On Detecting High-Level Changes in RDF/S KBs

V. Papavassiliou[1,2], G. Flouris[1], I. Fundulaki[1], D. Kotzinos[1,3], and V. Christophides[1,2]

[1] FORTH-ICS, Greece
[2] University of Crete, Greece
[3] TEI of Serres, Greece
{papavas,fgeo,fundul,kotzino,christop}@ics.forth.gr

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

1 Introduction

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

Unless they are assisted by collaborative ontology development tools [8,9], ontology editors are rarely able or willing to systematically record the changes performed to obtain an ontology version. In particular, when there is no central authority responsible for ontology curation, manually created deltas are often absent, incomplete, or even erroneous [22]. Existing ontology diff tools, such as PromptDiff [14], SemVersion [23] and others [24] aim to satisfy this need. These tools are essentially based on a *language of changes*, which describes the semantics of the different change operations that the underlying algorithm understands and detects.

[1] www.geneontology.org, cidoc.ics.forth.gr.

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

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

The main contributions of our work are:

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

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

2 Motivating Example

In Figure 1 an example inspired from the CIDOC Conceptual Reference Model [4] is depicted; CIDOC is a core ontology intended to facilitate the integration, mediation and interchange of heterogeneous cultural heritage information. Table 1 shows the added and deleted triples (the low-level delta) as well as the high-level change operations that our approach will detect in this example. The table makes clear that even though the low-level delta contains all the changes that have been performed, it is not really useful as it captures the syntactical manipulations that led to the change, rather than the intentions of the editor. Our work is motivated by the belief that the "aggregation"

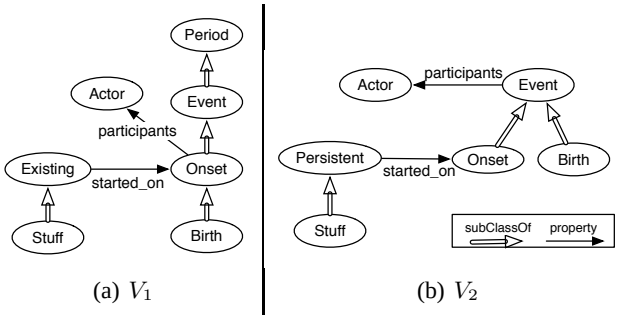


Fig. 1. Motivating Example

of several low-level changes into more *coarse-grained*, *concise* and *intuitive* high-level changes (third column of Table 1) would lead to more useful deltas.

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

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

Another interesting example is the *Rename_Class* operation, which is reported instead of the deletion of class $Existing$ and the subsequent addition of $Persistent$. Unlike the changes discussed so far, the detection of *Rename* (as well as other operations, such as *Merge* and *Split*) requires the use of a matcher that would identify the two concepts ($Existing$ and $Persistent$) to be the same entity using heuristics.

Note also that all the triples of the low-level delta (in Table 1) are associated with one, and only one, high-level change. This allows the partition of low-level changes into well-defined high-level changes, in a unique way. This property guarantees that the detection algorithm will be able to handle all possible low-level deltas in a *deterministic* manner, i.e., that any set of low-level changes between two versions would be associated

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

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

with *one, and only one*, set of high-level changes. The latter requirement calls for a careful definition of the change operations and is not directly related to the detection algorithm *per se*. Thus, we claim that the detection algorithm should be based on the defined language of changes, instead of the other way around.

