, changes at the data level are fairly simple, compared to changes at the schema level. For these reasons, our evaluation gives more emphasis on the schema level changes, which are more useful in practice and exhibit greater variety and complexity.

8.2. Intuitiveness

The CIDOC-CRM project provides definitions and a formal structure for describing the explicit and implicit concepts and relationships used in cultural heritage documentation. CIDOC ontology consists of nearly 80 classes and 250 properties, but has no instances. CIDOC was selected for use in our experiments because of its wide use in the cultural domain. On the practical side, it puts emphasis on properties, unlike the other ontologies used (GO and MO), allowing us to test property-related changes. In addi-

Table VII. CIDOC: Overview of High-level Changes Between Versions v.3.2.1 - v.5.0.1

Operation	Type	Frequency
Delete_Superproperty	Basic	58
Delete_Superclass_From_Class	Basic	65
Add_Superclass_To_Class	Basic	67
Add_Superproperty	Basic	82
Delete_Comment	Basic	91
Add_Range	Basic	150
Delete_Range	Basic	150
Add_Domain	Basic	158
Delete_Domain	Basic	158
Add_Comment	Basic	167
Add_Label	Basic	253
Ungroup_Classes	Composite	1
Change_To_Datatype_Property	Composite	2
Pull_down_Class	Composite	2
Ungroup_Properties	Composite	2
Change_Superproperties	Composite	3
Change_Domain	Composite	3
Change_Range	Composite	4
Delete_Class	Composite	4
Add_Class	Composite	12
Generalize_Domain	Composite	22
Generalize_Range	Composite	22
Delete_Property	Composite	22
Add_Property	Composite	77
Rename_Class	Heuristic	70
Rename_Property	Heuristic	183

tion, CIDOC's editors are (manually) recording the changes they perform, and every version is accompanied by release notes describing in natural language the differences with the previous version; this allowed us to verify that the changes detected by our algorithm are close to the editors' perception. Moreover, the aforementioned naming policy of CIDOC allowed us to develop a special-purpose matcher whose accuracy (precision and recall) was 100%, thereby avoiding any pitfalls related to poor mappings. 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 11.2009), which are available in [CIDOC 2010].

The basic changes detected in CIDOC mostly concern properties (as expected) and include several additions of comments and labels. Detected composite changes include additions/deletions of properties as well as changes in the properties' domains/ranges. Finally, the majority of the detected heuristic changes involved the *Rename_Property* operation (see Table VII).

Figure 4(a) shows the distribution of basic, composite and heuristic changes in the deltas computed between different versions of CIDOC. We observe that the high-level deltas consist mostly of basic changes except for the pair v3.3.2-v3.4.9 where our algorithm detected mainly heuristic changes. For the pair v3.4.9-v4.2 the detected basic changes comprise over 90% of the total detected changes caused by the large number of changes in labels and comments, where the corresponding string literals were unmatched. In the same manner, for the pair v4.2-v5.0.1 the detected basic changes comprise over 80% of the high-level delta. The reason for the large percentage of basic changes in these cases is that almost all classes and properties have been renamed, and, as a consequence of Definition 5.2, triples which are involved in more than one mappings (e.g., $(A$, subClassOf, $B)$ where both A, B are renamed) are not consumed by any heuristic change; thus, they are reported as basic changes (addition and deletion of a subsumption, in the above case).

We compared our detected changes against CIDOC's release notes [CIDOC 2010] and found that our results mostly matched the changes described in the release notes. It is interesting here to note that the unmatched changes were found to be typos, omissions and other mistakes in CIDOC's release notes, rather than in our detection process, as also verified by Martin Doerr, one of CIDOC's editors. The differences between the changes described in the release notes and those detected by our algorithm for versions v3.2.1-v3.3.2 can be seen in Table VIII. These findings highlight the need for automated change detection algorithms, as they show that even the most careful manual change recording process may contain mistakes.

Our second case study involved GO [Hill et al. 2008], whose size and update rate make it one of the largest and most representative ontologies for studying problems related to change management in ontologies. GO describes 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, one meta-class, and 1350 property instances of a single property ("obsolete") which is, sometimes, used by the GO editors to mark classes as obsolete as an alternative to deleting them. GO was selected primarily due to its size, which allowed us to experiment on a large KB.

Table VIII. Differences Between Release Notes and Results of Algorithm (for CIDOC)

	Editor Notes	**Change Detection Algorithm**
Delete Class	3	6
Add Property	54	58
Delete Property	16	18
Rename Property	24	31
Changes in Property's Range	14	17

Table IX. GO: Overview of High-level Changes Between Versions v.25.11.2008 - v.20.04.2010

Operation	Type	Frequency
Add_Type_To_Class	Basic	17
Delete_Type_From_Class	Basic	18
Add_Property_Instance	Basic	74
Delete_Comment	Basic	254
Delete_Superclass_From_Class	Basic	334
Add_Superclass_To_Class	Basic	467
Delete_Label	Basic	706
Add_Comment	Basic	945
Add_Label	Basic	1401
Delete_Class	Composite	115
Move_Class	Composite	127
Pull_up_Class	Composite	139
Change_Superclasses	Composite	149
Ungroup_Classes	Composite	173
Group_Classes	Composite	212
Pull_down_Class	Composite	460
Add_Class	Composite	4167
Split_Class_Into_Existing	Heuristic	15
Merge_Classes_Into_Existing	Heuristic	18
Change_Label	Heuristic	923
Change_Comment	Heuristic	4057

Moreover, its emphasis on classes makes it very different in structure from CIDOC and MO, and allows us to test class-related operations. Note that GO also involves changes at the meta-level. 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 [Bairoch et al. 2005], which use RDFS semantics. GO is updated on a daily basis, but UniProt releases a new version every month and only the latest version is available for download[15]. During the time of our experiments we were able to retrieve 8 versions of GO (dated 25.11.08, 16.12.08, 24.03.09, 05.05.09, 26.05.09, 16.06.09, 22.09.09 and 20.04.10).

In our experiment with GO, we used the inaccurate neighborhood matcher. The detected heuristic changes on URIs (namely, *Merge_Classes_Into_Existing* and *Split_Class_Into_Existing*) were very few (1.7% of the total) as shown in Table IX; on the other hand, we detected several *Change_Comment* and *Change_Label* operations. The detected basic changes were mostly additions and deletions of comments, labels and subsumption relationships, whereas the detected composite changes were mostly additions of classes.

From Figure 4(b), we observe that for GO the high-level deltas comprise equally of basic and composite changes, with the number of detected composite changes being slightly larger, whereas the number of heuristic changes vary. 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., [Zhdanova 2008]) show that the detected operations capture the perception of the editors.

The third case study that we experimented with is MO, which provides a vocabulary of concepts and properties for describing a wide range of music-related information. MO consists of 95 classes, 190 properties, 17 individuals and 437 property instances. As MO contains data, it provides us the means to test the expressiveness of our language for changes related to the data layer, which was not the case with the former

[15] `ftp.uniprot.org/pub/databases/uniprot_datafiles_by_format/rdf`

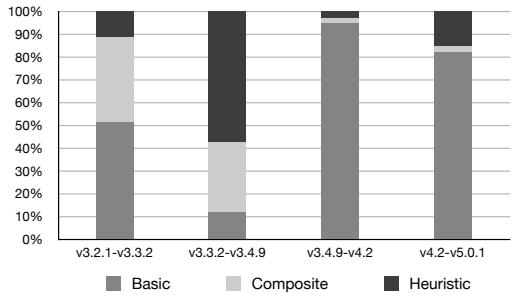
Table X. MO: Overview of High-level Changes Between Versions v12.08.2007 - v13.02.2010

Operation	Type	Frequency
Delete_Domain	Basic	2
Delete_Range	Basic	2
Add_Comment	Basic	3
Add_Superclass_to_Class	Basic	6
Delete_Superclass_from_Class	Basic	6
Delete_Superproperty	Basic	7
Add_Domain	Basic	10
Add_Range	Basic	10
Delete_Comment	Basic	12
Add_Superproperty	Basic	13
Add_Label	Basic	24
Add_Property_Instance	Basic	95
Delete_Property_Instance	Basic	110
Specialize_Range	Composite	1
Move_Property	Composite	1
Pull_up_Class	Composite	1
Delete_Individual	Composite	1
Add_Individual	Composite	2
Generalize_Domain	Composite	3
Generalize_Range	Composite	3
Change_Domain	Composite	5
Delete_Class	Composite	13
Change_Range	Composite	15
Add_Class	Composite	15
Delete_Property	Composite	17
Add_Property	Composite	26
Merge_Classes_into_existing	Heuristic	1
Merge_Properties_into_existing	Heuristic	1
Rename_Class	Heuristic	1
Split_Property	Heuristic	1
Split_Property_into_existing	Heuristic	3
Rename_Property	Heuristic	3
Change_Comment	Heuristic	4
Change_Label	Heuristic	10

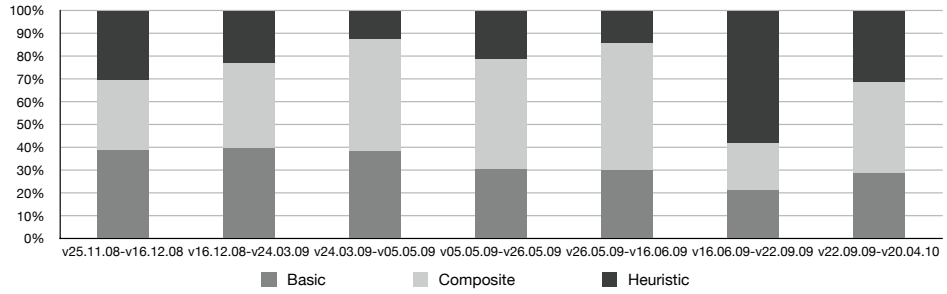
two case studies (CIDOC and GO); note however, that changes in the data layer are simpler than changes in the schema layer. For our experiments, we used 6 versions of MO (dated 12.08.07, 18.09.07, 06.12.07, 28.07.08, 28.10.08, 13.02.10), retrieved from MO's repository[16].

From Table X we can see that most basic changes are related to property instances and involve the operations *Add_Property_Instance* and *Delete_Property_ Instance*; most composite changes are operations that delete and add properties and many of them add/delete classes. The detected heuristic changes (according to the mappings reported by the neighborhood matcher) concerned classes, properties and string literals with the majority of them involving the operation *Change_Label*. We observe here that the detected changes in MO include a larger variety of operations compared to our other test cases. The distribution of changes in the three categories (basic, composite and heuristic) as shown in Figure 4(c) varies between different versions. For the first two pairs of versions (v12.08.07-v18.09.07, v18.09.07-v06.12.07) the number of detected basic changes is approximately equal to the number of detected composite changes whereas for the rest of the pairs the basic changes outnumber the other types of high-

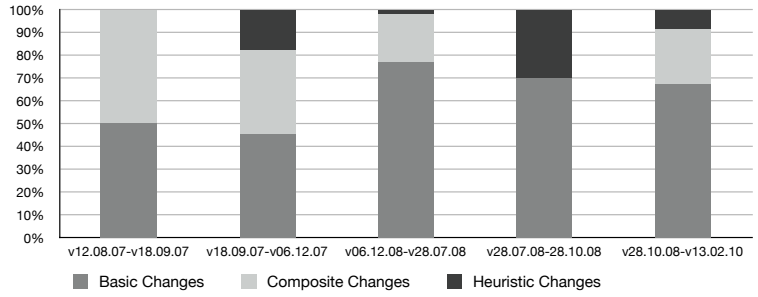
[16] `motools.svn.sourceforge.net/viewvc/motools/mo/rdf/musicontology.rdfs?view=log`



(a) CIDOC



(b) GO



(c) MO

Fig. 4. Distribution of High-level Changes (in CIDOC, GO, MO)

level changes. In these, the majority of the basic changes are changes that add or delete property instances, which were not subsumed by any composite change.

8.3. Effect of the Matcher

Table XI shows the high-level delta computed between two specific versions of CIDOC, namely v.3.2.1-v.3.3.2. These two deltas have been computed using different matchers: the "accurate matcher" exploits CIDOC's naming policy to provide accurate mappings (100% precision and recall), whereas the "inaccurate matcher" is the neighborhood matcher described above.

The effect of the matcher on the heuristic changes is straightforward, because each mapping corresponds to one detected heuristic change. Given this, one initial obser-

Table XI. Effect of the Quality of the Matcher on the Detection Results (CIDOC: v.3.2.1-v.3.3.2)

Operation	Type	Frequency (Accurate)	Frequency (Inaccurate)
Add_Superclass_to_Class	Basic	2	1
Delete_Superclass_from_Class	Basic	3	3
Add_Superproperty	Basic	3	8
Delete_Superproperty	Basic	3	14
Add_Domain	Basic	0	27
Delete_Domain	Basic	0	15
Add_Range	Basic	1	26
Delete_Range	Basic	1	13
Add_Comment	Basic	71	65
Delete_Comment	Basic	92	86
Add_Class	Composite	6	12
Delete_Class	Composite	3	8
Pull_down_Class	Composite	2	1
Add_Property	Composite	58	56
Delete_Property	Composite	18	29
Change_To_Datatype_Property	Composite	2	0
Group_Properties	Composite	4	1
Ungroup_Properties	Composite	4	2
Change_Domain	Composite	0	10
Generalize_Domain	Composite	14	11
Specialize_Domain	Composite	1	0
Change_Range	Composite	1	9
Generalize_Range	Composite	14	9
Specialize_Range	Composite	1	1
Rename_Class	Heuristic	8	2
Merge_Classes_Into_Existing	Heuristic	0	1
Rename_Property	Heuristic	31	4
Split_Property	Heuristic	0	4
Merge_Properties	Heuristic	0	3
Split_Property_Into_Existing	Heuristic	0	9
Merge_Properties_Into_Existing	Heuristic	0	4

vation is that the accurate matcher reports more mappings, which, in turn, lead to more detected heuristic changes (39 versus 27). As seen in Table XI, these mappings (and their corresponding detected heuristic changes) are different in nature, i.e., the inaccurate matcher reported several false positives and several false negatives, so the mappings of the inaccurate matcher are not a subset of those of the accurate one. However, these observations cannot be generalized, as, in principle, it is also possible that an inaccurate matcher would report more mappings than the accurate one (having many false negatives).

The effect of the different mappings on the basic and composite changes is harder to predict. The different heuristic changes consume different low-level changes from Δ, thus leaving a different set of low-level changes to be consumed by the basic and composite changes. As the detected basic and composite changes depend directly on the changes left in the low-level delta after the detection of heuristic changes, the detection results are affected in various ways.

A striking example of this phenomenon in our case appears for classes. The accurate matcher reports 8 *Rename_Class* operations, 6 more than the inaccurate one. Given the lack of mappings to justify the detection of the extra renamings, the inaccurate matcher caused the detection of 5 additional pairs of *Add_Class-Delete_Class* operations. The sixth renaming was perceived as a *Merge_Class_Into_Existing* (due to the false reporting of the corresponding mapping), combined with an additional *Add_Class*;

in effect, the class that was actually renamed was understood as being merged with another class, whereas the new name (of the renamed class) was understood as an entirely new class (and reported as *Add_Class*).

Similar, but more complicated observations can be made for properties. In that case, only 4 out of 31 renamings were detected by the inaccurate matcher, whereas the rest of the renamings were reported using a variety of other operations, both heuristic, and basic/composite. For example, the increased number of detections involving *Split_Property_Into_Existing* and *Merge_Property_Into_Existing* operations explains the increased number of basic changes. In particular, the above heuristic changes involve mappings of the form $S_1 \rightsquigarrow S_2$, where $S_1 \cap S_2 \neq \emptyset$, so, by Lemma A.4 (see Appendix A), they do not consume changes in the neighborhood of the merged/split elements. As a result, several low-level changes remain in Δ and reported as basic changes, increasing their number.

8.4. Conciseness

Table XII shows the number of changes detected between different pairs of CIDOC, GO and MO 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 returned by our algorithm by disabling or enabling the detection of composite and heuristic changes. We note that 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.

