Journal of Biomedical Informatics 46 (2013) 15–32

Contents lists available at SciVerse ScienceDirect

Journal of Biomedical Informatics

journal homepage: www.elsevier.com/locate/yjbin

COnto–Diff: generation of complex evolution mappings for life science ontologies ☆

Michael Hartung *, Anika Groß, Erhard Rahm

Interdisciplinary Center for Bioinformatics, University of Leipzig, Härtelstraße 16-18, 04107 Leipzig, Germany
Department of Computer Science, University of Leipzig, P.O. Box 100920, 04009 Leipzig, Germany

ARTICLE INFO

Article history:
Received 25 July 2011
Accepted 7 April 2012
Available online 8 May 2012

Keywords:
Ontology evolution
Ontology versions
Diff
Life science ontologies

ABSTRACT

Life science ontologies evolve frequently to meet new requirements or to better reflect the current domain knowledge. The development and adaptation of large and complex ontologies is typically performed collaboratively by several curators. To effectively manage the evolution of ontologies it is essential to identify the difference (Diff) between ontology versions. Such a Diff supports the synchronization of changes in collaborative curation, the adaptation of dependent data such as annotations, and ontology version management. We propose a novel approach **COnto–Diff** to determine an expressive and invertible diff evolution mapping between given versions of an ontology. Our approach first matches the ontology versions and determines an initial evolution mapping consisting of basic change operations (insert/update/delete). To semantically enrich the evolution mapping we adopt a rule-based approach to transform the basic change operations into a smaller set of more complex change operations, such as merge, split, or changes of entire subgraphs. The proposed algorithm is customizable in different ways to meet the requirements of diverse ontologies and application scenarios. We evaluate the proposed approach for large life science ontologies including the Gene Ontology and the NCI Thesaurus and compare it with PromptDiff. We further show how the Diff results can be used for version management and annotation migration in collaborative curation.

1. Introduction

Ontologies have become increasingly important, e.g., to semantically and consistently annotate and categorize information. In life sciences, large biomedical ontologies such as the Gene Ontology (GO) [12] are used to describe the functions of genes or proteins, e.g., in SwissProt [5] or Ensembl [10]. The ontological information is utilized in many analytical studies such as for term enrichment analysis [26,36]. The number of life science ontologies is continuously growing. For instance, the Open Biological and Biomedical Ontologies Foundry (OBO) [40] or the BioPortal [29] currently provide access to about 250 ontologies covering knowledge from different domains such as anatomy, phenotype, biological function or biochemistry.

Most life science ontologies evolve heavily to meet new requirements, correct previous design errors or better incorporate new domain knowledge [17]. Hence there is a continuous release of new versions of an ontology that reflect the latest changes. The analysis of the version history reveals that important ontologies such as the Gene Ontology or the NCI Thesaurus [39] doubled their size since 2004 [16]. Large ontologies are typically developed and adapted collaboratively by several curators and experts [30,44]. For example, the Gene Ontology is maintained by a consortium with members from several international organizations and projects[1]. While ontology developers typically focus on areas of their expertise it is still valuable to them to know the overall changes of the ontology that have already been applied.

In this paper we propose a new approach to automatically determine the changes between two given versions of an ontology. The changes are collected within a so-called diff evolution mapping which is helpful for ontology developers, ontology users and applications:

- Ontology developers can see from the diff result how the ontology has evolved and what changes are dominating. In collaborative ontology development, it is important for both the coordinators as well as individual developers to exactly know what changes have already been performed and what changes may still be missing. Previous changes may in fact serve as a starting point for further modifications. Early approaches such

☆ This work is supported by the German Research Foundation (DFG), Grant RA 497/18-1 (“Evolution of Ontologies and Mappings”).

* Corresponding author at: Department of Computer Science, University of Leipzig, P.O. Box 100920, 04009 Leipzig, Germany. Fax: +49 0341 97 32209.

E-mail addresses: hartung@izbi.uni-leipzig.de (M. Hartung), gross@informatik.uni-leipzig.de (A. Groß), rahm@informatik.uni-leipzig.de (E. Rahm).

[1] GO Consortium: http://www.geneontology.org/GO.consortiumlist.shtml

1532-0464/$ - see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jbi.2012.04.009

as the Protégé plugin PromptDiff [32] already show the need to determine ontology changes to support collaborative ontology development.

- The computed evolution mapping can be used to find out whether existing applications or analysis studies are affected by ontology changes so that they may have to be adapted or redone. For instance, ontology changes may invalidate previous findings of a term enrichment analysis [21] so that the analysis may have to be repeated.
- The evolution mapping can support the propagation of ontology changes to ontology-dependent artifacts such as annotations or ontology mappings. For instance, curators of annotations can quickly find and remove outdated annotations referring to deleted or obsolete ontology concepts [16]. Annotation curators may also collaboratively decide about how to deal best with ontology changes.
- Evolution mappings can be used to optimize the management of ontology versions by only storing the version differences instead of the entire ontology versions [23].