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

3 Change Detection Framework, Language and Algorithm

3.1 Formal Definitions

The representation of knowledge in RDF [12] is based on triples of the form (*subject*, *predicate*, *object*). Assuming two disjoint and infinite sets $\mathbf{U}$, $\mathbf{L}$, denoting the URIs and literals respectively, $\mathcal{T} = \mathbf{U} \times \mathbf{U} \times (\mathbf{U} \cup \mathbf{L})$ is the set of all triples. An *RDF Graph* V is defined as a set of triples, i.e., $V \subseteq \mathcal{T}$. RDFS [2] introduces some built-in classes (class, property) which are used to determine the *type* of each resource. Following the approach of [20], we assume that each resource is associated with one type determined by the triples that the resource participates in. The typing mechanism allows us to concentrate on nodes of RDF Graphs, rather than triples, which is closer to ontology curators' perception and useful for defining intuitive high-level changes. RDFS [2] provides also *inference semantics*, which is of two types, namely *structural inference* (provided mainly by the transitivity of subsumption relations) and *type inference* (provided by the typing system, e.g., if p is a property, the triple $(p, \mathsf{type}, \mathsf{property})$ can be inferred). The RDF Graph containing all triples that are either explicit or can be inferred from explicit triples in an RDF Graph V (using *both* types of inference), is called the *closure* of V and is denoted by $Cl(V)$. An *RDF/S Knowledge Base* (*RDF/S KB*) V is an RDF Graph which is closed with respect to *type inference*, i.e., it contains all the triples that can be inferred from V using type inference.

For a pair of RDF/S KBs ($V_1, V_2 \subseteq \mathcal{T}$), we define their *low-level delta* in a manner similar to symmetric difference:

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

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

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

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

For simplicity, we will denote by $\delta_1(c)$ ($\delta_2(c)$) the required added (deleted) triples of a change c, and by $\phi(c)$ the required conditions of c. Tables 2, 3 show the definition of some high-level changes. The complete list of defined changes can be found at [16]. We restrict our attention to changes for which $\delta_1 \cup \delta_2 \neq \emptyset$ and $\delta_1 \cap \delta_2 = \emptyset$. The first condition guarantees that at least *something* must be in Δ_1 or Δ_2 for a change to be detected. The second condition guarantees that no change would require the addition and deletion of the same triple to happen at the same time.

As discussed in Section 2, both fine-grained and coarse-grained high-level changes are necessary in order to support determinism, conciseness and intuitiveness. For this reason, we follow a common approach in the literature [10,17,21] and classify high-level changes into *basic* and *composite*. Basic changes are fine-grained and describe a change in one node or edge of the RDF/S KB taking into account RDF/S semantics. On the other hand, composite changes are coarse-grained and closer to the user's intuition, as they describe, in a concise way, changes affecting several nodes and/or edges of the RDF/S KB. The introduction of the two levels should be done carefully, as it may cause problems with determinism. For instance, in the motivating example (Figure 1 and Table 1), the deleted triple $(participants, \mathsf{domain}, Onset)$ could be associated with the basic change *Delete_Domain(participants,Onset)*, as well as with the composite change *Generalize_Domain(participants,Onset,Event)* (see also Tables 2, 3). This double association would jeopardize determinism, because the same low-level delta would correspond to two different high-level deltas. To avoid this problem, we define two different

Change	Delete_Superclass(x,y)	Add_Property_Instance(x_1, x_2,y)	Delete_Domain(x,y)
Intuition	IsA between x, y is deleted	Add property instance (x_1, y, x_2)	Domain y of property x is deleted
$\boldsymbol{\delta_1}$	$\emptyset$	$\{(x_1, y, x_2)\}$	$\emptyset$
$\boldsymbol{\delta_2}$	$\{(x, \mathsf{subClassOf}, y)\}$	$\emptyset$	$\{(x, \mathsf{domain}, y)\}$
$\boldsymbol{\phi}$	$(x, \mathsf{type}, \mathsf{class}) \in Cl(V_1)$	$(y, \mathsf{type}, \mathsf{property}) \in Cl(V_2)$	$(x, \mathsf{type}, \mathsf{property}) \in Cl(V_1)$