More specifically, for CIDOC versions the reduction of the delta when detecting all types of high-level changes ranges from 46% to 79% with the most concise high-level delta being reported for the pair of versions v3.3.2-v3.4.9. For GO the reduction of the delta when detecting all types of high-level changes ranges from 60% to 77% and the most concise high-level delta is reported for versions v26.05.09 - v16.06.09. Finally, for MO the delta is reduced in the range of 33%-69% when all types of high-level changes are detected with the most concise high-level delta (compared to the low-level one) being reported for the pair of versions v18.09.07-v06.12.07. The differences in the rates of size reduction for the delta in the various cases can be explained by looking at the

Table XII. Conciseness Evaluation

Versions	V_1	V_2	Δ	Basic	Basic+Composite+Heuristic
CIDOC					
v3.2.1 - v3.3.2	955	1082	870	834	176 + 128 + 39 = 343
v3.3.2 - v3.4.9	1082	1111	287	285	7 + 18 + 34 = 59
v3.4.9 - v4.2	1111	1256	570	538	293 + 6 + 10 = 309
v4.2 - v5.0.1	1256	3283	4700	2860	1728 + 53 + 323 = 2104
GO					
v25.11.08 - v16.12.08	185961	187237	2978	2260	386 + 304 + 305 = 995
v16.12.08 - v24.03.09	187237	190815	7296	5038	765 + 713 + 452 = 1930
v24.03.09 - v05.05.09	190815	192657	3093	2309	291 + 376 + 99 = 766
v05.05.09 - v26.05.09	192657	193978	2662	1983	214 + 340 + 150 = 704
v26.05.09 - v16.06.09	193978	194470	796	593	54 + 100 + 26 = 180
v16.06.09 - v22.09.09	194470	196583	7480	6935	620 + 616 + 1734 = 2970
v22.09.09 - v20.04.10	196583	211569	30463	22785	2409 + 3347 + 2655 = 8421
MO					
v12.08.07 - v18.09.07	1781	1778	12	12	4 + 4 + 0 = 8
v18.09.07 - v06.12.07	1778	1808	197	194	28 + 23 + 11 = 62
v06.12.07 - v28.07.08	1808	1626	285	272	141 + 38 + 4 = 183
v28.07.08 - v28.10.08	1626	1625	25	23	7 + 0 + 3 = 10
v28.10.08 - v13.02.10	1625	1893	427	392	130 + 46 + 17 = 193

distribution of the different types of high-level changes (basic, composite and heuristic) in the detected high-level deltas (see Figure 4), because basic changes consume 1-3 low-level changes from the low-level delta, whereas composite and heuristic changes usually consume more.

It should be noted here that, even though basic changes reduce the conciseness of the reported deltas, they are necessary to capture subtle changes in versions that cannot be described using a more coarse-grained composite change. Having said that, it is true that one could define a richer set of composite changes in order to further reduce the number of reported basic changes; in this work, we opted for this particular set of composite changes, because we believe that introducing further (and artificial) changes would reduce the intuitiveness of the language.

8.5. Performance

Table XIII 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 and 1.5GB of main memory. The major observation that is immediately obvious from the table is that the time required for the computation of the low-level Δ (second column) is much larger than the time required for the computation of basic, composite and heuristic changes combined. More specifically, the computation of Δ takes more than 90% of the total time in all but three cases, namely v3.2.1 - v3.3.2 (CIDOC), v4.2 - v5.0.1 (CIDOC) and v22.09.09 - v20.04.10 (GO), where the corresponding percentages are approximately 88%, 68% and 82% respectively. Thus, our first conclusion is that the overhead required for computing a high-level delta (rather than a low-level one) is very small compared to the advantages (conciseness, usefulness etc.) of high-level deltas over low-level ones.

Regarding the computation time itself, we note that the time required for finding the Δ alone is basically analogous to the quantity $N_V \cdot \log N_V$ where N_V is the size of the largest of the two input versions. Moreover, the computation time for the basic changes is largely analogous to the size of Δ. For composite and heuristic changes no such strong correlation can be found. As far as composite changes are concerned, the reason is that the computation time largely depends on the number and type of conditions (ϕ) that need to be checked in line 20 of Algorithm 1, which, in turn, depends

Table XIII. Performance Evaluation

Versions	Δ	Basic	Composite	Heuristic	Total
CIDOC					
v3.2.1 - v3.3.2	132.58 ms	2.83 ms	5.33 ms	9.41 ms	150.15 ms
v3.3.2 - v3.4.9	76.42 ms	0.39 ms	1.18 ms	4.00 ms	81.99 ms
v3.4.9 - v4.2	98.73 ms	1.50 ms	0.58 ms	5.70 ms	106.51 ms
v4.2 - v5.0.1	411.67 ms	16.90 ms	17.07 ms	158.80 ms	604.44 ms
GO					
v25.11.08 - v16.12.08	26170.17 ms	16.79 ms	29.24 ms	89.48 ms	26305.68 ms
v16.12.08 - v24.03.09	27373.40 ms	42.56 ms	55.46 ms	251.28 ms	27722.70 ms
v24.03.09 - v05.05.09	27431.95 ms	15.93 ms	32.70 ms	28.06 ms	27508.64 ms
v05.05.09 - v26.05.09	27960.82 ms	12.91 ms	32.56 ms	36.86 ms	28043.15 ms
v26.05.09 - v16.06.09	28139.97 ms	3.67 ms	7.09 ms	3.04 ms	28153.77 ms
v16.06.09 - v22.09.09	29129.17 ms	27.43 ms	45.54 ms	771.00 ms	29973.14 ms
v22.09.09 - v20.04.10	32983.74 ms	161.90 ms	251.70 ms	7101.98 ms	40499.32 ms
MO					
v12.08.07 - v18.09.07	123.12 ms	0.16 ms	0.21 ms	0.01 ms	123.50 ms
v18.09.07 - v06.12.07	130.19 ms	0.48 ms	0.71 ms	1.81 ms	133.19 ms
v06.12.07 - v28.07.08	154.82 ms	2.13 ms	1.77 ms	0.36 ms	159.08 ms
v28.07.08 - v28.10.08	112.70 ms	0.09 ms	0.06 ms	0.25 ms	113.10 ms
v28.10.08 - v13.02.10	130.53 ms	2.36 ms	2.83 ms	1.83 ms	137.55 ms

on the changes that are put in $potCC$ (line 18 of Algorithm 1) for further evaluation. As an example, when 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 (detection of composite changes), despite the increase of the input size and the Δ size (see Table XII). This is due to the difference in the number of detected composite changes, as in the first pair the composite changes make up 30% of the high-level delta, whereas in the second they constitute only 0.1% of the total detected changes (cf. Figure 4(a)).

Similarly, the execution times related to the detection of heuristic changes depend on the number of detected mappings, as one heuristic change will be detected for each mapping. Our experiments highlight this, as, for instance, for the pairs of versions v25.11.08-v16.12.08 and v24.03.09-v05.05.09 (for GO) the execution time (for heuristic changes) decreases although the size of the input and the size of Δ increase (see Table XII). This happens because the number of detected heuristic changes (i.e., the size of the table containing the mappings) is larger for the first pair (given that 30% of the total changes were heuristic ones, whereas for the second pair such changes constitute 12% of the total – see Figure 4(b)).

9. RELATED WORK

Works related to change detection can be classified under two basic dimensions, namely the level of changes they support (low-level or high-level) and the underlying representation language assumed (e.g., Description Logics [Baader et al. 2002], RDF(S) [McBride et al. 2004; Brickley and Guha 2004] etc.). Low-level change detection algorithms (e.g., [Volkel et al. 2005]) report simple add/delete operations which are not concise or intuitive enough to guarantee human-readability. These works focus on machine readability, and some of them introduce useful formal properties for deltas (e.g., [Zeginis et al. 2011; Franconi et al. 2010]). On the other hand, high-level approaches provide more human-readable deltas, whose changes, like in our paper, are usually distinguished in basic and composite ([Palma et al. 2007; Plessers and De Troyer 2005; Stojanovic et al. 2002]). However, there is no agreed-upon list of basic/composite changes that are necessary for any given context. Various high-level operations and the intuition behind them are presented in [Klein 2004; Noy and Musen 2002; Palma et al. 2007; Plessers et al. 2007; Rogozan and Paquette 2005; Stojanovic 2004]; unfortunately though, high-level change detection approaches do not present formal semantics of such operations ([Klein 2004; Noy and Musen 2002; Palma et al. 2007; Rogozan and Paquette 2005]), or of the corresponding detection process ([Palma et al. 2007]); thus, no useful formal properties can be guaranteed.

In [Franconi et al. 2010], the authors discuss a low-level detection approach for propositional KBs, which can be easily extended to apply to KBs represented under any classical knowledge representation formalism. The authors focus on application semantics, and present a number of desirable formal properties for change detection languages (which are similar to ours), like delta uniqueness, reversibility of changes and the ability to move backwards and forwards in the history of versions using the deltas. A similar set of properties appears also in [Zeginis et al. 2011], where a low-level change detection formalism for RDF(S) KBs is presented. In [Konev et al. 2008], a low-level change detection approach for the Description Logic $\mathcal{EL}$ is described. This work focuses on a concept-based description of the changes, and the returned delta is a set of concepts whose position in the class hierarchy changed. In [Kontchakov et al. 2008], a formal low-level change detection approach for DL-Lite ontologies is presented, which focuses on a semantical description of the changes. As already mentioned, since these works focus on low-level changes, they result in non-concise deltas, which are difficult for a human to understand and do not capture well the perception of the editor.

The authors in [Auer and Herre 2006] present an approach to define high-level changes and formalize their application semantics. Changes are represented as sequences of triples and their application is a simple set-theoretic operation. Their work focuses on defining a formal way to represent changes and does not include the description of a detection process nor of a specific language of changes; in fact, no actual changes have been proposed, only a method for defining them.

Authors in [Klein 2004; Noy and Musen 2002] describe a fixed-point algorithm for detecting changes, which is implemented in PromptDiff, an extension of Protégé [Protege 2002]. The algorithm incorporates heuristic-based matchers in order to detect changes between two versions. Thus, the entire detection process is heuristic-based and introduces an uncertainty in the 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 [Plessers et al. 2007] the Change Definition Language (CDL) is proposed as a means to define 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 must be updated whenever a change occurs; this 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 delta can be produced a posteriori. In [Plessers et al. 2007] terminological changes (such as renamings) are not supported, which reduces the usefulness of the approach.

An earlier version of our work appeared in [Papavasileiou et al. 2009], where a similar high-level language was proposed and formalized. Note however that heuristic changes were not integrated in the framework of [Papavasileiou et al. 2009]. Moreover, in the present paper, we repeated and extended all experiments of [Papavasileiou et al. 2009] using the extended formalization and implementation, and included MO in order to study changes affecting the data level. In addition, we proposed an application algorithm, which was missing from [Papavasileiou et al. 2009]. Finally, proofs for all the presented results as well as the formal definition of the changes in our language are provided in the current version.

10. CONCLUSIONS AND FUTURE WORK

The need for dynamic curated KBs makes the automatic identification of deltas between versions increasingly important for several reasons (storing and communication efficiency, visualization and documentation of deltas, efficient synchronization, study of the KBs' evolution history etc.). Unfortunately, it is often difficult or impossible for curators or editors to accurately record such deltas without automated tools, because manually created deltas are often incomplete, or erroneous [Stojanovic et al. 2002]. Surprisingly, this is true even for centrally curated KBs such as the CIDOC ontology, for which a systematic but manual recording of changes proved to be incomplete (see Section 8).

Deltas can be recorded using either low-level changes or high-level ones. High-level changes are usually preferable because they result in more concise deltas which describe better the perception of the editor. In this paper, we addressed the problem of a posteriori detecting high-level changes. Our approach does not require change logging nor any other kind of user input; the detection of high-level changes uses the low-level delta, which can be easily computed from the given versions. Moreover, our algorithm is not based on heuristics, as the approximation techniques required for the detection of heuristic changes are limited to the matching process and are not an integral part of the detection process.

The main contribution of this work is the proposal of a formal framework that was used for the definition of our language of changes ($\mathcal{L}$), where every change has well-defined detection and application semantics. For our language, we categorized the high-level changes into basic, composite and heuristic. The proposed language consists of 132 changes, of which 54 are basic, 51 are composite and 27 are heuristic changes. Our framework allowed us to define and prove a set of necessary and desirable properties for a language of changes, which guarantee that the produced deltas are both human-interpretable (i.e., a human can read them and understand the perception behind the changes) and machine-interpretable (i.e., a machine can process them in a consistent and deterministic manner). In most existing approaches the definition of the language is based just on intuitive requirements and its semantics are derived by the detection algorithm instead of the other way round, so no formal properties can be guaranteed. In addition, most works focus either on human interpretability (especially those that produce high-level deltas) or machine-interpretability (those that produce low-level deltas), but not both.

The detection semantics of our language was used as the basis for a deterministic detection process. We proved that our language is *complete* and *unambiguous*, in the sense that any difference that might occur between a pair of versions can be described by some unique delta containing changes from $\mathcal{L}$. A novelty of our approach is the incorporation of heuristic changes in a transparent way, so that the same properties that are true for basic and composite changes are also true for heuristic changes. Our detection semantics guarantees that our algorithm will always return, for any given input, exactly one and always the same high-level delta.

Regarding the application semantics, we proved that it is consistent to the detection semantics, so the detected high-level deltas can be used to produce subsequent versions. More specifically, we showed that the bulk application of high-level changes always leads to a correct result. Under certain conditions, the application can also be made sequentially, and the result is the same, *independently* of the application order (composable deltas). Finally, we showed that the effects of a delta can be canceled by applying its (unique) reverse. This ability also allows us to move forwards and backwards in the history of versions, so long as any one version has been kept, as well as the deltas that lead to/from this version.

The detection algorithm that we implemented was proved sound and complete with respect to our language. The complexity of the algorithm is theoretically quadratic over the size of the input, even though it was found to be much smaller in practice. Our evaluation showed that the computational time of the algorithm in practice is dominated by the computation of the low-level delta, so the overhead of generating high-level deltas from low-level ones is small.

Our evaluation showed that the language we propose is not an artificial language but rather intuitive, as its changes appear often in practice. In addition, we studied the effects of the matcher quality on the detection results. Finally, we showed the conciseness of the generated deltas and studied the performance of our algorithm. The evaluation was based on three real-world ontologies, namely CIDOC, GO and MO. These ontologies are different in structure, as the first contains many properties, the second consists mostly of classes, whereas the third is the only one of the three that contains instances. CIDOC was selected mainly because it is a representative ontology for studying change detection, as the editors accompany every version with release notes describing the differences with the previous version, thereby allowing us to measure the correctness and intuitiveness of our detected changes; this allowed us to verify that the changes detected by an automatic detection algorithm are understandable by humans and close to their perception, an issue of major importance, as well as to show that manually recorded deltas are often incomplete. Furthermore, the naming policy

of CIDOC allowed us to develop a matcher with 100% precision and recall, and use it to evaluate the effect of the matcher quality on the detection results. The second ontology (GO) is representative for studying problems related to the change management of ontologies because of its size and update rate, and revealed (among other things) that our algorithm is scalable. The third ontology provided us the means to study the expressiveness of our language regarding changes that happen at the data layer as well as to study the correctness of our algorithm regarding these changes.

In the future, we plan to extend our approach to apply to more expressive representation languages. Moreover, we plan to conduct empirical studies involving real users to evaluate further the intuitiveness of the changes in $\mathcal{L}$ and consider extensions or modifications that would further improve human understandability without jeopardizing machine understandability. More generally, it would also be of interest to formulate meta-conditions (or at least rough guidelines) that could be used by a designer to develop alternative languages that enjoy similar properties as $\mathcal{L}$. In addition, we plan to develop a visualization tool for changes, that will allow more user-friendly change management.

ACKNOWLEDGMENTS

The authors would like to thank Martin Doerr for his assistance with CIDOC-CRM.

References

ARENAS, M., CONSENS, M., AND MALLEA, A. 2010. Revisiting blank nodes in RDF to avoid the semantic mismatch with SPARQL. In *Proceedings of the RDF Next Steps Workshop*.

AUER, S. AND HERRE, H. 2006. A versioning and evolution framework for RDF knowledge bases. In *Perspectives of Systems Informatics, 6^{th} International Andrei Ershov Memorial Conference*.

BAADER, F., CALVANESE, D., MCGUINNESS, D., NARDI, D., AND PATEL-SCHNEIDER, P., Eds. 2002. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press.

BAIROCH, A., APWEILER, R., WU, C., BARKER, W., BOECKMANN, B., FERRO, S., ET AL. 2005. The universal protein resource (UniProt). *Nucleic Acids Research*.

BANERJEE, J., KIM, W., KIM, H., AND KORTH, H. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

BERNERS-LEE, T., HENDLER, J., AND LASSILA, O. 2001. The semantic web. *Scientific American 284*, 34–43.

BIZER, C., HEATH, T., AND BERNERS-LEE, T. 2009. Linked data - the story so far. *International Journal of Semantic Web and Information Systems (IJSWIS) 5,* 3, 1–22.