Table 2. Formal Definition of some Basic Changes

Change	Generalize_Domain(x,y,z)	Change_Domain(x,y,z)	Reclassify_Individual_Higher(x,Y,Z)
Intuition	Domain of property x changes from y to a superclass z	Domain of property x changes from y to a non-subclass/superclass z	Individual x is reclassified from class(es) Y to superclass(es) Z
$\boldsymbol{\delta_1}$	$\{(x, \mathsf{domain}, z)\}$	$\{(x, \mathsf{domain}, z)\}$	$\{(x, \mathsf{type}, z) \mid z \in Z\}$
$\boldsymbol{\delta_2}$	$\{(x, \mathsf{domain}, y)\}$	$\{(x, \mathsf{domain}, y)\}$	$\{(x, \mathsf{type}, y) \mid y \in Y\}$
$\boldsymbol{\phi}$	$(x, \mathsf{type}, \mathsf{property}) \in Cl(V_1) \wedge$ $(x, \mathsf{type}, \mathsf{property}) \in Cl(V_2) \wedge$ $(y, \mathsf{subClassOf}, z) \in Cl(V_1) \wedge$ $(y, \mathsf{subClassOf}, z) \in Cl(V_2)$	$(x, \mathsf{type}, \mathsf{property}) \in Cl(V_1) \wedge$ $(x, \mathsf{type}, \mathsf{property}) \in Cl(V_2) \wedge$ $((y, \mathsf{subClassOf}, z) \notin Cl(V_1) \vee$ $(y, \mathsf{subClassOf}, z) \notin Cl(V_2)) \wedge$ $((z, \mathsf{subClassOf}, y) \notin Cl(V_1) \vee$ $(z, \mathsf{subClassOf}, y) \notin Cl(V_2))$	$(x, \mathsf{type}, \mathsf{resource}) \in Cl(V_1) \wedge$ $(x, \mathsf{type}, \mathsf{resource}) \in Cl(V_2) \wedge$ $\forall y \in Y, \forall z \in Z :$ $(y, \mathsf{subClassOf}, z) \in Cl(V_1) \wedge$ $(y, \mathsf{subClassOf}, z) \in Cl(V_2)$

Table 3. Formal Definition of some Composite Changes

notions, *detectability* and *initial detectability*, and postulate that the detection of composite changes takes precedence over the detection of basic ones.

Definition 3. *Consider two RDF/S KBs V_1, V_2, their respective $\Delta(V_1, V_2)$ and a change c. Then, c is* initially detectable *iff $\delta_i(c) \subseteq \Delta_i$, $i \in \{1, 2\}$, and $\phi(c)$ is true.*
If c is a composite change, then c is detectable *iff it is initially detectable.*
If c is a basic change, then c is detectable *iff it is initially detectable and there is no initially detectable composite change (say c') for which $\delta_i(c) \subseteq \delta_i(c')$, $i \in \{1, 2\}$ and $\phi(c') \vdash \phi(c)$.*

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

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

Definition 4. *Consider an RDF/S KB V and a change c. The* application *of c upon V, denoted by $V \bullet c$ is defined as: $V \bullet c = (V \cup \delta_1(c)) \setminus \delta_2(c)$.*

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

3.2 Formal Results on the Proposed Language of Changes

Our framework was used to define $\mathcal{L}$, a specific language of changes (some of which are shown in Tables 2, 3) that satisfies several interesting properties. For a full definition of $\mathcal{L}$ and the proofs of the described properties, see [16].

First of all, $\mathcal{L}$ should conform to the property of *Completeness* by capturing any possible change, so that the detection algorithm can always process the input and return a delta. Moreover, the language should satisfy the property of *Non-ambiguity* by associating each low-level change with one, and only one, high-level change, and each set of low-level changes with one, and only one, set of high-level ones. These two properties are needed in order to guarantee that $\mathcal{L}$ supports a *deterministic detection process*.

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

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

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

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

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

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

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

Theorems 1 and 3 guarantee a deterministic detection process. To see this, take any V_1, V_2, i.e., any Δ, and any triple $t \in \Delta$: by Theorem 1, t is associated with at least one

detectable high-level change; moreover, by theorem 3, all detectable high-level changes have disjoint sets of required added (and deleted) triples; thus, t is associated with exactly one detectable high-level change. This means that any set of low-level changes can be fully partitioned into disjoint subsets, each subset being associated with a single detectable high-level change.

In the rest of this subsection, we will consider the application of changes and show that the detection and application semantics are such that, given V_1, V_2, the application of the set of detectable changes between them upon V_1 would give V_2. Before showing that, we must generalize Definition 4 to apply for sets of changes; given that elements in a set are unordered, before doing this generalization, we must first guarantee that the order of application does not matter.

Definition 5. *Two changes c_1, c_2 are called* conflicting *iff $(\delta_1(c_1) \cap \delta_2(c_2)) \cup (\delta_1(c_2) \cap \delta_2(c_1)) \neq \emptyset$. A set C of changes is called* conflicting *iff C contains at least one pair of conflicting changes.*

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

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

Definition 6. *Consider an RDF/S KB V and a non-conflicting set of changes $C = \{c_1, \ldots, c_n\}$. The* application *of C upon V, denoted by $V \bullet C$, is defined as: $V \bullet C = (\ldots((V \bullet c_1) \bullet c_2) \bullet \ldots) \bullet c_n$.*

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

Definition 6 is the generalization of Definition 4 for non-conflicting sets. Given that we cannot define the application of sets of changes for conflicting sets, the result that C is non-conflicting is a critical part of Theorem 5. Theorem 5 shows that we can apply the detected delta upon one version in order to get the other.

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

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

Another useful property of $\mathcal{L}$ is *Reversibility*, i.e., for each change c, there is some change whose application cancels the effects of c. Thus, by keeping only the newest version of an RDF/S KB and the changes that led to it, previous versions can be restored.

Low-Level Change Considered	Low-Level Change(s) Searched For	Potential High-Level Change
$(x, \mathsf{domain}, z) \in \Delta_2$	–	*Delete_Domain(x,z)*
$(x, \mathsf{domain}, z) \in \Delta_2$	$(x, \mathsf{domain}, y) \in \Delta_1$	*Change_Domain(x,y,z)*
$(x, \mathsf{domain}, z) \in \Delta_2$	$(x, \mathsf{domain}, y) \in \Delta_1$	*Generalize_Domain(x,y,z)*

Table 4. Look-up Table (Excerpt)

Definition 7. *A change c_1 is called the* reverse *of c_2 iff $\delta_1(c_1)$=$\delta_2(c_2)$ and $\delta_2(c_1)$=$\delta_1(c_2)$.*

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

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

Theorem 8 shows that the reverse of a change always exists and is unique; we will denote by c^{-1} the reverse of $c \in \mathcal{L}$. For example, the reverse of *Change_Domain (participants,Onset,Event)* is *Change_Domain(participants,Event,Onset)*.

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

Theorem 9 shows how a set of changes can be canceled by applying its reverse upon the result. This allows for both "undoing" an unwanted change, and reproducing older versions of an RDF/S KB.

3.3 Change Detection Algorithm

An essential part of our approach is the detection algorithm for $\mathcal{L}$ (Algorithm 1), which should be efficient, scalable and should correctly return the detectable changes. The first step of the algorithm is to pick a low-level change (i.e., a triple in Δ_1 or Δ_2), say $(participants, \mathsf{domain}, Onset) \in \Delta_2$ (cf. Figure 1 and Table 1). Regardless of the particular input (V_1, V_2), there are certain high-level changes whose detection cannot be triggered by a given low-level change. For example, the deletion of triple ($participants$, domain, $Onset$) cannot be related to the detection of *Delete_Superclass*, as no low-level change of this form appears in the required deleted triples of *Delete_Superclass* (see Table 2). On the other hand, it can potentially trigger the detection of a *Delete_Domain* or a *Change_Domain* operation if the latter is coupled with some other low-level change in Δ specifying the addition of a new domain for $participants$ (Table 3).