BRICKLEY, D. AND GUHA, R. 2004. RDF vocabulary description language 1.0: RDF Schema. `www.w3.org/TR/2004/REC-rdf-schema-20040210`.

BUNEMAN, P. 2008. Curated databases. Keynote Talk, Proceedings of the 28^{th} ACM SIGMOD/PODS Conference (PODS-08).

CHAWATHE, S. AND GARCIA-MOLINA, H. 1997. Meaningful change detection in structured data. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

CHAWATHE, S., RAJARAMAN, A., GARCIA-MOLINA, H., AND WIDOM, J. 1996. Change detection in hierarchically structured information. In *Proceedings of the International Conference on Management of Data (SIGMOD)*.

CHRISTOPHIDES, V., KARVOUNARAKIS, G., PLEXOUSAKIS, D., SCHOLL, M., AND TOURTOUNIS, S. 2004. Optimizing taxonomic semantic web queries using labeling schemes. *Web Semantics: Science, Services and Agents on the WWW*.

CIDOC. 2010. *The CIDOC Conceptual Reference Model*. `cidoc.ics.forth.gr/official_release_cidoc.html`.

CLORAN, R. AND IRWIN, B. 2005. Transmitting RDF graph deltas for a cheaper semantic web. In *Proceedings of the 8^{th} Annual Southern African Telecommunication Networks and Applications Conference (SATNAC-05)*.

COBENA, G., ABITEBOUL, S., AND MARIAN, A. 2001. Detecting changes in XML documents. In *Proceedings of the International Conference on Data Engineering (ICDE-01)*.

CURINO, C., MOON, H., AND ZANIOLO, C. 2008. Graceful database schema evolution: the prism workbench. In *Proceedings of the 34^{th} International Conference on Very Large Data Bases*.

EUZENAT, J. AND SHVAIKO, P. 2007. *Ontology Matching*. Springer-Verlag.

FERRARA, A., NIKOLOV, A., AND SCHARFFE, F. 2011. Data linking for the semantic web. *International Journal on Semantic Web and Information Systems (IJSWIS) 7,* 3.

FLOURIS, G., MANAKANATAS, D., KONDYLAKIS, H., PLEXOUSAKIS, D., AND ANTONIOU, G. 2008. Ontology change: Classification and survey. *Knowledge Engineering Review (KER) 23,* 2.

FRANCONI, E., MEYER, T., AND VARZINCZAK, I. 2010. Semantic diff as the basis for knowledge base versioning. In *Proceedings of the 13^{th} International Workshop on Non-Monotonic Reasoning*.

HARTIG, O., BIZER, C., AND FREYTAG, J. C. 2009. Executing SPARQL queries over the web of Linked Data. In *Proceedings of the 8^{th} International Semantic Web Conference (ISWC-09)*.

HILL, D., SMITH, B., MCANDREWS-HILL, M., AND BLAKE, J. 2008. Gene ontology annotations: What they mean and where they come from. *BMC Bioinformatics*.

HORROCKS, I., PATEL-SCHNEIDER, P., AND VAN HARMELEN, F. 2003. From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics 1,* 1, 7–26.

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

KONEV, B., WALTHER, D., AND WOLTER, F. 2008. The logical difference problem for description logic terminologies. In *Proceedings of the 4^{th} International Joint Conference on Automated Reasoning (IJCAR-08)*. 259–274.

KONTCHAKOV, R., WOLTER, F., AND ZAKHARYASCHEV, M. 2008. Can you tell the difference between DL-Lite ontologies? In *Proceedings of the 11^{th} International Conference on Principles of Knowledge Representation and Reasoning (KR-08)*. 285–295.

LAUSEN, G., MEIER, M., AND SCHMIDT, M. 2008. SPARQLing constraints for RDF. In *Proceedings of 11^{th} International Conference on Extending Database Technology (EDBT-08)*. 499–509.

LERNER, B. 2000. A model for compound type changes encountered in schema evolution. *ACM Transactions on Database Systems (TODS) 25,* 1, 83–127.

LEVENSHTEIN, V. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics-Doklady*. Vol. 10.

MARIAN, A., ABITEBOUL, S., COBENA, G., AND MIGNET, L. 2001. Change-centric management of versions in an XML warehouse. In *Proceedings of the International Conference on Very Large Data Bases (VLDB-01)*. 581–590.

MCBRIDE, B., MANOLA, F., AND MILLER, E. 2004. RDF primer. `www.w3.org/TR/rdf-primer`.

MOTIK, B., HORROCKS, I., AND SATTLER, U. 2007. Bridging the gap between OWL and relational databases. In *Proceedings of 17^{th} International World Wide Web Conference*.

NGUYEN, G. AND RIEU, D. 1989. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering 4,* 1, 43–67.

NOY, N., CHUGH, A., LIU, W., AND MUSEN, M. 2006. A framework for ontology evolution in collaborative environments. In *Proceedings of the 5^{th} International Semantic Web Conference*.

NOY, N. AND MUSEN, M. 2002. PromptDiff: A fixed-point algorithm for comparing ontology versions. In *Proceedings of the 18^{th} National Conference on Artificial Intelligence (AAAI-02)*.

PALMA, A., HAASE, P., WANG, Y., AND D'AQUIN, M. 2007. D1.3.1 propagation models and strategies. NeOn Deliverable D1.3.1.

PAPAVASILEIOU, V., FLOURIS, G., FUNDULAKI, I., KOTZINOS, D., AND CHRISTOPHIDES, V. 2009. On detecting high-level changes in RDF(S) KBs. In *Proceedings of the 8^{th} International Semantic Web Conference (ISWC-09)*.

PETERS, R. AND OZSU, M. 1997. An axiomatic model of dynamic schema evolution in objectbase systems. *ACM Transactions on Database Systems (TODS) 22,* 1, 75–114.

PLESSERS, P. AND DE TROYER, O. 2005. Ontology change detection using a version log. In *Proceedings of the 4^{th} International Semantic Web Conference (ISWC-05)*.

PLESSERS, P., DE TROYER, O., AND CASTELEYN, S. 2007. Understanding ontology evolution: A change detection approach. *Web Semantics: Science, Services and Agents on the WWW*.

PROTEGE. 2002. `protege.stanford.edu`.

RAIMOND, Y., GIASSON, F., JACOBSON, K., FAZEKAS, G., AND GANGLER, T. 2010. *Music Ontology Specification*. `musicontology.com/`.

ROGOZAN, D. AND PAQUETTE, G. 2005. Managing ontology changes on the semantic web. In *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence*.

SCHMEDDING, F. 2011. Incremental SPARQL evaluation for query answering on Linked Data. In *Proceedings of the 2^{nd} International Workshop on Consuming Linked Data (COLD-11)*.

SERFIOTIS, G., KOFFINA, I., CHRISTOPHIDES, V., AND TANNEN, V. 2005. Containment and minimization of RDF(S) query patterns. In *Proceedings of the 4^{th} International Semantic Web Conference (ISWC-05)*.

SHADBOLT, N., BERNERS-LEE, T., AND HALL, W. 2006. The semantic web revisited. *IEEE Intelligent Systems 21,* 3, 96–101.

SKARRA, A. AND ZDONIK, S. 1986. The management of changing types in an object-oriented database. In *Proceedings of the Conference on Object-oriented programming systems, languages and applications (OOPLSA-86)*.

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

STOJANOVIC, L., MAEDCHE, A., MOTIK, B., AND STOJANOVIC, N. 2002. User-driven ontology evolution management. In *Proceedings of the 13^{th} International Conference on Knowledge Engineering and Knowledge Management (EKAW-02)*. 285–300.

TAO, J., SIRIN, E., BAO, J., AND MCGUINNESS, D. 2010. Extending OWL with integrity constraints. In *Proceedings of the 23^{rd} International Workshop on Description Logics*.

VOLKEL, M., WINKLER, W., SURE, Y., KRUK, S., AND SYNAK, M. 2005. SemVersion: A versioning system for RDF and ontologies. In *Proceedings of the 2^{nd} European Semantic Web Conference*.

ZEGINIS, D., TZITZIKAS, Y., AND CHRISTOPHIDES, V. 2011. On computing deltas of RDF(S) knowledge bases. *ACM Transactions on the Web (TWEB)*.

ZHDANOVA, A. 2008. Community-driven ontology evolution: Gene ontology case study. In *Proceedings of the 11^{th} International Conference on Business Information Systems (BIS-08)*.

A. PROOFS

This appendix contains the proofs for all theorems appearing in the main text, as well as for some lemmas (which do not appear in the main text).

Theorem 5.1. If $t_1 \rightsquigarrow t_2$ then $t_1 \in \xi^-_M(\mu, c, V_1, V_2)$ iff $t_2 \in \xi^+_M(\mu, c, V_1, V_2)$.

PROOF. ($\Rightarrow$) Take t'_1 such that $t_1 \rightsquigarrow t'_1$. Then, $t'_1 \in \Delta^+$, $t'_1 \notin \delta^+(c)$, so $t_2 \in \Delta^+$, $t_2 \notin \delta^+(c)$. Now take t'_2 such that $t'_2 \rightsquigarrow t_2$. Then, $t'_2 \in \Delta^-$, $t'_2 \notin \delta^-(c)$. Thus, $t_2 \in \xi^+_M(\mu, c, V_1, V_2)$.
($\Leftarrow$) Analogous. □

LEMMA A.1. *Consider two RDF(S) KBs V_1, V_2, and two overlapping changes, c_1, c_2, such that c_1 is a basic change, $\phi(c_1)$ is true, $\phi(c_2)$ is true. Then:*
$\xi^+(c_1, V_1, V_2) \subseteq \xi^+(c_2, V_1, V_2)$ *and* $\xi^-(c_1, V_1, V_2) \subseteq \xi^-(c_2, V_1, V_2)$.

PROOF. For most basic changes c_1, $|\delta^+(c_1) \cup \delta^-(c_1)| = 1$, so the result is obvious. The basic changes for which $|\delta^+(c_1) \cup \delta^-(c_1)| > 1$ are the following: *Add_Type_Class*, *Add_Type_Metaclass*, *Add_Type_Metaproperty*, *Delete_Type_Class*, *Delete_Type_Metaclass*, *Delete_Type_Metaproperty*, and the *Retype_...* operations.
Let us consider c_1 =*Add_Type_Class(a)*. We will show that for all changes c_2 that overlap with c_1, either (a) it cannot be the case that both $\phi(c_1)$, $\phi(c_2)$ are true, or, (b) $\xi^+(c_1, V_1, V_2) \subseteq \xi^+(c_2, V_1, V_2)$ and $\xi^-(c_1, V_1, V_2) \subseteq \xi^-(c_2, V_1, V_2)$.
One such case is some *Retype_...* operations (e.g., *Retype_Metaclass_To_Class(a)*), but the conditions of these operations require that a appears in V_1 (as a metaclass, in our example), whereas the conditions of *Add_Type_Class(a)* require that a does not appear in V_1, so (a) is true.
Another case is *Add_Class(a)* for which (b) holds (see Appendix B).
The same is true for: *Rename_Class(b,a)*, *Merge_Classes(B,a)*, *Split_Class(b,A)* for $a \in A$, *Merge_Classes_Into_Existing(B,a)* and *Split_Classes_Into_Existing(b,A)* for $a \in A$, where (b) is true.
The same arguments can be used when taking c_1 to be one of the following changes: *Add_Type_Metaclass*, *Add_Type_Metaproperty*, *Delete_Type_Class*, *Delete_Type_Metaclass*, *Delete_Type_Metaproperty*.
If we take c_1 to be one of the *Retype_...* operations, then the proof is based on the fact that these operations require a to be of a certain type in both V_1 and V_2. Thus their conditions cannot be true simultaneously with the other overlapping operations, like the basic changes *Add_Type_...*, *Delete_Type_...*, the composite changes *Add_...*, *Delete_...*, the heuristic changes *Rename_...*, *Merge_...*, *Split_...*, as well as the rest of the *Retype_...* operations. □

LEMMA A.2. *Consider two RDF(S) KBs V_1, V_2 and their respective $\Delta = \langle \Delta^+, \Delta^- \rangle$. Consider some basic change $c \in \mathcal{L}$, such that $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$ and $\phi(c)$ is true, and some triple t in Δ. Then, if $t \in \delta^+(c)$ then there is some $c' \in \mathcal{L}$, c': detectable and $t \in \xi^+(c')$. Similarly, if $t \in \delta^-(c)$ then there is some $c' \in \mathcal{L}$, c': detectable and $t \in \xi^-(c')$.*

PROOF. If c is detectable then set $c' = c$ and the proof is complete. If c is not detectable, then, by Definition 5.4, there is some detectable heuristic or composite change (e.g., c') that overlaps with c. Thus, $\phi(c')$ is true, so by Lemma A.1, it follows that $\xi^+(c, V_1, V_2) \subseteq \xi^+(c', V_1, V_2)$ and $\xi^-(c, V_1, V_2) \subseteq \xi^-(c', V_1, V_2)$, so c' is the detectable change we are looking for. □

Theorem 5.5. Any low-level change in the delta between two RDF(S) KBs is consumed by some detectable high-level change.

PROOF. We will consider all the different triple patterns t that can appear in Δ. By Lemma A.2, it suffices to show that for each such pattern there will be at least one

basic change c such that $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and either $t \in \delta^+(c)$ or $t \in \delta^-(c)$:

(1) $t =(x, \mathsf{type}, y) \in \Delta^+$, 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 (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 if $c =$*Add_Type_To_Individual(x,y)* then $t \in \xi^+(c)$ and that $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true (see also Appendix B). 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)*.

(2) $t = (x, \mathsf{type}, y) \in \Delta^-$, 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*.

(3) $t = (x, \mathsf{subClassOf}, y) \in \Delta^+$, 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 if $c =$ *Add_Superclass(x,y)*, or $c =$ *Add_SuperMetaclass(x,y)*, or $c =$ *Add_SuperMetaproperty(x,y)* then $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true, and $t \in \xi^+(c)$.

(4) $t = (x, \mathsf{subClassOf}, y) \in \Delta^-$, 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) $t = (x, \mathsf{subPropertyOf}, y) \in \Delta^+$, 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 by setting $c =$ *Add_Superproperty(x,y)*, $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^+(c)$.

(6) $t = (x, \mathsf{subPropertyOf}, y) \in \Delta^-$, where $x, y \in \mathbf{U}$. Same as above, we can conclude that for $c =$*Delete_Superproperty(x,y)*, $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^-(c)$.

(7) $t = (x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta^+$, where $x \in \mathbf{U}$. By definition, $t \in V_2$, so x is a schema class in V_2; by type inference, $(x, \mathsf{type}, \mathsf{class}) \in V_2$. 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 $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^+(c)$. 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, take c to be one of the following operations:
— $c =$ *Retype_Individual_To_Class(x)*
— $c =$ *Retype_Metaclass_To_Class(x)*
— $c =$ *Retype_Metaproperty_To_Class(x)*
— $c =$ *Retype_Property_To_Class(x)*
Then $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^+(c)$.

(8) $t = (x, \mathsf{subClassOf}, \mathsf{resource}) \in \Delta^-$, 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 (in V_2), take c to be one of the following operations:
— $c =$ *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)*
Then $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^-(c)$.