This kind of reasoning allows us to build a look-up table (excerpt shown in Table 4), which is used by $findPotentialChanges(t, \Delta)$ (line 3) to return the set of high-level changes $potC$, whose detection could, potentially, be triggered by the selected low-level change (t). $findPotentialChanges$ works as follows: if the selected t is in the left column of Table 4, then we check whether the low-level changes in the middle column appear in Δ; if so, then t *could* trigger the detection of the high-level change in the right column, so this high-level change is put in $potC$. In our example, $potC$ will contain

Algorithm 1 Change Detection Algorithm

```
1: changes = ∅
2: for all low-level changes t do
3:     potC := findPotentialChanges(t,Δ)
4:     for all c ∈ potC do
5:         if φ(c) = true then
6:             changes := changes ∪ {c}
7:             Δ1 := (Δ1 \ δ1(c)), Δ2 := (Δ2 \ δ2(c))
8:             break
9:         end if
10:    end for
11: end for
12: return changes
```

Delete_Domain(participants,Onset), Change_Domain(participants,Onset,Event) and *Generalize_Domain(participants,Onset, Event)*.

Method $findPotentialChanges$ performs a first "filtering", guaranteeing that the only (per Theorem 3) detectable high-level change associated with the low-level change under question is one of the changes in $potC$. To find it, we check the required conditions of each member of $potC$, and, once we find one whose conditions are true, we add it to the list of detectable changes (line 6). In our running example, *Generalize_Domain(participants,Onset,Event)* will be detected. Note that in order for the correct detection to take place, composite changes should be considered first in the for loop of line 4. Therefore, even though the conditions of the basic change *Delete_Domain(participants,Onset)* are also true, the algorithm will never reach that point (due to the "break" command in line 8). This is according to our definition that a basic change is detectable only if there is no composite change that is also detectable and more general. Note also that the elimination of the associated low-level changes from Δ (line 7) would not cause problems thanks to *Non-ambiguity* (done for performance purposes). The presented algorithm is sound and complete with respect to $\mathcal{L}$:

Theorem 10. *A change $c \in \mathcal{L}$ will be returned by Algorithm 1 iff c is detectable.*

Now suppose that the size of Δ is N. The look-up table used by $findPotential$-$Changes$ has a constant size, so it takes $O(1)$ time to search it. For each matching low-level change (left column in Table 4), a full search of the Δ is made for finding out the required low-level changes (middle column) by using a hash table, so it takes $O(N)$ time (worst-case). This determines the potential changes to be put in $potC$, per the right column of Table 4. Since the table is of constant size, the size of $potC$ will be $O(1)$ as well; therefore, computing $potC$ takes $O(N)$ in total.

For each change in $potC$, we need to determine whether its conditions are true. The time required for this depends on the change considered. For some changes (e.g., *Delete_Domain*), it takes $O(1)$ number of checks; for others, the cost is either $O(M)$ (e.g., *Delete_Class*) or $O(M^2)$ (e.g., *Reclassify_Individual_Higher*), where M is the number of triples in δ_1 and δ_2 of the respective high-level change. Note that each indi-

vidual check can be done in $O(1)$, a result which can be achieved using sophisticated labeling algorithms, as described in [3].

To calculate the complexity of the algorithm, we will consider the worst-case scenario. The for loop (line 2) iterates over the low-level changes. Let us consider the i-th iteration: for the selected change, we need $O(N)$ time for $findPotentialChanges$, plus $O(1)$ iterations of $O(M_i^2)$ cost (lines 4-10), where M_i is the total size of δ_1 and δ_2 for the high-level change considered. Then, the total cost (for the entire algorithm) is: $O(\sum_{i=1,\ldots,N}(N + M_i^2))$. However, note that: $\sum_{i=1,\ldots,N}(N + M_i^2) = N^2 + \sum_{i=1,\ldots,N} M_i^2 \leq N^2 + (\sum_{i=1,\ldots,N} M_i)^2$. The sum in the last equation cannot exceed the size of Δ by more than a constant factor (i.e., it is $O(N)$). Therefore, the complexity of the algorithm is $O(N^2)$. As a final note, recall that the cost of computing $\Delta(V_1, V_2)$ is linear with respect to the larger of V_1, V_2. Thus:

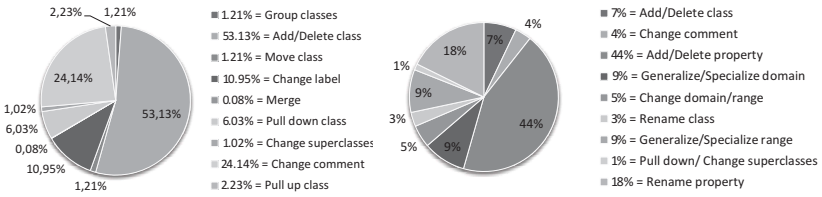
Theorem 11. *The complexity of Algorithm 1 for input V_1, V_2 is $O(max\{N_1, N_2, N^2\})$, where N_i is the size (in triples) of V_i $(i = 1, 2)$ and N is the size of $\Delta(V_1, V_2)$.*

In practice, our algorithm will rarely exhibit the quadratic worst-case complexity described in Theorem 11. There are several reasons for that. First of all, the complexity of searching through Δ (in $findPotentialChanges$) was calculated to be $O(N)$; for most changes, this will be $O(1)$ on average, due to the use of hash tables. Secondly, evaluating the conditions (line 5) varies from constant to quadratic over M_i, depending on the type of changes in $potC$. Furthermore, even though M_i may, in the worst case, be comparable to N, this will rarely be the case; therefore, even operations that exhibit quadratic complexity over M_i, will rarely exhibit quadratic complexity over N. The above arguments appear more emphatically for basic changes, as the cost of evaluating the conditions of any basic change is $O(1)$. The above observations will be verified by the results of our experiments (Section 5).

4 Operations Based on Heuristics

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

For evaluation purposes, we implemented a simple matcher that associates elements based on the similarity of their "neighborhoods", i.e., the sets of nodes and links that are pointing from/to the elements under question. If the similarity exceeds a certain threshold, then a matching is reported. In particular, if an element in V_1 is matched with an element in V_2, we detect a *Rename* operation, whereas if it is matched with a set of



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

Fig. 2. Overview of Composite and Heuristic Changes

elements in V_2, we detect a *Split* operation; on the other hand, if a set of elements in V_1 are matched with an element in V_2, a *Merge* operation is detected.

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

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

The focus of this paper is *not* on developing a sophisticated matcher, but on change detection. Our design was modular, so that any custom-made or off-the-shelf matcher could be used to calculate the required matchings; moreover, the user may choose to circumvent the matching process altogether. Thus, the matching process can be viewed as an optional, pre-processing phase to the actual change detection algorithm, and is an extension of our basic framework.

5 Experimental Evaluation

The evaluation of our approach was based on experiments performed on two well-established ontologies from the cultural (CIDOC [4]) and biological (GO [7]) domains. It aims at showing the intuitiveness and conciseness of the changes contained in $\mathcal{L}$ (Figure 2 and Table 5) as well as verifying that the performance of the implemented algorithm conforms to the average-case analysis of Section 3.3 (Table 6).

CIDOC consists of nearly 80 classes and 250 properties, but has no instances. For our experiments, we used versions v3.2.1 (dated 02.2002), v3.3.2 (dated 10.2002),

v3.4.9 (dated 12.2003), v4.2 (dated 06.2005) and v5.0.1 (dated 04.2009), which are encoded in RDF and are available in [4]. The detected changes apply mostly on properties, and many involve the heuristic change *Rename* (see Figure 2). For the detection of the heuristic changes a special-purpose matcher was developed, that exploited CIDOC's naming policy which attaches a unique, change preserving ID in the names; this way, the precision and recall of the matchings for CIDOC was 100%. CIDOC versions are accompanied by release notes describing in natural language the differences with the previous version; our algorithm uncovered certain typos, omissions and other mistakes in these notes, which were verified by one of CIDOC's editors. This highlights the need for automated change detection algorithms, as even the most careful manual recording process may be inaccurate.