(9) Using similar arguments as in the above two cases, we can handle all the triples of the form (x, type, y) in Δ^+ or Δ^-, $(x, \mathsf{subClassOf}, y)$ in Δ^+ or Δ^-, where $x \in \mathbf{U}$,

$y \in \{\mathsf{resource}, \mathsf{class}, \mathsf{property}\}$. In particular: $t = (x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta^+$, where $x \in \mathbf{U}$ is the same as in case (7), except that now x is a metaclass, and we should take c to be one of the following:
— $c =$ *Add_Type_Metaclass(x)*
— $c =$ *Retype_Individual_To_Metaclass(x)*
— $c =$ *Retype_Class_To_Metaclass(x)*
— $c =$ *Retype_Metaproperty_To_Metaclass(x)*
— $c =$ *Retype_Property_To_Metaclass(x)*
$t = (x, \mathsf{subClassOf}, \mathsf{class}) \in \Delta^-$, where $x \in \mathbf{U}$ is the same as in case (8), except that now x is a metaclass, and we should take c to be one of the following:
— $c =$ *Delete_Type_Metaclass(x)*
— $c =$ *Retype_Metaclass_To_Individual(x)*
— $c =$ *Retype_Metaclass_To_Class(x)*
— $c =$ *Retype_Metaclass_To_Metaproperty(x)*
— $c =$ *Retype_Metaclass_To_Property(x)*
$t = (x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta^+$, where $x \in \mathbf{U}$ is the same as in case (7), except that now x is a metaproperty, and we should take c to be one of the following:
— $c =$ *Add_Type_Metaproperty(x)*
— $c =$ *Retype_Individual_To_Metaproperty(x)*
— $c =$ *Retype_Class_To_Metaproperty(x)*
— $c =$ *Retype_Metaclass_To_Metaproperty(x)*
— $c =$ *Retype_Property_To_Metaproperty(x)*
$t = (x, \mathsf{subClassOf}, \mathsf{property}) \in \Delta^-$, where $x \in \mathbf{U}$ is the same as in case (8), except that now x is a metaproperty, and we should take c to be one of the following:
— $c =$ *Delete_Type_Metaproperty(x)*
— $c =$ *Retype_Metaproperty_To_Individual(x)*
— $c =$ *Retype_Metaproperty_To_Class(x)*
— $c =$ *Retype_Metaproperty_To_Metaclass(x)*
— $c =$ *Retype_Metaproperty_To_Property(x)*
$t = (x, \mathsf{type}, \mathsf{resource}) \in \Delta^+$, where $x \in \mathbf{U}$ is the same as in case (7), except that now x is an individual, and we should take c to be one of the following:
— $c =$ *Add_Type_Individual(x)*
— $c =$ *Retype_Metaproperty_To_Individual(x)*
— $c =$ *Retype_Class_To_Individual(x)*
— $c =$ *Retype_Metaclass_To_Individual(x)*
— $c =$ *Retype_Property_To_Individual(x)*
$t = (x, \mathsf{type}, \mathsf{resource}) \in \Delta^-$, where $x \in \mathbf{U}$ is the same as in case (8), except that now x is an individual, and we should take c to be one of the following:
— $c =$ *Delete_Type_Individual(x)*
— $c =$ *Retype_Individual_To_Metaproperty(x)*
— $c =$ *Retype_Individual_To_Class(x)*
— $c =$ *Retype_Individual_To_Metaclass(x)*
— $c =$ *Retype_Individual_To_Property(x)*
$t = (x, \mathsf{type}, \mathsf{property}) \in \Delta^+$, where $x \in \mathbf{U}$ is the same as in case (7), except that now x is a property, and we should take c to be one of the following:
— $c =$ *Add_Type_Property(x)*
— $c =$ *Retype_Metaproperty_To_Property(x)*
— $c =$ *Retype_Class_To_Property(x)*
— $c =$ *Retype_Metaclass_To_Property(x)*
— $c =$ *Retype_Individual_To_Property(x)*

$t = (x, \mathsf{type}, \mathsf{property}) \in \Delta^-$, where $x \in \mathbf{U}$ is the same as in case (8), except that now x is a metaproperty, and we should take c to be one of the following:
— $c =$ *Delete_Type_Property(x)*
— $c =$ *Retype_Property_To_Metaproperty(x)*
— $c =$ *Retype_Property_To_Class(x)*
— $c =$ *Retype_Property_To_Metaclass(x)*
— $c =$ *Retype_Property_To_Individual(x)*
$t = (x, \mathsf{type}, \mathsf{class}) \in \Delta^+$, where $x \in \mathbf{U}$ is the same as in case (7), except that now x is either a class, or a metaclass, or metaproperty, and we should take c to be one of the following:
— $c =$ *Add_Type_Class(x)*
— $c =$ *Retype_Individual_To_Class(x)*
— $c =$ *Retype_Metaclass_To_Class(x)*
— $c =$ *Retype_Metaproperty_To_Class(x)*
— $c =$ *Retype_Property_To_Class(x)*
— $c =$ *Add_Type_Metaclass(x)*
— $c =$ *Retype_Individual_To_Metaclass(x)*
— $c =$ *Retype_Class_To_Metaclass(x)*
— $c =$ *Retype_Metaproperty_To_Metaclass(x)*
— $c =$ *Retype_Property_To_Metaclass(x)*
— $c =$ *Add_Type_Metaproperty(x)*
— $c =$ *Retype_Individual_To_Metaproperty(x)*
— $c =$ *Retype_Class_To_Metaproperty(x)*
— $c =$ *Retype_Metaclass_To_Metaproperty(x)*
— $c =$ *Retype_Property_To_Metaproperty(x)*
$t = (x, \mathsf{type}, \mathsf{class}) \in \Delta^-$, where $x \in \mathbf{U}$ is the same as in case (8), except that now x is either a class, or a metaclass, or metaproperty, and we should take c to be one of the following:
— $c =$ *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)*
— $c =$ *Delete_Type_Metaclass(x)*
— $c =$ *Retype_Metaclass_To_Individual(x)*
— $c =$ *Retype_Metaclass_To_Class(x)*
— $c =$ *Retype_Metaclass_To_Metaproperty(x)*
— $c =$ *Retype_Metaclass_To_Property(x)*
— $c =$ *Delete_Type_Metaproperty(x)*
— $c =$ *Retype_Metaproperty_To_Individual(x)*
— $c =$ *Retype_Metaproperty_To_Class(x)*
— $c =$ *Retype_Metaproperty_To_Metaclass(x)*
— $c =$ *Retype_Metaproperty_To_Property(x)*

(10) $t = (x, \mathsf{domain}, y) \in \Delta^+$, 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 $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^-(c)$.

(11) $t = (x, \mathsf{domain}, y) \in \Delta^-$, 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 Δ^+ or Δ^-, where $x \in \mathbf{U}, y \in \mathbf{U} \cup \{\mathsf{resource}, \mathsf{class}, \mathsf{property}, \mathsf{literal}\}$. Same as above, for $c =$*Add_Range(x,y)* or $c =$*Delete_Range(x,y)*.

(13) $t = (x, \mathsf{comment}, y) \in \Delta^+$, 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)*, $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^+(c)$.
(14) $t = (x, \mathsf{comment}, y) \in \Delta^-$, 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 Δ^+ or Δ^-, where $x \in \mathbf{U}, y \in \mathbf{L}$. Same as in the case of comments, for $c =$*Add_Label(x,y)* or $c =$*Delete_Label(x,y)*.
(16) $t = (x, y, z) \in \Delta^+$, 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 $\delta^+(c) \subseteq \Delta^+$, $\delta^-(c) \subseteq \Delta^-$, $\phi(c)$ is true and $t \in \xi^+(c)$.
(17) $t = (x, y, z) \in \Delta^-$, 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)*.

□

Theorem 5.6. The set of detectable changes for any pair of RDF(S) KBs does not contain overlapping changes.

PROOF. Take any two changes c_1, c_2. We will show that if c_1, c_2 are overlapping and c_1 is detectable, then c_2 is not detectable. If c_1, c_2 are of different types (basic, composite, heuristic) then by Definition 5.4, only one of them can be detectable, so c_2 is not detectable. So we will suppose that c_1, c_2 are of the same type.
Suppose first that c_1, c_2 are basic changes. Basic changes have been defined in such a way that, whenever they overlap, their conditions require that the parameters of the basic change are of a certain type at V_1 and/or V_2. This, along with the fact that any resource can be of one type only in any given version, guarantees that such overlapping basic changes cannot be both detectable, because their conditions cannot be true at the same time. For example, *Add_Superclass(a,b)* and *Add_SuperMetaclass(a,b)* overlap, but the former requires a to be a schema class in V_2, whereas the latter requires a to be a metaclass. The same is true for the other pairs of overlapping basic changes, e.g., *Add_Type_To_Class(a,b)*-*Add_Type_To_Metaclass(a,b)*, *Add_Type_Class(a)*-*Retype_Property_To_Class(a)*,
Delete_Type_Class(a)-*Retype_Class_To_Individual(a)* etc.
Now suppose that c_1 and c_2 are composite changes. Let's initially suppose that $c_1 =$*Add_Class(*$a, P_1, P_2, P_3, P_4, P_5, P_6$*)*. Even though c_1 overlaps with several composite changes, its conditions allow us to show the result. For example, the operation $c_2 =$*Add_Property(*$p, P_1, P_2, P_3, P_4, a, p_6, P_7, P_8$*)* overlaps with c_1 over the triple (p, domain, a). However, the requirement of c_2 that a should be of the same type (thus, exist) in both V_1 and V_2 conflicts with the condition of c_1 stating that a should not exist in V_1. Same arguments can be given for the other operations, such as *Pull_up_Class(*a, B, C*)*, *Reclassify_Individual(*p, A, B*)* for $a \in B$ etc.
Using similar arguments we can address all the cases where c_1 is one of the *Add_...* or *Delete_...* operations.
The pairs dealing with hierarchies (i.e., involving one of the operations *Pull_up_...*, *Pull_down_...*, *Move_...*, *Change_...*) are similarly handled, by using the assumption that hierarchies need to be acyclic. Furthermore, the requirements of these operations guarantee that the parameters B, C will contain all the old/new parents, so there can be no conflict of a *Pull_up_...* operation with another *Pull_up_...* operation. Finally, the conditions that $B \neq \emptyset$, $C \neq \emptyset$ guarantee that there can be no conflict with the *Group_...* or *Ungroup_...* operations.
Same arguments can be used for pairs involving *Reclassify_...* operations, as well as for pairs involving *Specialize_Domain*, *Generalize_Domain*, *Change_Domain* opera-

tions (same for range). Finally, the requirements of *Change_To_Datatype_Property* and *Change_To_Object_Property* regarding the typing of the involved resources guarantee that their conditions cannot be true at the same time with another composite operation (e.g., *Change_Range*), despite the overlap.
For heuristic changes we use the requirement to have valid sets of mappings (see Definition 3.1), which guarantees that each URI will be involved in at most one mapping. Given that each mapping corresponds to exactly one operation, there cannot be two overlapping heuristic changes that are both detectable. For example, *Rename_Class*(a_1, a_2), *Rename_Class*(a_1, a_3), are overlapping, but the former would require the mapping $\{a_1\} \rightsquigarrow \{a_2\}$ and the latter the mapping $\{a_1\} \rightsquigarrow \{a_3\}$ for their detection; however, these mappings cannot be both included in a valid set of mappings, so at least one of these operations is not detectable. The same can be said for all pairs of heuristic changes. □

Theorem 6.1. If $t_1 \rightsquigarrow t_2$ then $t_1 \in \alpha^-_M(\mu, C, V)$ iff $t_2 \in \alpha^+_M(\mu, C, V)$.

PROOF. ($\Rightarrow$) Take t'_1 such that $t_1 \rightsquigarrow t'_1$. Then, $t'_1 \notin V$ and there is no $c \in C$ such that $t'_1 \in \delta^+(c)$, so $t_2 \in \Delta^+$ and there is no $c \in C$ such that $t_2 \in \delta^+(c)$. Now take t'_2 such that $t'_2 \rightsquigarrow t_2$. Then, $t'_2 \in V$ and there is no $c \in C$ such that $t'_2 \in \delta^-(c)$. Thus, $t_2 \in \alpha^+_M(\mu, C, V)$.
($\Leftarrow$) Analogous. □

LEMMA A.3. *Consider two RDF(S) KBs V_1, V_2 and a valid set of mappings $\mathcal{M}$. Take also some detectable change c, such that $\mu = S_1 \rightsquigarrow S_2 \in \mathcal{M}(c)$ and some $t = (s, p, o)$. Then:*

—*If* $t \in \xi^+_M(\mu, c, V_1, V_2)$ *then* $s \in S_2$ *or* $p \in S_2$ *or* $o \in S_2$.
—*If* $t \in \xi^-_M(\mu, c, V_1, V_2)$ *then* $s \in S_1$ *or* $p \in S_1$ *or* $o \in S_1$.
—*If* $t \in \alpha^+_M(\mu, C, V_1)$ *then* $s \in S_2$ *or* $p \in S_2$ *or* $o \in S_2$.
—*If* $t \in \alpha^-_M(\mu, C, V_1)$ *then* $s \in S_1$ *or* $p \in S_1$ *or* $o \in S_1$.

PROOF. For the first, suppose that $s \notin S_2, p \notin S_2, o \notin S_2$. Then $t \rightsquigarrow t$. Given that $t \in \xi^+_M(\mu, c, V_1, V_2)$, $t \in \Delta^+$; given that $t \rightsquigarrow t$, it follows that $t \in \Delta^-$. This is a contradiction. We can use similar arguments for the second bullet.
For the third, suppose that $s \notin S_2, p \notin S_2, o \notin S_2$. Then $t \rightsquigarrow t$. Given that $t \in \alpha^+_M(\mu, C, V_1)$, $t \notin V$; given that $t \rightsquigarrow t$, it follows that $t \in V$. This is a contradiction. We can use similar arguments for the fourth bullet. □

LEMMA A.4. *Consider two RDF(S) KBs V_1, V_2, a valid set of mappings $\mathcal{M}$, and the set of changes $C = \{c \in \mathcal{L} \mid c$ is detectable$\}$. Take also some $c \in C$ and $\mu = S_1 \rightsquigarrow S_2 \in \mathcal{M}(c)$ such that $S_1 \cap S_2 \neq \emptyset$. Then: $\xi^+_M(\mu, c, V_1, V_2) = \xi^-_M(\mu, c, V_1, V_2) = \alpha^+_M(\mu, C, V_1) = \alpha^-_M(\mu, C, V_1) = \emptyset$.*

PROOF. Suppose that $S_1 \cap S_2 = \{u\}$. Take some $t = (s, p, o)$ such that $t \in \xi^+_M(\mu, c, V_1, V_2)$, and set t_u the triple that is obtained from t by replacing all URIs of t that appear in S_2 with u. By definition $t_u \rightsquigarrow t$, so $t_u \in \xi^-_M(\mu, c, V_1, V_2)$. Also $t_u \rightsquigarrow t_u$, so by the definition of $\xi^-_M(\mu, c, V_1, V_2)$, if follows that $t_u \in \Delta^+$, $t_u \in \Delta^-$, a contradiction. Thus, $\xi^+_M(\mu, c, V_1, V_2) = \emptyset$. Using similar arguments, we can show that $\xi^-_M(\mu, c, V_1, V_2) = \emptyset$.
Take some $t = (s, p, o)$ such that $t \in \alpha^+_M(\mu, C, V_1)$, and set t_u the triple that is obtained from t by replacing all URIs of t that appear in S_2 with u. By definition $t_u \rightsquigarrow t$, so $t_u \in \alpha^-_M(\mu, C, V_1)$. Also $t_u \rightsquigarrow t_u$, so by the definition of $\alpha^-_M(\mu, C, V_1)$, if follows that $t_u \in V$, $t_u \notin V$, a contradiction. Thus, $\alpha^+_M(\mu, C, V_1) = \emptyset$. Using similar arguments, we can show that $\alpha^-_M(\mu, C, V_1) = \emptyset$. □

LEMMA A.5. *Consider two RDF(S) KBs V_1, V_2, a valid set of mappings $\mathcal{M}$, and the corresponding set of detectable changes C. Take also some $c, c' \in C$ and $\mu \in \mathcal{M}(c)$, $\mu' \in \mathcal{M}(c')$. Then:*

—$t \in \xi^+_M(\mu, c, V_1, V_2) \cap \xi^+_M(\mu', c', V_1, V_2)$ $\xi^+_M(\mu, c, V_1, V_2) \cap \xi^+_M(\mu', c', V_1, V_2) \neq \emptyset$ *implies* $\mu = \mu'$ *and* $c = c'$.
—$t \in \xi^-_M(\mu, c, V_1, V_2) \cap \xi^-_M(\mu', c', V_1, V_2)$ $\xi^-_M(\mu, c, V_1, V_2) \cap \xi^-_M(\mu', c', V_1, V_2) \neq \emptyset$ *implies* $\mu = \mu'$ *and* $c = c'$.
—$t \in \alpha^+_M(\mu, C, V_1) \cap \alpha^+_M(\mu', C, V_1)$ $\alpha^+_M(\mu, C, V_1) \cap \alpha^+_M(\mu', C, V_1) \neq \emptyset$ *implies* $\mu = \mu'$ *and* $c = c'$.
—$t \in \alpha^-_M(\mu, C, V_1) \cap \alpha^-_M(\mu', C, V_1)$ $\alpha^-_M(\mu, C, V_1) \cap \alpha^-_M(\mu', C, V_1) \neq \emptyset$ *implies* $\mu = \mu'$ *and* $c = c'$.

PROOF. For the first bullet, take some $t \in \xi^+_M(\mu, c, V_1, V_2) \cap \xi^+_M(\mu', c', V_1, V_2)$ and suppose that $t = (s, p, o)$, $\mu = S_1 \rightsquigarrow S_2$, $\mu' = S'_1 \rightsquigarrow S'_2$ and $\mu \neq \mu'$. Then, by the definition of valid sets of mappings and Lemma A.4, it follows that all four sets S_1, S_2, S'_1, S'_2 are disjoint. By Lemma A.3, $s \in S'_2$ or $p \in S'_2$ or $o \in S'_2$. Without loss of generality, let us assume that $s \in S'_2$. Then $s \notin S_2$. Take some t_0 such that $t_0 \rightsquigarrow_\mu t$. Given that $t \in \xi^+_M(\mu, c, V_1, V_2)$, it follows that $t_0 \in \Delta^-$, so $t_0 \in V_1$. But, t_0 is obtained by replacing all URIs of t that appear in S_2 with some URI from S_1; since $s \notin S_2$, s will not be replaced, i.e., s will appear in t_0. But $s \in S'_2$, and by the definition of mappings and the fact that $S'_1 \cap S'_2 = \emptyset$, it follows that S'_2 is disjoint from $\mathbf{U}(V_1)$, i.e., $s \notin \mathbf{U}(V_1)$ but $t_0 \in V_1$, a contradiction. Thus, $\mu = \mu'$. By the definition of our language, $\mathcal{L}$ (see Appendix B), it follows that, if $\mu \in \mathcal{M}(c)$, $\mu' \in \mathcal{M}(c')$ and $\mu = \mu'$ then $c = c'$.
The other bullets can be shown using similar arguments. □

Theorem 6.3. Consider two RDF(S) KBs V_1, V_2 and their corresponding set of detectable changes C. Then $V_1 \bullet C = V_2$.

PROOF. Note that $V_1 \bullet C = (V_1 \cup \alpha^+(C, V_1)) \setminus \alpha^-(C, V_1)$ and $V_2 = (V_1 \cup (V_2 \setminus V_1)) \setminus (V_1 \setminus V_2)$ $= (V_1 \cup \Delta^+) \setminus \Delta^-$. Thus, it suffices to show that $\Delta^+ = \alpha^+(C, V_1)$, $\Delta^- = \alpha^-(C, V_1)$.
Take some $t \in \Delta^+$. By Theorem 5.5, $t \in \xi^+(c, V_1, V_2)$ for some $c \in C$. If $t \in \delta^+(c)$ then $t \in \alpha^+(C, V_1)$, so suppose that $t \notin \delta^+(c)$ for all $c \in C$; thus, $t \in \xi^+_M(\mu, c, V_1, V_2)$ for some $c \in C$, $\mu \in \mathcal{M}(c)$. Then, $t' \in \Delta^-$ for all t' such that $t' \rightsquigarrow t$, i.e., $t' \in V_1$ and $t' \notin \delta^-(c)$. Suppose that there is some $c' \in C$, $c \neq c'$ such that $t' \in \delta^-(c')$. Then, $t' \in \xi^-(c', V_1, V_2)$. By Theorem 5.6, we conclude that $t' \notin \xi^-(c, V_1, V_2)$, so by Theorem 5.1 $t \notin \xi^+(c, V_1, V_2)$, a contradiction. Thus, there is no $c' \in C$ such that $t' \in \delta^-(c')$. Similarly, $t'' \in \Delta^+$ for all t'' such that $t' \rightsquigarrow t''$, so $t'' \notin V_1$ and $t'' \notin \delta^+(c)$. If there is some $c' \in C$ such that $t'' \in \delta^+(c')$ we reach a contradiction as before. We conclude that $t \in \alpha^+_M(\mu, C, V_1)$, so $t \in \alpha^+(C, V_1)$ and $\Delta^+ \subseteq \alpha^+(C, V_1)$.
Now take some $t \in \Delta^-$. Using similar arguments, we conclude that $t \in \alpha^-(C, V_1)$, so $\Delta^- \subseteq \alpha^-(C, V_1)$.
Now take some $t \in \alpha^+(C, V_1)$. If $t \in \delta^+(c)$ for some $c \in C$ then $t \in \xi^+(c, V_1, V_2)$, so $t \in \Delta^+$. So, let us suppose that $t \notin \delta^+(c)$ for all $c \in C$. Then, there is some $c \in C$, $\mu \in \mathcal{M}(c)$ such that $t \in \alpha^+_M(\mu, C, V_1)$. Suppose that $\mu = S_1 \rightsquigarrow S_2$. By Lemma A.4, it follows that $S_1 \cap S_2 = \emptyset$. Take any t' such that $t' \rightsquigarrow t$. By the definition of α^+_M, it follows that $t' \in V_1$. Also $t' \in \alpha^-_M(\mu, C, V_1)$ (by Theorem 6.1), so t' contains at least one URI/literal in S_1 (by Lemma A.3). By Definition 3.1, since $S_1 \cap S_2 = \emptyset$, S_1 does not contain URIs/literals that appear in V_2. Thus, $t' \notin V_2$, i.e., $t' \in \Delta^-$. Since $t' \in \Delta^-$, by Theorem 5.5, there is some $c' \in C$ such that $t' \in \xi^-(c', V_1, V_2)$. If $t' \in \delta^-(c')$ then $t \notin \alpha^+_M(\mu, C, V_1)$, a contradiction. Thus, there is some $c' \in C$, $\mu' \in \mathcal{M}(c')$, such that $t' \in \xi^-_M(\mu', c', V_1, V_2)$. Repeating the reasoning of previous parts of this proof, $t' \in \alpha^-_M(\mu', C, V_1)$. By Theorem 6.1, it follows

that $t \in \alpha^+_M(\mu', C, V_1)$, so by Lemma A.5, it follows that $\mu = \mu'$, $c = c'$. We conclude that $t' \in \xi^-_M(\mu, c, V_1, V_2)$, thus, by Theorem 5.1, $t \in \xi^+_M(\mu, c, V_1, V_2)$. The latter implies $t \in \Delta^+$. We conclude that $\alpha^+(C, V_1) \subseteq \Delta^+$.
Finally, take some $t \in \alpha^-(C, V_1)$. Using similar arguments, we conclude that $t \in \Delta^-$, so $\alpha^-(C, V_1) \subseteq \Delta^-$.
The end result is that $\Delta^+ = \alpha^+(C, V_1)$, $\Delta^- = \alpha^-(C, V_1)$ and the proof is complete. □

Theorem 6.4. The set of detectable high-level changes of any pair of RDF(S) KBs is not conflicting.

PROOF. Take C the set of detectable changes and any pair of changes $c_1, c_2 \in C$, $c_1 \neq c_2$. Then:
$\delta^+(c_1) \cap \delta^-(c_2) \subseteq \xi^+(c_1) \cap \xi^-(c_2) \subseteq \Delta^+ \cap \Delta^- = \emptyset$, thus, $\delta^+(c_1) \cap \delta^-(c_2) = \emptyset$. Similarly, we can show that $\delta^-(c_1) \cap \delta^+(c_2) = \emptyset$. Thus, C is non-conflicting. □

Theorem 6.5. Consider an RDF(S) KB V and a set of changes $C = \{c_1, \ldots, c_n\}$, such that C is non-conflicting and independent. Then, $(\ldots((V \bullet \{c_{\pi(1)}\}) \bullet \{c_{\pi(2)}\}) \bullet \ldots) \bullet \{c_{\pi(n)}\} = V \bullet C$ for any permutation π over the set of indices $\{1, \ldots, n\}$.

PROOF. By the definition of α^+_M, α^-_M and the definition of independence, for all $c \in C$ and $\mu \in \mathcal{M}(c)$: $\alpha^+_M(\mu, \{c\}, V_1) = \alpha^+_M(\mu, C, V_1)$ and $\alpha^-_M(\mu, \{c\}, V_1) = \alpha^-_M(\mu, C, V_1)$. Also, $\alpha^+_M(\mu, C, V_1) \cap \alpha^-_M(\mu, C, V_1) = \emptyset$ for all μ. Given that C is non-conflicting, and the above result, it follows that $\alpha^+(C, V) \cap \alpha^-(C, V) = \emptyset$. Combining these results and the related definitions, for any given permutation π, we can show using simple set-theoretic manipulations that $(\ldots((V \bullet \{c_{\pi(1)}\}) \bullet \{c_{\pi(2)}\}) \bullet \ldots) \bullet \{c_{\pi(n)}\} = V \bullet C$. □

Theorem 6.9. Every change in $\mathcal{L}$ has a unique reverse.

PROOF. Taking any change c, it is trivial to show that the corresponding reverse (c^{-1}) listed in Appendix B is indeed the reverse of c. □

Theorem 6.10. If c_2 is the reverse of c_1, then c_1 is the reverse of c_2 and c_1, c_2 are conflicting.

PROOF. Obvious by Definition 6.8. □

Theorem 6.11. For any two RDF(S) KBs V_1, V_2 and change c, the following are true: $\xi^+(c^{-1}, V_2, V_1) = \xi^-(c, V_1, V_2)$ and $\xi^-(c^{-1}, V_2, V_1) = \xi^+(c, V_1, V_2)$.

PROOF. The result is obvious by the definitions of ξ^+, ξ^- and Definition 6.8. □

Theorem 6.12. Consider two RDF(S) KBs V_1, V_2 and their corresponding set of detectable changes C. Set $C^{-1} = \{c^{-1} \mid c \in C\}$. Then C^{-1} is non-conflicting and $V_2 \bullet C^{-1} = V_1$.

PROOF. Take some $c \in C$, $\mu \in \mathcal{M}(c)$ and $t \in \alpha^+_M(\mu, C, V_1)$. Then $t \in \alpha^+(C, V_1)$. Since $t \in \alpha^+_M(\mu, C, V_1)$ it follows that $t \notin V_1$, so $t \notin \alpha^-_M(\mu_0, C, V_1)$ for any μ_0. Also, by the definition of $\alpha^+_M(\mu, C, V_1)$, it follows that there is no $c_0 \in C$ such that $t \in \delta^-(c_0)$. Thus, $t \notin \alpha^-(C, V_1)$. It follows that $t \in V_2$. Similarly, we can show that if $t \in \alpha^-_M(\mu, C, V_1)$ then $t \notin V_2$. Take some $\mu = S_1 \rightsquigarrow S_2 \in \mathcal{M}(c)$ for some $c \in C$. Then, $\mu^{-1} = S_2 \rightsquigarrow S_1 \in \mathcal{M}(c^{-1})$. Thus, $t \rightsquigarrow_\mu t'$ iff $t' \rightsquigarrow_{\mu^{-1}} t$. Furthermore, $\delta^+(c) = \delta^-(c^{-1})$ and $\delta^-(c) = \delta^+(c^{-1})$.
So, now take $c \in C$, $\mu = S_1 \rightsquigarrow S_2 \in \mathcal{M}(c)$ and $t \in \alpha^+_M(\mu, C, V_1)$. Set $\mu^{-1} = S_2 \rightsquigarrow S_1 \in \mathcal{M}(c^{-1})$. Then, using the above results and the definition of α^+_M, α^-_M, we conclude that $t \in \alpha^-_M(\mu^{-1}, C^{-1}, V_2)$. Similarly, if $t \in \alpha^-_M(\mu, C, V_1)$ then $t \in \alpha^+_M(\mu^{-1}, C^{-1}, V_2)$. Finally, $\delta^+(c) = \delta^-(c^{-1})$ and $\delta^-(c) = \delta^+(c^{-1})$. So, $\alpha^+(C, V_1) = \alpha^-(C^{-1}, V_2)$, $\alpha^-(C, V_1) = \alpha^+(C^{-1}, V_2)$. As we showed in Theorem 6.3, $\alpha^+(C, V_1) = \Delta^+(V_1, V_2) = \Delta^-(V_2, V_1)$,

$\alpha^-(C, V_1) = \Delta^-(V_1, V_2) = \Delta^+(V_2, V_1)$, so $\alpha^+(C^{-1}, V_2) = \Delta^+(V_2, V_1)$, $\alpha^-(C^{-1}, V_2) = \Delta^-(V_2, V_1)$. We conclude that $V_2 \bullet C^{-1} = V_1$. □

Theorem 7.1. For any composite change c the following are true: $\cup_{b\in\Sigma(c)}\delta^+(b) = \delta^+(c)$ and $\cup_{b\in\Sigma(c)}\delta^-(b) = \delta^-(c)$.

PROOF. In Appendix B we list all composite changes and their subsumed basic changes. The correctness of the theorem can be established by verifying that the above relations hold for each composite change. □

Theorem 7.2. A change c will be returned by Algorithm 1 with input $V_1, V_2, \mathcal{M}$ iff c is detectable.

PROOF. Set $C = \{c \in \mathcal{L} \mid c \text{ is detectable}\}$, $\Omega = \{c \in \mathcal{L} \mid c$ was returned by Algorithm 1 $\}$.
Consider a change $c \in \Omega$.
If c is a heuristic change then by $findHeuristicChanges$ we know that there is an appropriate mapping in $\mathcal{M}$. In addition, by looking at the definition of the heuristic changes (Appendix B) and the definition of mappings (Definition 3.1), we conclude that, if the appropriate mapping exists, then $\delta^+(c) \subseteq \Delta^+, \delta^-(c) \subseteq \Delta^-$ and the corresponding conditions ($\phi(c)$) will be true. Thus c is detectable ($c \in C$).
If c is a basic change then by $findPotentialChanges$ we know that $\delta^+(c) \subseteq \Delta^+, \delta^-(c) \subseteq \Delta^-$. Moreover, by line 10 of the main algorithm, we also know that $\phi(c)$ is true. Since c is contained in the output of the algorithm we know by line 22 that it is not a subsumed basic change of any composite change nor are its required added and deleted triples consumed by a detectable heuristic change. The latter argument is true because ξ^+, ξ^- are correctly computed in lines 29-30 of Algorithm 2. So $c \in C$.
If c is a composite change then by $findPotentialChanges$ we know that all $b \in \Sigma(c)$ were initially in $basic\_changes$. Moreover, by line 20 of the main algorithm, we also know that $\phi(c)$ is true. Since $c \in \Omega$, its required added and deleted triples are not consumed by any detectable heuristic change. We conclude that c is detectable, i.e., $c \in C$. Thus, $\Omega \subseteq C$.
Now consider a change $c \in C$.
Suppose initially that c is a heuristic change. Since c is detectable, $\mathcal{M}$ will contain an appropriate mapping, so algorithm $findHeuristicChanges$ will detect this change.
Suppose now that c is a basic change. Since c is detectable, $\delta^+ \subseteq \Delta^+$ and $\delta^- \subseteq \Delta^-$, so there will be a low-level change, e.g., t in Δ triggering its detection, unless this low-level change is consumed by some other change. Suppose that the algorithm returns some other change, c', that consumes t (in line 12); then $c' \in \Omega \subseteq C$, so c, c' are detectable and consume the same low-level change (t), a contradiction, by Theorem 5.6. So, the algorithm will eventually read t (in the FOR loop of line 7) and call $findPotentialChanges$ with this triple. Since c is a detectable change, it will be returned by $findPotentialChanges$, its conditions will be true, and it won't be contained in any detectable composite change's set of subsumed changes so it will also be returned by Algorithm 1.
If, on the other hand, c is a composite change, the same argumentation holds, the only difference being that its detection is triggered by some basic change that has been previously detected. By Theorem 7.1, all subsumed basic changes of c were initially in $basic\_changes$. Moreover, as before, Theorem 5.6 guarantees that the basic change triggering the detection of c will not be removed from $basic\_changes$ (in line 22), because there is no other detectable change that contains the same basic changes in its set of subsumed basic changes (otherwise, there would be a conflict in the corresponding low-level changes, a contradiction by Theorem 5.6). Finally, as explained above, c will

be checked for its conditions being true in line 20 and c will be detected.
Therefore, $C \subseteq \Omega$ and the proof is complete. □

Theorem 7.3. The complexity of Algorithm 1 is $O(N^2)$, where N is the total size of the input $(V_1, V_2, \mathcal{M})$.