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

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

Table 5. Evaluation Results

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

Table 5 shows the number of detected changes between different pairs of CIDOC and GO versions. The columns report the compared versions and their sizes, the size of Δ and the number of detected basic (only) and high-level (in general, i.e., basic, composite and heuristic) changes. The number of detected basic changes is comparable to the size of Δ, showing that deltas consisting entirely of basic changes are not concise. On the other hand, the size of the delta is significantly reduced (44%-78% for CIDOC, 59%-74% for GO) when composite and heuristic changes are also considered.

Table 6 reports the running time of the detection algorithm, measured on a Linux machine equipped with a Pentium 4 processor running at 3.4GHz and 1.5GB of main memory. The times for the detection of basic changes were, in general, linear to the input verifying our average-case analysis in Section 3. With respect to composite changes, the execution time reveals some interesting anomalies. For example, comparing the results for versions v3.3.2-v3.4.9 and v3.4.9-v4.2 (for CIDOC) we see a reduction in the running time, despite the increase of the input size (cf. Table 5). This is due to the very small number of detected composite changes for the second input (see Table 5). Also, when comparing the results of v16.12.08-v24.03.09 to v25.11.08-v16.12.08 (GO) we see that the running time increases in a sub-linear fashion with respect to the input. This can be explained by considering the types of detected composite changes, which reveals that for versions v16.12.08-v24.03.09 the changes whose complexity for evaluating the conditions is quadratic are 4.5% of the total, whereas for v25.11.08-v16.12.08 such changes constitute 15% of the total. The slow execution times related to heuristic changes is due to the overhead caused by the employed matcher.

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

Table 6. Running Time

6 Related Work

Change detection algorithms in the literature report either low-level deltas ([23,24]), or high-level ones, which, like in our paper, are usually distinguished in basic and composite ([15,17,22]). In [10,14,15,18,19,21] authors describe several operations and the intuition behind them. However, a formal definition of the semantics of such operations

([10,14,15,19]), or of the corresponding detection process ([15]), is usually missing; thus, they cannot guarantee any useful formal properties.

Authors in [10,14] describe a fixed-point algorithm for detecting changes, which is implemented in PromptDiff, an extension of Protégé [8]. The algorithm incorporates heuristic-based matchers in order to detect the changes that occurred between two versions. Therefore, the entire detection process is heuristic-based, thereby introducing 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 [18] the Change Definition Language (CDL) is proposed as a means to define a language of high-level changes. In CDL, a change is defined and detected using temporal queries over a version log that contains recordings of the applied low-level changes. The version log is updated when a change occurs which overrules the use of this approach in non-curated or distributed environments. In our work, version logs are not necessary for the detection, as the low-level delta can be produced also *a posteriori*. Note also that, in [18] changes that require heuristics for their detection (such as *Rename*) are completely ignored. This reduces the usefulness of the proposed language.

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

7 Conclusion and Future Work

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

In this paper, we addressed the problem of automatically identifying deltas. We proposed a formal framework and used it for defining a language of high-level changes for both schema and data, $\mathcal{L}$, and an algorithm that correctly detects changes from $\mathcal{L}$. We proved that $\mathcal{L}$ satisfies several intuitive properties (*Completeness*, *Non-ambiguity*, *Reversibility*). Note that the existence of other languages satisfying these properties is not ruled out. However, if the intuitiveness of the changes is not taken into account, the languages will end up being artificial and without practical use in real-world scenarios. Thus, the intuitiveness of the changes that $\mathcal{L}$ contains was a critical factor in our design and experimental evidence on the usefulness of $\mathcal{L}$ was provided. The detection algorithm itself was shown to be quite efficient, namely of quadratic worst-case complexity (even though, in practice, it seems to exhibit linear average-case complexity). The approach can be extended to more expressive ontology languages but the details depend on the semantics of the language and must be determined. As future work, we plan to extend $\mathcal{L}$ by considering *complex changes*, which aggregate several composite changes together. Moreover, we plan to conduct empirical studies involving real users.

Acknowledgements

This work was partially supported by the EU project KP-Lab (FP6-2004-IST-4).