PROOF. Let us denote by N_i the size of V_i $(i = 1, 2)$.
It will be helpful to sort V_1, V_2 in a pre-processing phase, which requires $O(N_1 \log N_1 + N_2 \log N_2)$. To compute $\Delta(V_1, V_2)$ we can simply scan and compare the sorted V_1, V_2 (any differences are put in $\Delta(V_1, V_2)$), with a total cost of $O(N_1 + N_2)$. Let us assume that the size of Δ is N_Δ. It will be useful to sort Δ as well, taking $O(N_\Delta \log N_\Delta)$ time. Thus, the total time required for the calculation and sorting of Δ is $O(N_1 \log N_1 + N_2 \log N_2 + N_\Delta \log N_\Delta)$.
The next step of the algorithm is the detection of heuristic changes given a set of mappings. Note that the mapping is given as input, so the cost of computing it is not included in our calculations. For simplicity, we set $M = |\mathcal{M}|$.
Given a mapping $\mu = \{A_1, \ldots, A_n\} \rightsquigarrow \{B_1, \ldots, B_m\}$, determining heuristic changes requires identifying the type (class, property, label, etc.) of A_i, B_i. To do so, we perform a pre-processing phase that involves a full scan over the triples in V_1, V_2 to identify the triples which determine the type of each element. This takes $O(N_1 + N_2)$ time. The type of each element can now be stored along with the element, so that any future queries for the type of any element can be made in constant time. Therefore, all the type checking performed in lines 5, 8, 12 etc. of Algorithm 2 can be made in $O(1)$ time for each mapped element, i.e., $O(M)$ in total.
The checks in lines 13, 22 require $O(m), O(n)$ time respectively, i.e., at most the size of the mapping; thus the total cost over all the mappings cannot exceed $O(M)$.
The next step is to compute ξ^+, ξ^- using Algorithm 3. So, consider the i^{th} call to Algorithm 3, for the mapping μ_i; let's assume that μ_i is one-to-many, i.e., of the form $\mu_i = \{A_1\} \rightsquigarrow \{B_1, \ldots, B_m\}$. We loop over all triples t in Δ^- which contain A_1 (line 4 of Algorithm 3), and for which $t \notin \delta^-(c)$. Using an appropriate index, we can identify these triples by searching for the triples whose subject/predicate/object is A_1 (needs $O(\log N_\Delta)$). For the triples found, we must compute T (line 5). In practice, T need not be explicitly computed, but we can take each t' such that $t \rightsquigarrow t'$ and check whether $t' \in \Delta^+$ and $t' \notin \delta^+(c)$. Computing the "next" such t' takes $O(1)$, checking the conditions takes $O(\log N_\Delta)$, and we continue this process until no further t' can be found, or one check fails. Suppose that the total number of checks required for μ_i, over all t identified in line 4, is k_i.
Given that the same reasoning applies if μ_i is many-to-one or one-to-one, we conclude that the total time for Algorithm 3 for call i (under μ_i) is $(k_i+3) \cdot O(\log N_\Delta)$. For all calls to Algorithm 3, assuming that $\mathcal{M}$ has $N_\mathcal{M}$ rows, the total time is $\sum_{i=1}^{N_\mathcal{M}} (k_i+3) \cdot O(\log N_\Delta)$ which is equal to $O(\log N_\Delta) \cdot (3 \cdot N_\mathcal{M} + \sum_{i=1}^{N_\mathcal{M}} k_i)$.
However, note that each triple in Δ can be checked in line 6 at most 3 times (once if its subject is involved in a mapping, once for the predicate and once for the object) over all calls to Algorithm 3. So, the total number of successful searches is at most $3 \cdot N_\Delta$. In addition, we may make some checks for triples that are not in Δ, but for each triple $t \in \Delta^- \setminus \delta^-(c)$ (or $t \in \Delta^+ \setminus \delta^+(c)$) we can make at most one failed check (because after a failed check we stop). Each triple can satisfy the above condition for at most 3 mappings, because each URI appears at most once in the table $\mathcal{M}$; thus, the total number of failed searches is at most equal to $3 \cdot N_\Delta$. We conclude that $\sum_{i=1}^{n} k_i \leq 6 \cdot N_\Delta$. Furthermore, $N_\mathcal{M} \leq M$.
Combining these facts, we conclude that the total time needed to execute line 29 of Algorithm 2, over all $\mu \in \mathcal{M}$ is at most $O((N_\Delta + M) \cdot \log N_\Delta)$. For line 32, the cost is

$O(N_\Delta \cdot \log N_\Delta)$.
Combining all the above results, we conclude that the total time required for Algorithm 2 is $O(N_1 + N_2)$ (for finding the types, in a preprocessing phase), plus $O(M)$ (to find types in lines 5, 8, 12 etc.), plus $O(M)$ (to search for overlapping elements, e.g., line 13), plus $O((M + N_\Delta) \cdot \log N_\Delta)$ (for line 29), plus $O(N_\Delta \log N_\Delta)$ (for line 32). The combined cost is therefore: $O(N_1 + N_2 + (M + N_\Delta) \cdot \log N_\Delta)$.
After detecting heuristic changes, we proceed with the detection of basic and composite ones. For every triple in Δ (N_Δ in total) we call $findPotentialChanges$ to figure out the basic changes whose detection is triggered by it. The potential changes to check in the FOR loop in line 2 of Algorithm 4 can be determined at design time, and they are constant in number. Each such change contains a constant number of changes in $\delta^+ \cup \delta^-$ (set $toCheck$), namely 3 at most (see Appendix B). Thus, each call to Algorithm 4 will spawn a constant number of searches for low-level changes in Δ; the search cost is $O(\log N_\Delta)$. Thus, computing pot takes $O(\log N_\Delta)$ in total.
Since every low-level change can trigger the detection of a finite and predetermined number of basic changes, the size of $potBC$ returned will be $O(1)$. For each potentially detectable basic change in $potBC$, we need to determine whether its conditions are true (line 10 in Algorithm 1). The time required for this depends on the change considered, but for basic changes it always takes $O(1)$ number of checks (see Appendix B). Each individual check can be done in $O(1)$, a result which can be achieved using sophisticated labeling algorithms, as described in [Christophides et al. 2004]. If the check succeeds, we need to add this change to $basic\_changes$ (cost is $O(1)$) and delete the corresponding low-level changes from Δ (cost is $O(\log N_\Delta)$ for each low-level change, of which there are at most 3). Thus, the FOR loop (lines 9-15) will be executed $O(1)$ number of times with a cost of $O(\log N_\Delta)$ for each execution.
We conclude that lines 8-15 will be executed $O(N_\Delta)$ times, each time requiring $O(\log N_\Delta)$ (for line 8), plus $O(\log N_\Delta)$ (for lines 9-15), so the total cost for lines 7-16 of Algorithm 1 is $O(N_\Delta \cdot \log N_\Delta)$. Note also that, the total number of changes in $basic\_changes$ cannot exceed N_Δ; it will also be useful in the following if we sort the changes in $basic\_changes$, an operation that requires $O(N_\Delta \log N_\Delta)$ time.
Similar reasoning can be made for lines 17-26 of Algorithm 1. Again, the method $findPotentialChanges$ will be called $O(N_\Delta)$ number of times, and the FOR loop in line 2 of Algorithm 4 will consider a constant number of changes. The only difference here with respect to the previous analysis of Algorithm 4 is that the number of basic changes in Σ (set $toCheck$) is not constant. However, if the number of basic changes in Σ is more than $|basic\_changes|$ then we don't need to check anything (because the check in lines 5-10 will definitely fail); if the number of basic changes in Σ is less than $|basic\_changes|$ then the check can be made at $O(|basic\_changes| + |\Sigma|)$ time, i.e., $O(N_\Delta)$. We conclude that the total cost of one call to Algorithm 4 for composite changes is $O(N_\Delta)$.
As in the case of basic changes, the size of $potCC$ returned will be $O(1)$. The time required for determining whether the conditions of a change in $potCC$ are true (line 20 in Algorithm 1) depends on the change considered, and for composite changes (e.g., ***Pull_up_Class***), it can be at most quadratic on the size of its parameters (see Appendix B), which is analogous to the size of Σ. For example, in the case of ***Pull_up_Class(a,B,C)***, the cost is $O(p^2)$ where $p = |B \cup C|$; note also that $p = O(|\Sigma|)$. Again, if the check succeeds, we need to add this change to $composite\_changes$ (cost is $O(1)$) and delete the corresponding basic changes from $basic\_changes$ (cost is $O(N_\Delta)$).
Now let us consider the worst-case scenario. The FOR loop in line 17 iterates over the basic changes in $basic\_changes$ ($O(N_\Delta)$ iterations). Let us consider the i^{th} iteration: for the selected change, we need $O(N_\Delta)$ time for line 18, plus $O(1)$ iterations of $O(N_\Delta + p_i^2)$ cost (lines 20-25), where $p_i = O(|\Sigma|)$ for the composite change considered. Then, the

total cost (for the entire loop, in lines 17-26) is: $O(\sum_{i=1}^{N_\Delta}(N_\Delta + N_\Delta + p_i^2))$. However, note that: $\sum_{i=1}^{N_\Delta}(N_\Delta + N_\Delta + p_i^2) = 2 \cdot N_\Delta^2 + \sum_{i=1}^{N_\Delta} p_i^2 \leq 2 \cdot N_\Delta^2 + (\sum_{i=1}^{N_\Delta} p_i)^2$. Given that $p_i = O(|\Sigma|)$ for the composite change considered, the sum $\sum_{i=1}^{N_\Delta} p_i$ cannot exceed the size of $basic\_changes$ so it is $O(N_\Delta)$. Thus, the expression becomes: $2 \cdot N_\Delta^2 + O(N_\Delta)^2$. Therefore, the complexity of lines 17-26 of Algorithm 1 is $O(N_\Delta^2)$.
Now let us combine all the results we have. First, the preprocessing phase (lines 1-4 of Algorithm 1) takes $O(N_1 \log N_1 + N_2 \log N_2 + N_\Delta \log N_\Delta)$. The cost for Algorithm 2 (i.e., the cost for line 5 of Algorithm 1) is $O(N_1 + N_2 + (M + N_\Delta) \cdot \log N_\Delta)$. Lines 7-16 of Algorithm 1 cost $O(N_\Delta \log N_\Delta)$ and are followed by the sorting of $basic\_changes$, which costs $O(N_\Delta \log N_\Delta)$ time. Finally, the cost of lines 17-26 of Algorithm 1 is $O(N_\Delta^2)$. Therefore, the total cost of Algorithm 1 is $O(N_1 \log N_1 + N_2 \log N_2 + N_\Delta^2 + M \cdot \log N_\Delta)$. Given that $N_1 = O(N), N_2 = O(N), M = O(N)$ and $N_\Delta \leq N_1 + N_2 \leq N$ the above result is simplified to $O(N^2)$. □

Theorem 7.4. The output of Algorithm 5 with input C, V is $V \bullet C$.

PROOF. The correctness of the algorithm is trivially established using the related definitions, once we note that T^- (computed in line 7 of Algorithm 5) will contain all the triples t'' for which there is a triple t' such that $t \rightsquigarrow t'$, $t'' \rightsquigarrow t'$. □

Theorem 7.5. The complexity of Algorithm 5 for input C, V is $O(N_V \cdot N_C^2 \cdot (\log N_C + \log N_V))$, where N_C, N_V are the sizes of C, V respectively.

PROOF. As a preprocessing phase, we sort the triples in V, which requires $O(N_V \cdot \log N_V)$ time.
The sum of the number of triples in $\delta^+(c)$ over all $c \in C$ are $O(N_C)$ (same for $\delta^-(c)$); thus, line 1 of Algorithm 5 requires $O(N_C)$ time and the sets $apply^+, apply^-$ can have at most $O(N_C)$ size. For efficiency purposes, we also sort $apply^+, apply^-$, which requires $O(N_C \cdot \log N_C)$ time.
Let us now consider a given mapping μ_i, with size n_i. Then, set $T_i = |T^+| + |T^-|$, where T^+, T^- the sets computed in lines 6-7. T_i cannot be larger than $n_i^2 + 1$; the latter will happen when a triple of the form (u, P, u) exists in V and μ_i is of the form: $u \rightsquigarrow \{u_1, \ldots, u_{n_i}\}$ (note that a triple of the form (u, u, u) cannot appear in V because of the assumption of validity). Line 8 requires at most T_i searches in $apply^+, apply^-, V$, costing a total of $O(T_i \cdot (\log N_C + \log N_V))$. Line 9 requires time equal to $O(T_i)$. This process (i.e., lines 6-10) may have to be repeated, in the worst case, $O(N_V)$ times (if most of the triples in V are of the above form). Thus, the total required time for lines 5-11, for μ_i, is $O(N_V \cdot T_i \cdot (\log N_C + \log N_V))$.
Thus, if there are k mappings in total in the changes in C, then the total time required for lines 2-13 is: $O(\sum_{i=1}^{k}(N_V \cdot T_i \cdot (\log N_C + \log N_V)))$, which can be written as: $O(N_V \cdot (\log N_C + \log N_V) \cdot \sum_{i=1}^{k} T_i)$. Given that $T_i \leq n_i^2 + 1$, we deduce that $T_i = O(n_i^2)$. Furthermore, $\sum_{i=1}^{k}(n_i^2) \leq (\sum_{i=1}^{k} n_i)^2 \leq N_C^2$, so $\sum_{i=1}^{k} T_i = O(N_C^2)$. Thus, the total time required for lines 2-13 is: $O(N_V \cdot N_C^2 \cdot (\log N_C + \log N_V))$.
The maximum size of $applyM^+$ is equal to $\sum_{i=1}^{k} T_i$, which is at most $O(N_C^2)$. Given that $applyM^-$ must be a subset of V (line 8), the maximum size of $applyM^-$ is N_V. Thus, line 14 requires $O(N_C)$ searches in V (to remove $apply^-$), then $O(N_V)$ searches (to remove $applyM^-$), then $O(N_C)$ time (to add $apply^+$), then $O(N_C^2)$ time (to add $applyM^+$). This gives a total of $O(N_C \cdot \log N_V + N_V \cdot \log N_V + N_C + N_C^2)$.
Combining all the above results, the complexity of Algorithm 5 is: $O(N_V \cdot N_C^2 \cdot (\log N_C + \log N_V))$. □

B. CHANGES IN THE LANGUAGE

In this appendix, we list the changes defined in our language, $\mathcal{L}$. In particular, the tables below list, for each change, its parameters, the intuition it captures, its formal semantics ($\delta^+, \delta^-, \mathcal{M}, \phi$) and its reverse change. For composite changes, the corresponding subsumed basic changes (Σ) are also listed. Note that the mapping ($\mathcal{M}$) is omitted for basic and composite changes as it is by definition equal to $\emptyset$.

For simplicity, the conditions in the following tables use clauses like "a is a metaclass in V_1", rather than the more verbose (and formal) statement: $(a, \mathsf{type}, \mathsf{class}) \in V_1 \wedge (a, \mathsf{subClassOf}, \mathsf{class}) \in V_1$.

Note that in the conditions of composite changes, when a URI is asked to exist in both versions, mappings are also taken into account so that renamed classes are considered the "same". For example, consider the case of a class A that is moved into a "higher" position in the hierarchy; suppose that its old direct superclass was B and its new one is C, a superclass of B. In that case, the proper operation to detect is *Pull_up_Class*, whose conditions require that C is a superclass of B in both V_1 and V_2. Suppose now that during the same change session, C is renamed into C'. In that case, the conditions of *Pull_up_Class* would be false, thus the operation would not be detected. To address this case, we need to consider the mappings, and require that B is a subclass of C in V_1 and a subclass of C' in V_2.

This provision guarantees that a composite change will not be "missed" just because one of the URIs involved was renamed. Note that such a condition would not make sense for merged or split classes (or URIs in general). To avoid cluttering the tables with these complex conditions, the composite changes below use the simple version of the condition. Note also that this more complicated check is only necessary for composite changes; for basic changes, the conditions are related to one of the versions only, so there would be no problem with renamed classes.

B.1. Basic changes

Change	***Add_Type_Class***(a)	***Delete_Type_Class***(a)
Intuition	Add object a of type class	Delete object a of type class
Parameters	a = The added object	a = The deleted object
δ^+	(a, type, class), (a, subClassOf, resource)	$\emptyset$
δ^-	$\emptyset$	(a, type, class), (a, subClassOf, resource)
ϕ	a does not appear in V_1	a does not appear in V_2
c^{-1}	*Delete_Type_Class(a)*	*Add_Type_Class(a)*

The changes *Add_Type_Metaclass*, *Delete_Type_Metaclass*, *Add_Type_Metaproperty* and *Delete_Type_Metaproperty* are defined analogously with the exception that $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ respectively should be in δ^+ (δ^-) instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$.

Change	***Add_Type_Property***(a)	***Delete_Type_Property***(a)
Intuition	Add object a of type property	Delete object a of type property
Parameters	a = The added object	a = The deleted object
δ^+	$(a,$ type, property$)$	$\emptyset$
δ^-	$\emptyset$	$(a,$ type, property$)$
ϕ	a does not appear in V_1	a does not appear in V_2
c^{-1}	*Delete_Type_Property(a)*	*Add_Type_Property(a)*

The changes *Add_Type_Individual* and *Delete_Type_Individual* are defined analogously with the exception that $(a, \text{type}, \text{resource})$ should be in δ^+ (δ^-) instead of $(a, \text{type}, \text{property})$.

Change	***Retype_Class_To_Metaclass***(a)	***Retype_Metaclass_To_Class***(a)
Intuition	Retype class a to a metaclass	Retype metaclass a to a class
Parameters	a = The retyped object	a = The retyped object
δ^+	$(a,$ subClassOf, class$)$	$(a,$ subClassOf, resource$)$
δ^-	$(a,$ subClassOf, resource$)$	$(a,$ subClassOf, class$)$
ϕ	a is a schema class in $V_1 \wedge$ a is a metaclass in V_2	a is a metaclass in $V_1 \wedge$ a is a schema class in V_2
c^{-1}	*Retype_Metaclass_To_Class(a)*	*Retype_Class_To_Metaclass(a)*

The rest of the retyping operations, namely:

— *Retype_Class_To_Metaproperty*(a)
— *Retype_Class_To_Individual*(a)
— *Retype_Class_To_Property*(a)
— *Retype_Metaclass_To_Metaproperty*(a)
— *Retype_Metaclass_To_Individual*(a)
— *Retype_Metaclass_To_Property*(a)
— *Retype_Metaproperty_To_Class*(a)
— *Retype_Metaproperty_To_Metaclass*(a)
— *Retype_Metaproperty_To_Individual*(a)
— *Retype_Metaproperty_To_Property*(a)
— *Retype_Individual_To_Class*(a)
— *Retype_Individual_To_Metaclass*(a)
— *Retype_Individual_To_Individual*(a)
— *Retype_Individual_To_Property*(a)
— *Retype_Property_To_Class*(a)
— *Retype_Property_To_Metaclass*(a)
— *Retype_Property_To_Metaproperty*(a)
— *Retype_Property_To_Individual*(a)

are defined analogously (details omitted).

Change	***Add_Superclass***(a,b)	***Delete_Superclass***(a,b)
Intuition	Parent b of class a is added	Parent b of class a is deleted
Parameters	a = The class b = The new parent	a = The class b = The old parent
δ^+	$(a,$ subClassOf, $b)$	$\emptyset$
δ^-	$\emptyset$	$(a,$ subClassOf, $b)$
ϕ	a is a schema class in $V_2 \wedge$ $b \neq$ resource	a is a schema class in $V_1 \wedge$ $b \neq$ resource
c^{-1}	*Delete_Superclass(a,b)*	*Add_Superclass(a,b)*

The changes *Add_SuperMetaclass*, *Delete_SuperMetaclass*, *Add_SuperMetaproperty* and *Delete_SuperMetaproperty* are defined analogously with the exception that $(a, \text{subClassOf}, \text{class})$ and $(a, \text{subClassOf}, \text{property})$ should be in δ^+ (δ^-) instead of $(a, \text{subClassOf}, \text{resource})$, and that the conditions should be adapted analogously to require that a is of the proper type.

Change	***Add_Superproperty***(a,b)	***Delete_Superproperty***(a,b)
Intuition	Parent b of property a is added	Parent b of property a is deleted
Parameters	a = The property b = The new parent	a = The property b = The old parent
δ^+	$(a, \mathsf{subPropertyOf}, b)$	$\emptyset$
δ^-	$\emptyset$	$(a, \mathsf{subPropertyOf}, b)$
ϕ	–	–
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
Parameters	a = The class b = The new type (metaclass)	a = The class b = The old type (metaclass)
δ^+	(a, type, b)	$\emptyset$
δ^-	$\emptyset$	(a, type, b)
ϕ	a is a schema class in $V_2 \wedge$ $b \neq$ class	a is a schema class in $V_1 \wedge$ $b \neq$ class
c^{-1}	*Delete_Type_From_Class(a,b)*	*Add_Type_To_Class(a,b)*

The changes *Add_Type_To_Property*, *Delete_Type_From_Property*, *Add_Type_To_Individual* and *Delete_Type_From_Individual* are defined analogously with the exception that the conditions should be adapted analogously to require that a is of the proper type.

Change	***Add_Property_Instance***(a_1,a_2,b)	***Delete_Property_Instance***(a_1,a_2,b)
Intuition	Add property instance of property b	Delete property instance of property b
Parameters	a_1 = The subject a_2 = The object b = The property	a_1 = The subject a_2 = The object b = The property
δ^+	(a_1, b, a_2)	$\emptyset$
δ^-	$\emptyset$	(a_1, b, a_2)
ϕ	$b \notin$ {subClassOf, subPropertyOf, type, comment, label, domain, range}	$b \notin$ {subClassOf, subPropertyOf, type, comment, label, domain, 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
Parameters	a = The property b = The domain	a = The property b = The domain
δ^+	(a, domain, b)	$\emptyset$
δ^-	$\emptyset$	(a, domain, b)
ϕ	–	–
c^{-1}	*Delete_Domain(a,b)*	*Add_Domain(a,b)*

The changes *Add_Range* and *Delete_Range* are defined analogously with the exception that (a, range, b) should be in δ^+ (δ^-) instead of (a, domain, 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
Parameters	a = The object b = The new comment	a = The object b = The old comment
δ^+	$(a, \mathsf{comment}, b)$	$\emptyset$
δ^-	$\emptyset$	$(a, \mathsf{comment}, b)$
ϕ	–	–
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
Parameters	a = The object b = The new comment	a = The object b = The old comment
δ^+	(a, label, b)	$\emptyset$
δ^-	$\emptyset$	(a, label, b)
ϕ	–	–
c^{-1}	*Delete_Label(a,b)*	*Add_Label(a,b)*

B.2. 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
Parameters	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^+	$\forall p \in P_1 : (a, \mathsf{subClassOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subClassOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p \in P_4 : (p, \mathsf{type}, a)$, $\forall p \in P_5 : (a, \mathsf{comment}, p)$, $\forall p \in P_6 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$\emptyset$
δ^-	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{subClassOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subClassOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall p \in P_4 : (p, \mathsf{type}, a)$, $\forall p \in P_5 : (a, \mathsf{comment}, p)$, $\forall p \in P_6 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
ϕ	a does not appear in $V_1 \wedge$ $\forall p \in P_1 \cup P_2$: p is a schema class in $V_1 \wedge$ $\forall p \in P_3$: p is a metaclass in $V_1 \wedge$ $\forall p \in P_4$: p is an individual in V_1	a does not appear in $V_2 \wedge$ $\forall p \in P_1 \cup P_2$: p is a schema class in $V_2 \wedge$ $\forall p \in P_3$: p is a metaclass in $V_2 \wedge$ $\forall p \in P_4$: p is an individual in V_2
Σ	*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)

The changes *Add_Metaclass*, *Add_Metaproperty*, *Delete_Metaclass* and *Delete_Metaproperty* are defined analogously by adapting the required added/deleted triples (δ^+, δ^-) and conditions (ϕ) accordingly.

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
Parameters	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 pairs 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 metaproperties of a, P_4 = set of pairs 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^+	$\forall p \in P_1 : (a, \mathsf{subPropertyOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subPropertyOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall (p_1, p_2) \in P_4 : (p_1, a, p_2)$, $(a, \mathsf{domain}, p_5)$, (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^-	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{subPropertyOf}, p)$, $\forall p \in P_2 : (p, \mathsf{subPropertyOf}, a)$, $\forall p \in P_3 : (a, \mathsf{type}, p)$, $\forall (p_1, p_2) \in P_4 : (p_1, a, p_2)$, $\forall p \in P_7 : (a, \mathsf{comment}, p)$, $\forall p \in P_8 : (a, \mathsf{label}, p)$, $(a, \mathsf{domain}, p_5)$, (a, range, p_6), $(a, \mathsf{type}, \mathsf{property})$
ϕ	a does not exist in $V_1 \wedge$ $\forall p \in P_1 \cup P_2$ p is a property in $V_1 \wedge$ $\forall p \in P_3$ p is a metaproperty in $V_1 \wedge$ $\forall (p_1, p_2) \in P_4$ p_1 is of the same type in both V_1, V_2, and p_2 is of the same type in both $V_1, V_2 \wedge$ p_5 is of the same type in both $V_1, V_2 \wedge$ p_6 is of the same type in both V_1, V_2	a does not exist in $V_2 \wedge$ $\forall p \in P_1 \cup P_2$ p is a property in $V_2 \wedge$ $\forall p \in P_3$ p is a metaproperty in $V_2 \wedge$ $\forall (p_1, p_2) \in P_4$ p_1 is of the same type in both V_1, V_2, and p_2 is of the same type in both $V_1, V_2 \wedge$ p_5 is of the same type in both $V_1, V_2 \wedge$ p_6 is of the same type in both V_1, V_2
Σ	*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	***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
Parameters	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^+	$\forall p \in P_1 : (a, \mathsf{type}, p)$, $\forall p \in P_2 : (a, \mathsf{comment}, p)$, $\forall p \in P_3 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{resource})$	$\emptyset$
δ^-	$\emptyset$	$\forall p \in P_1 : (a, \mathsf{type}, p)$, $\forall p \in P_2 : (a, \mathsf{comment}, p)$, $\forall p \in P_3 : (a, \mathsf{label}, p)$, $(a, \mathsf{type}, \mathsf{resource})$
ϕ	a does not appear in $V_1 \wedge$ $\forall p \in P_1$ p is a schema class in V_1	a does not appear in $V_2 \wedge$ $\forall p \in P_1$ p is a schema class in V_2
Σ	*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	***Pull_up_Class(***a***,*** B_1***,*** B_2***)***	***Pull_down_Class(a,*** B_1***,*** B_2***)***
Intuition	Move class a to a higher position in the subsumption hierarchy	Move a class to a lower position in the subsumption hierarchy
Parameters	B_1 = set of old parents of a, B_2 = set of new parents of a	B_1 = set of old parents of a, B_2 = set of new parents of a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$
ϕ	a is a schema class in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : ((b_1, \mathsf{subClassOf}, b_2) \in Cl(V_1) \wedge (b_1, \mathsf{subClassOf}, b_2) \in Cl(V_2)) \wedge$ $\forall b \notin B_1 \cup B_2 : ((a, \mathsf{subClassOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subClassOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$	a is a schema class in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : ((b_2, \mathsf{subClassOf}, b_1) \in Cl(V_1) \wedge (b_2, \mathsf{subClassOf}, b_1) \in Cl(V_2)) \wedge$ $\forall b \notin B \cup C : ((a, \mathsf{subClassOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subClassOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: *Add_Superclass(a,*b_2*)*, $\forall b_1 \in B_1$: *Delete_Superclass(a,*b_1*)*	$\forall b_2 \in B_2$: *Add_Superclass(a,*b_2*)*, $\forall b_1 \in B_1$: *Delete_Superclass(a,*b_1*)*
c^{-1}	Pull_down_Class(a,B_2,B_1)	Pull_up_Class(a,B_2,B_1)

The changes *Pull_up_Metaclass*, *Pull_up_Metaproperty*, *Pull_down_Metaclass* and *Pull_down_Metaproperty* are defined analogously by adapting the conditions to require that a, $b_1 \in B_1$ and $b_2 \in B_2$ are of the proper type.

Change	***Move_Class(a,*** B_1***,*** B_2***)***	***Change_Superclasses(a,*** B_1***,*** B_2***)***
Intuition	Move a class to a different subsumption hierarchy	Change the parents of class a
Parameters	B_1 = set of old parents of a, B_2 = set of new parents of a	B_1 = set of old parents of a, B_2 = set of new parents of a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$
ϕ	a is a schema class in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \mathsf{subClassOf}, b_1) \notin Cl(V_1) \wedge$ $(b_2, \mathsf{subClassOf}, b_1) \notin Cl(V_2) \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_1, \mathsf{subClassOf}, b_2) \notin Cl(V_1) \wedge$ $(b_1, \mathsf{subClassOf}, b_2) \notin Cl(V_2) \wedge$ $\forall b \notin B_1 \cup B_2 : ((a, \mathsf{subClassOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subClassOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$	a is a schema class in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $(\neg\phi$***(Pull_up_Class(a,*** B_1, B_2***))*** $\wedge$ $\neg\phi$***(Pull_down_Class(a,*** B_1, B_2***))*** $\wedge$ $\neg\phi$***(Move_Class(a,*** B_1, B_2***)))*** $\wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: ***Add_Superclass(a,*** b_2***)***, $\forall b_1 \in B_1$: ***Delete_Superclass(a,*** b_1***)***	$\forall b_2 \in B_2$: ***Add_Superclass(a,*** b_2***)***, $\forall b_1 \in B_1$: ***Delete_Superclass(a,*** b_1***)***
c^{-1}	Move_Class(a,B_2,B_1)	Change_Superclasses(a,B_2,B_1)

The changes *Move_Metaclass*, *Move_Metaproperty*, *Change_SuperMetaclass* and *Change_SuperMetaproperties* are defined analogously by adapting the conditions to require that a, $b_1 \in B_1$ and $b_2 \in B_2$ are of the proper type.

Change	***Group_Classes(*** A***,*** b***)***	***Ungroup_Classes(*** A***,*** b***)***
Intuition	Group classes in A under b	Ungroup classes in A
Parameters	A = set of classes that have as new parent b, b = new parent class	A = set of classes that had as parent b, b = old parent class
δ^+	$\forall a \in A : (a, \mathsf{subClassOf}, b)$	$\emptyset$
δ^-	$\emptyset$	$\forall a \in A : (a, \mathsf{subClassOf}, b)$
ϕ	$\forall a \in A : a$ is a schema class in both V_1 and $V_2 \wedge$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subClassOf}, x) \in V_1 \rightarrow (a, \mathsf{subClassOf}, x) \in V_2$	$\forall a \in A : a$ is a schema class in both V_1 and $V_2 \wedge$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subClassOf}, x) \in V_2 \rightarrow (a, \mathsf{subClassOf}, x) \in V_1$
Σ	$\forall a \in A$: ***Add_Superclass(a,b)***	$\forall a \in A$: ***Delete_Superclass(a,b)***
c^{-1}	Ungroup_Classes(A,b)	Group_Classes(A,b)

The changes *Group_Metalasses*, *Group_Metaproperties*, *Ungroup_Metalasses* and *Ungroup_Metaproperties* are defined analogously, by adapting the conditions to require that $a \in A$ and b are of the proper type.