References

1. A. Bairoch, R. Apweiler, C.H. Wu, W.C. Barker, B. Boeckmann, S. Ferro, E. Gasteiger, H. Huang, et al. The Universal Protein Resource (UniProt). *Nucleic Acids Research*, 2005.
2. D. Brickley and R.V. Guha. Rdf vocabulary description language 1.0: Rdf schema. www.w3.org/TR/2004/REC-rdf-schema-20040210, 2004.
3. V. Christophides, G. Karvounarakis, D. Plexousakis, M. Scholl, and S. Tourtounis. Optimizing taxonomic semantic web queries using labeling schemes. *Web Semantics: Science, Services and Agents on the WWW*, 2004.
4. CIDOC-CRM. http://cidoc.ics.forth.gr/official_release_cidoc.html.
5. R. Cloran and B. Irwin. Transmitting rdf graph deltas for a cheaper semantic web. In *Proc. of SATNAC*, 2005.
6. J. Euzenat and P. Shvaiko. *Ontology Matching*. Springer-Verlag, 2007.
7. D.P. Hill, B. Smith, M.S. McAndrews-Hill, and J.A. Blake. Gene ontology annotations: What they mean and where they come from. *BMC Bioinformatics*, 2008.
8. http://protege.stanford.edu. Protege Project, 2002.
9. http://www.hozo.jp/. Hozo.
10. M. Klein. *Change Management for Distributed Ontologies*. PhD thesis, Vrije Univ., 2004.
11. VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In *Soviet Physics-Doklady*, volume 10, 1966.
12. B. McBride, F. Manola, and E. Miller. Rdf primer. www.w3.org/TR/rdf-primer, 2004.
13. N.F. Noy, A. Chugh, W. Liu, and M.A. Musen. A Framework for Ontology Evolution in Collaborative Environments. In *Proc. of ISWC*, 2006.
14. N.F. Noy and M.A. Musen. PromptDiff: A Fixed-Point Algorithm for Comparing Ontology Versions. In *Proc. of AAAI*, 2002.
15. A. Palma, P. Haase, Y. Wang, and M. dAquin. D1.3.1 propagation models and strategies. Technical report, NeOn Deliverable D1.3.1, 2007.
16. V. Papavassiliou, G. Flouris, I. Fundulaki, D. Kotzinos, and V. Christophides. Formalizing high-level change detection for rdf/s kbs. Technical Report TR-398, FORTH-ICS, 2009.
17. P. Plessers and O. De Troyer. Ontology Change Detection Using a Version Log. In *Proc. of ISWC*, 2005.
18. P. Plessers, O. De Troyer, and S. Casteleyn. Understanding Ontology Evolution: A Change Detection Approach. *Web Semantics: Science, Services and Agents on the WWW*, 2007.
19. D. Rogozan and G. Paquette. Managing Ontology Changes on the Semantic Web. In *Proc. of IEEE/WIC/ACM on Web Intelligence*, 2005.
20. G. Serfiotis, I. Koffina, V. Christophides, and V. Tannen. Containment and minimization of rdf/s query patterns. In *Proc. of ISWC*, 2005.
21. L. Stojanovic. *Methods and Tools for Ontology Evolution*. PhD thesis, Univ. of Karlsruhe, 2004.
22. L. Stojanovic, A. Maedche, B. Motik, and N. Stojanovic. User-Driven Ontology Evolution Management. In *Proc. of EKAW. Ontologies and the Semantic Web*, 2002.
23. M. Volkel, W. Winkler, Y. Sure, S. Ryszard Kruk, and M. Synak. Semversion: A versioning system for rdf and ontologies. In *Proc. of ESWC*, 2005.
24. D. Zeginis, Y. Tzitzikas, and V. Christophides. On the Foundations of Computing Deltas Between RDF Models. In *Proc. of ISWC+ ASWC*, 2007.
25. A.V. Zhdanova. Community-Driven Ontology Evolution: Gene Ontology Case Study. In *Proc. of BIS*, 2008.