Change	***Pull_up_Property***(a, B_1, B_2)	***Pull_down_Property***(a, B_1, B_2)
Intuition	Move property a to a higher position in the subsumption hierarchy	Move property a to a lower position in the subsumption hierarchy
Parameters	B_1 = set of old parents of a, B_2 = set of new parents of a	B_1 = set of old parents of a, B_2 = set of new parents of a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{subPropertyOf}, b_2)$	$\forall b_2 \in B_2 : (a, \mathsf{subPropertyOf}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{subPropertyOf}, b_1)$	$\forall b_1 \in B_1 : (a, \mathsf{subPropertyOf}, b_1)$
ϕ	a is a property in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a property in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2$ ($(b_1, \mathsf{subPropertyOf}, b_2) \in Cl(V_1) \wedge$ $(b_1, \mathsf{subPropertyOf}, b_2) \in Cl(V_2) (\wedge$ $\forall b \notin B_1 \cup B_2$ $((a, \mathsf{subPropertyOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subPropertyOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$	a is a property in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a property in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \mathsf{subPropertyOf}, b_1) \in Cl(V_1) \wedge$ $(b_2, \mathsf{subPropertyOf}, b_1) \in Cl(V_2) \wedge$ $\forall b \notin B_1 \cup B_2$ $((a, \mathsf{subPropertyOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subPropertyOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: *Add_Superproperty(a,b_2)*, $\forall b_1 \in B_1$: *Delete_Superproperty(a,b_1)*	$\forall b_2 \in B_2$: *Add_Superproperty(a,b_2)*, $\forall b_1 \in B_1$: *Delete_Superproperty(a,b_1)*
c^{-1}	*Pull_down_Property(a,B_2,B_1)*	*Pull_up_Property(a,B_2,B_1)*

Change	***Move_Property***(a, B_1, B_2)	***Change_Superproperties***(a, B_1, B_2)
Intuition	Move property a to a different subsumption hierarchy	Change the parents of property a
Parameters	B_1 = set of old parents of a, B_2 = set of new parents of a	B_1 = set of old parents of a, B_2 = set of new parents of a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$	$\forall b_2 \in B_2 : (a, \mathsf{subClassOf}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$	$\forall b_1 \in B_1 : (a, \mathsf{subClassOf}, b_1)$
ϕ	a is a property in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a property in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 ($ $(b_2, \mathsf{subPropertyOf}, b_1) \notin Cl(V_1) \wedge$ $(b_2, \mathsf{subPropertyOf}, b_1) \notin Cl(V_2) \wedge$ $(b_1, \mathsf{subPropertyOf}, b_2) \notin Cl(V_1) \wedge$ $(b_1, \mathsf{subPropertyOf}, b_2) \notin Cl(V_2)) \wedge$ $\forall b \notin B_1 \cup B_2$ $((a, \mathsf{subPropertyOf}, b) \in V_1 \leftrightarrow (a, \mathsf{subPropertyOf}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$	a is a property in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a property in both V_1 and $V_2 \wedge$ $(\neg\phi$(*Pull_up_Property(a,B_1,B_2)*$) \wedge$ $\neg\phi$(*Pull_down_Property(a,B_1,B_2)*$) \wedge$ $\neg\phi$(*Move_Property(a,B_1,B_2)*$)) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: *Add_Superproperty(a,b_2)*, $\forall b_1 \in B_1$: *Delete_Superproperty(a,b_1)*	$\forall b_2 \in B_2$: *Add_Superproperty(a,b_2)*, $\forall b_1 \in B_1$: *Delete_Superproperty(a,b_1)*
c^{-1}	Move_Property(a,B_2,B_1)	Change_Superproperties(a,B_2,B_1)

Change	***Group_Properties_Under***(A, b)	***Ungroup_Properties_Under***(A, b)
Intuition	Group properties in A under b	Ungroup properties in A under b
Parameters	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^+	$\forall a \in A : (a, \mathsf{subPropertyOf}, b)$	$\emptyset$
δ^-	$\emptyset$	$\forall a \in A : (a, \mathsf{subPropertyOf}, b)$
ϕ	$\forall a \in A : a$ is a property in both V_1 and $V_2 \wedge$ b is a property in both V_1 and $V_2 \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subPropertyOf}, x) \in Cl(V_1) \rightarrow$ $(a, \mathsf{subPropertyOf}, x) \in Cl(V_2)) \wedge$ $\forall a \in A : (a, \mathsf{subPropertyOf}, b) \notin Cl(V_1)$	$\forall a \in A : a$ is a property in both V_1 and $V_2 \wedge$ b is a property in both V_1 and $V_2 \wedge$ $\forall a \in A, \forall x : (a, \mathsf{subPropertyOf}, x) \in Cl(V_1) \rightarrow$ $(a, \mathsf{subPropertyOf}, x) \in Cl(V_2)) \wedge$ $\forall a \in A : (a, \mathsf{subPropertyOf}, b) \notin Cl(V_2)$
Σ	$\forall a \in A$: *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	***Reclassify_Individual_Higher*** ***(a,*** B_1***,*** B_2***)***	***Reclassify_Individual_Lower*** ***(a,*** B_1***,*** B_2***)***
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
Parameters	B_1 = set of old classes instantiating a, B_2 = set of new classes instantiating a	B_1 = set of old classes instantiating a, B_2 = set of new classes instantiating a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{type}, b_2)$	$\forall b_2 \in B_2 : (a, \mathsf{type}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{type}, b_1)$	$\forall b_1 \in B_1 : (a, \mathsf{type}, b_1)$
ϕ	a is an individual in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 ($ $(b_1, \mathsf{subClassOf}, b_2) \in Cl(V_1) \wedge$ $(b_1, \mathsf{subClassOf}, b_2) \in Cl(V_2)) \wedge$ $\forall b \notin B_1 \cup B_2 ($ $(a, \mathsf{type}, b) \in V_1 \leftrightarrow (a, \mathsf{type}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$	a is an individual in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \mathsf{subClassOf}, b_1) \in Cl(V_1) \wedge$ $(b_2, \mathsf{subClassOf}, b_1) \in Cl(V_2) \wedge$ $\forall b \notin B_1 \cup B_2 ($ $(a, \mathsf{type}, b) \in V_1 \leftrightarrow (a, \mathsf{type}, b) \in V_2) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: *Add_Type_To_Individual(a,*b_2*)*, $\forall b_1 \in B_1$: *Delete_Type_From_Individual(a,*b_1*)*	$\forall b_2 \in B_2$: *Add_Type_To_Individual(a,*b_2*)*, $\forall b_1 \in B_1$: *Delete_Type_From_Individual(a,*b_1*)*
c^{-1}	*Reclassify_Individual_Lower(a,*B_2*,*B_1*)*	*Reclassify_Individual_Higher(a,*B_2*,*B_1*)*

The changes *Reclassify_Class_Higher*, *Reclassify_Class_Lower*, *Reclassify_Property_Higher* and *Reclassify_Property_Lower* are defined analogously with the exception that the typing requirements in the conditions should be adapted analogously.

Change	***Reclassify_Individual(a,*** B_1***,*** B_2***)***
Intuition	Reclassify an individual
Parameters	B_1 = set of old classes instantiating a, B_2 = set of new classes instantiating a
δ^+	$\forall b_2 \in B_2 : (a, \mathsf{type}, b_2)$
δ^-	$\forall b_1 \in B_1 : (a, \mathsf{type}, b_1)$
ϕ	a is an individual in both V_1 and $V_2 \wedge$ $\forall b \in B_1 \cup B_2$ b is a schema class in both V_1 and $V_2 \wedge$ $(\neg\phi$(*Reclassify_Individual_Higher(a,*B_1, B_2*)*$) \wedge$ $\neg\phi$(*Reclassify_Individual_Lower(a,*B_1, B_2*)*$)) \wedge$ $B_1 \neq \emptyset \wedge$ $B_2 \neq \emptyset$
Σ	$\forall b_2 \in B_2$: *Add_Type_To_Individual(a,*b_2*)*, $\forall b_1 \in B_1$: *Delete_Type_From_Individual(a,*b_1*)*
c^{-1}	Reclassify_Individual(a,B_2,B_1)

The changes *Reclassify_Class* and *Reclassify_Property* are defined analogously with the exception that the typing requirements in the conditions should be adapted analogously.

Change	***Specialize_Domain***(a,b_1,b_2)	***Generalize_Domain***(a,b_1,b_2)
Intuition	Change the domain of property a to a subclass of it	Change the domain of property a to a superclass of it
Parameters	b_1 = old domain of a, b_2 = new domain of a	b_1 = old domain of a, b_2 = new domain of a
δ^+	$(a,$ domain, $b_2)$	$(a,$ domain, $b_2)$
δ^-	$(a,$ domain, $b_1)$	$(a,$ domain, $b_1)$
ϕ	a is a property in both V_1 and $V_2\wedge$ b_1 is of the same type in both V_1 and $V_2\wedge$ b_2 is of the same type in both V_1 and $V_2\wedge$ $(b_2,$ subClassOf, $b_1) \in Cl(V_1)\wedge$ $(b_2,$ subClassOf, $b_1) \in Cl(V_2)$	a is a property in both V_1 and $V_2\wedge$ b_1 is of the same type in both V_1 and $V_2\wedge$ b_2 is of the same type in both V_1 and $V_2\wedge$ $(b_1,$ subClassOf, $b_2) \in Cl(V_1)\wedge$ $(b_1,$ subClassOf, $b_2) \in Cl(V_2)$
Σ	*Add_Domain(a,b_2)*, *Delete_Domain(a,b_1)*	*Add_Domain(a,b_2)*, *Delete_Domain(a,b_1)*
c^{-1}	Specialize_Domain(a,b_2,b_1)	Generalize_Domain(a,b_2,b_1)

The changes *Specialize_Range* and *Generalize_Range* are defined analogously by replacing domain with range in all positions.

Change	***Change_Domain***(a,b_1,b_2)
Intuition	Change the domain of property a.
Parameters	b_1 = old domain of a, b_2 = new domain of a
δ^+	$(a,$ domain, $b_2)$
δ^-	$(a,$ domain, $b_1)$
ϕ	a is a property in both V_1 and $V_2\wedge$ b_1 is of the same type in both V_1 and $V_2\wedge$ b_2 is of the same type in both V_1 and $V_2\wedge$ $\neg\phi$(*Specialize_Domain(a,b_1,b_2)*)$\wedge$ $\neg\phi$(*Generalize_Domain(a,b_1,b_2)*)
Σ	*Add_Domain(a,b_2)*, *Delete_Domain(a,b_1)*
c^{-1}	*Change_Domain(a,b_2,b_1)*

The change *Change_Range* is defined analogously by replacing domain with range in all positions.

Change	***Change_To_Datatype_Property***(a,b_1,b_2)	***Change_To_Object_Property***(a,b_1,b_2)
Intuition	Change the range of property a to a datatype	Change the range of property a to an object
Parameters	b_1 = old range of a, b_2 = new range of a	b_1 = old range of a, b_2 = new range of a
δ^+	$(a,$ range, $b_2)$	$(a,$ range, $b_2)$
δ^-	$(a,$ range, $b_1)$	$(a,$ range, $b_1)$
ϕ	a is a property in both V_1 and $V_2\wedge$ b_1 is a schema class, or a metaclass, or a metaproperty in $V_1\wedge$ b_2 is a literal type in V_2	a is a property in both V_1 and $V_2\wedge$ b_1 is a literal type in $V_1\wedge$ b_2 is a schema class, or a metaclass, or a metaproperty in V_2
Σ	*Add_Range(a,b_2)*, *Delete_Range(a,b_1)*	*Add_Range(a,b_2)*, *Delete_Range(a,b_1)*
c^{-1}	*Change_To_Object_Property(a,b_2,b_1)*	*Change_To_Datatype_Property(a,b_2,b_1)*

B.3. Heuristic Changes

Change	***Rename_Class***(a,b)
Intuition	Rename class a to b
Parameters	a = the old name of the class, b = the new name of the class
δ^+	$(b, \mathsf{type}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$
δ^-	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
$\mathcal{M}$	$\{a\} \rightsquigarrow \{b\}$
ϕ	a does not appear in $V_2 \wedge$ b does not appear in V_1
c^{-1}	Rename_Class(b,a)

The changes *Rename_Metaclass* and *Rename_Metaproperty* are defined analogously with the exception that $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ should be in δ^+ (δ^-) instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$.

Change	***Rename_Property***(a,b)
Intuition	Rename property a to b
Parameters	a = the old name of the property, b = the new name of the property
δ^+	$(b, \mathsf{type}, \mathsf{property})$
δ^-	$(a, \mathsf{type}, \mathsf{property})$
$\mathcal{M}$	$\{a\} \rightsquigarrow \{b\}$
ϕ	a does not appear in $V_2 \wedge$ b does not appear in V_1
c^{-1}	*Rename_Property(b,a)*

The change *Rename_Individual* is defined analogously, by replacing property with resource in δ^+, δ^-.

Change	***Merge_Classes***(A,b)	***Split_Class***(a,B)
Intuition	Merge classes contained in A into b	Split class a into classes contained in B
Parameters	A = the set of old names of the classes, b = the new name of the class	a = the old name of the class, B = the set of new names of the classes
δ^+	$(b, \mathsf{type}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$	$\forall b \in B : (b, \mathsf{type}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$
δ^-	$\forall a \in A : (a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$(a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$
$\mathcal{M}$	$A \rightsquigarrow \{b\}$, where $b \notin A$, $\lvert A\rvert > 1$	$\{a\} \rightsquigarrow B$, where $a \notin B$, $\lvert B\rvert > 1$
ϕ	$\forall a \in A$ a does not appear in $V_2 \wedge$ b does not appear in V_1	a does not appear in $V_2 \wedge$ $\forall b \in B$ b does not appear in V_1
c^{-1}	*Split_Class(b,A)*	*Merge_Classes(A,b)*

The changes *Merge_Metaclasses*, *Merge_Metaproperties*, *Split_Metaclass* and *Split_Metaproperty* are defined analogously with the exception that $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ should be in δ^+ (δ^-) instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$.

Change	***Merge_Classes_Into_Existing (A,b)***	***Split_Class_Into_Existing (a,B)***
Intuition	Merge classes contained in A into b	Split class a into classes contained in B
Parameters	A = the set of old names of the classes, b = the new name of the class	a = the old name of the class, B = the set of new names of the classes
δ^+	$\emptyset$	$\forall b \in B \setminus \{a\} : (b, \mathsf{type}, \mathsf{class})$, $(b, \mathsf{subClassOf}, \mathsf{resource})$
δ^-	$\forall a \in A \setminus \{b\} : (a, \mathsf{type}, \mathsf{class})$, $(a, \mathsf{subClassOf}, \mathsf{resource})$	$\emptyset$
$\mathcal{M}$	$A \rightsquigarrow \{b\}$, where $b \in A$	$\{a\} \rightsquigarrow B$, where $a \in B$
ϕ	$\forall a \in A \setminus \{b\}$ a does not appear in $V_2 \wedge$ b is a schema class in both V_1 and V_2	a is a schema class in both V_1 and $V_2 \wedge$ $\forall b \in B \setminus \{a\}$ b does not appear in V_1
c^{-1}	*Split_Class_Into_Existing(b,A)*	*Merge_Classes_Into_Existing(A,b)*

The changes *Merge_Metaclasses_Into_Existing*, *Merge_Metaproperties_Into_Existing*, *Split_Metaclass_Into_Existing* and *Split_Metaproperty_Into_Existing* are defined analogously with the exception that $(a, \mathsf{subClassOf}, \mathsf{class})$ and $(a, \mathsf{subClassOf}, \mathsf{property})$ should be in δ^+ (δ^-) instead of $(a, \mathsf{subClassOf}, \mathsf{resource})$, and that the conditions should be adapted to the correct type.

Change	***Merge_Properties(A,b)***	***Split_Property(a,B)***
Intuition	Merge properties contained in A into b	Split property a into properties contained in B
Parameters	A = the set of old names of the properties, b = the new name of the property	a = the old name of the property, B = the set of new names of the properties
δ^+	$(b, \mathsf{type}, \mathsf{property})$	$\forall b \in B : (b, \mathsf{type}, \mathsf{property})$
δ^-	$\forall a \in A : (a, \mathsf{type}, \mathsf{property})$	$(a, \mathsf{type}, \mathsf{property})$
$\mathcal{M}$	$A \rightsquigarrow \{b\}$	$\{a\} \rightsquigarrow B$
ϕ	$\forall a \in A$ a does not appear in $V_2 \wedge$ b does not appear in V_1	a does not appear in $V_2 \wedge$ $\forall b \in B$ b does not appear in V_1
c^{-1}	*Split_Property(b,A)*	*Merge_Properties(A,b)*

The changes *Merge_Individuals* and *Split_Individual* are defined analogously, by replacing property with resource in δ^+, δ^-.

Change	***Merge_Properties_Into_Existing (A,b)***	***Split_Property_Into_Existing (a,B)***
Intuition	Merge properties contained in A into b	Split property a into properties contained in B
Parameters	A = the set of old names of the properties, b = the new name of the property	a = the old name of the property, B = the set of new names of the properties
δ^+	$\emptyset$	$\forall b \in B \setminus a : (b, \mathsf{type}, \mathsf{property})$
δ^-	$\forall a \in A \setminus b : (a, \mathsf{type}, \mathsf{property})$	$\emptyset$
$\mathcal{M}$	$A \rightsquigarrow \{b\}$, where $b \in A$	$\{a\} \rightsquigarrow B$, where $a \in B$
ϕ	$\forall a \in A \setminus \{b\}$ a does not appear in $V_2 \wedge$ b is a property in both V_1 and V_2	a is a property in both V_1 and $V_2 \wedge$ $\forall b \in B \setminus \{a\}$ b does not appear in V_1
c^{-1}	*Split_Property_Into_Existing(b,A)*	*Merge_Properties_Into_Existing(A,b)*

The changes *Merge_Individuals_Into_Existing* and *Split_Individual_Into_Existing* are defined analogously, by replacing property with resource in δ^+, δ^- and adapting the conditions to refer to the proper type.

Change	***Change_Comment***(u,a,b)	***Change_Label***(u,a,b)
Intuition	Change comment of resource u from a to b	Change label of resource u from a to b
Parameters	a = the old comment, b = the new comment	a = the old label, B = the new label
δ^+	$(u, \mathsf{comment}, b)$	(u, label, b)
δ^-	$(u, \mathsf{comment}, a)$	(u, label, a)
$\mathcal{M}$	$\{a\} \rightsquigarrow \{b\}$	$\{a\} \rightsquigarrow \{b\}$
ϕ	–	–
c^{-1}	*Change_Comment(u,b,a)*	*Change_Label(u,b,a)*