Our approach to determine the Diff between ontology versions is based on a previous matching of ontology versions which determines semantic correspondences between equivalent concepts. There has been a huge amount of research on semi-automatic ontology matching in the last decade [9,38] (see Section 2), and we can therefore leverage existing approaches to (semi-) automatically find correspondences. While Match focuses on the unchanged ontology elements, Diff also has to consider added and deleted ontology elements. Simple changes on individual ontology elements (concepts, relationships) and their additions/ deletions are relatively easy to find and supported by current developer tools such as OBO-Edit [8] or OBO Explorer [1]. However, we observe that such low-level mappings represented by long lists of primitive changes are of limited usefulness for human users. Especially for large ontologies it becomes difficult for individual curators to understand the semantics behind changes performed by others. We therefore aim at a much more compact and semantically more expressive diff representation capturing complex ontology changes such as merging, splitting and moving of ontology concepts or adding and deleting entire subgraphs. Such mappings are likely to serve much better the discussed purposes of evolution mappings, in particular for ontology developers and ontology users.

For illustration we use the running example in Fig. 1 on the evolution of a part of an anatomy ontology. The goal is to derive the evolution mapping between the two ontology versions. A basic Diff approach only supporting add/delete/change operations for individual ontology elements would derive a deletion of the 'accessory cochlear nucleus' and 'anterior cochlear nucleus' concepts although these concepts are actually merged into concept 'cochlear ventral nucleus'. We propose a more expressive Match-based Diff generation supporting complex changes. In Fig. 1, the names of the white concepts remain unchanged during the evolution and we assume that correspondences between these concepts are part of the match result. Dashed lines indicate further relevant matching concepts. The correspondences help our Diff approach to correctly determine that the redundant concepts 'accessory cochlear nucleus' and 'anterior cochlear nucleus' have been merged into 'cochlear ventral nucleus'. The original redundancy could have been introduced by different developers and this is now corrected by a merge change. In such cases the labels of the old concepts may be introduced as synonyms in the new concept. This information can be exploited to detect concept merges. Another complex change is the addition of the sub-ontology rooted at 'brain stem white matter'. Especially for large life science ontologies it is valuable to identify such larger changed ontology portions. Further complex changes include the move of concepts 'trigeminal sensory nucleus' and 'trigeminal motor nucleus' from 'brainstem nucleus' to the new inner concept 'trigeminal nucleus'.

The contributions of this paper are as follows:

- We introduce a model for diff evolution mappings between ontology versions that is based on a set of basic and complex change operations. Based on this model we present a generic Diff algorithm, **COnto–Diff** (Complex Ontology Diff), to automatically determine expressive (compact) diffs between given ontology versions. The approach is based on the result of a (semi-) automatic match operation. The evolution mapping is computed by a rule-based approach utilizing so-called COG (change operation generating) rules.
- We propose an algorithm using the evolution mapping to migrate an old ontology version to the newer one. We can also derive an inverse evolution mapping to migrate from a changed ontology version back to the original one. This underlines that evolution mappings are executable and useful for the management of ontology versions. We also discuss how our evolution mappings can be used to update annotations and thus support annotation curators.
- We comprehensively evaluate our approach for large life science ontologies including the Gene Ontology and the NCI Thesaurus, and provide a comparison with the PromptDiff approach. We also show that the determined diff mappings allow a correct migration between ontology versions. The results confirm the applicability and scalability of the approach to large ontologies with many changes.

We made the functionality of **COnto–Diff** available within a new web tool [15] so that users can interactively determine and analyze diffs between ontology versions. While **COnto–Diff** is based on a match step, matching is not the focus of this paper since it has already been widely investigated. We further focus on determining evolution mappings solely at the ontology level and leave related problems (e.g., instance migration) for future work.

We discuss related work in Section 2. In Section 3 we introduce our ontology model, the considered set of basic and complex

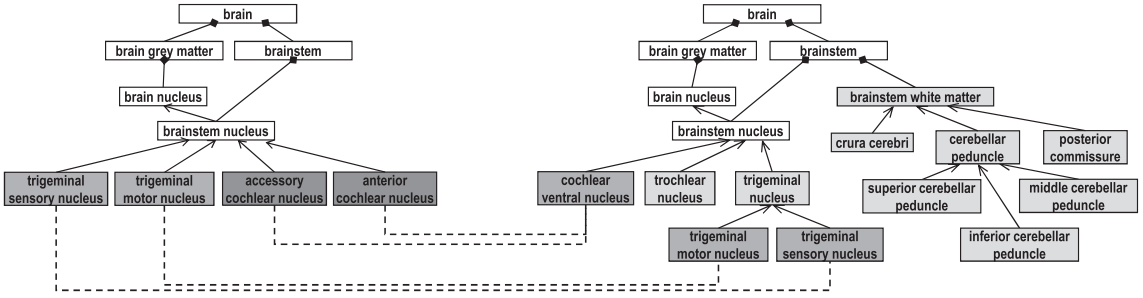

Fig. 1. Motivating example – evolution in a part of an anatomical ontology (left: old version, right: new version).

changes as well as our model of match mappings and evolution mappings. **COnto–Diff** is presented in Sections 4 and 5. Section 4 introduces the set of COG rules to determine the change operations that occurred during the evolution. Section 5 outlines the overall approach for determining the diff evolution mapping based on the introduced COG rules. The application of the diff evolution mappings for ontology and annotation migration is explained in Section 6. We present evaluation results in Section 7. We conclude with a summarizing discussion on the strengths and current limitations of our approach and on plans for future work. The appendix provides details such as correctness proofs for the proposed algorithms.

2. Related work

We first discuss related work on ontology evolution, in general. Before we describe previous approaches to determine ontology changes, we provide an overview of recent work on the related problem of ontology matching.

2.1. Ontology evolution

There is already a substantial amount of related work on ontology evolution (see [11,18] for surveys). The Protégé tool [30] supports both the development and evolution of ontologies. The evolution framework is based on a change and annotation ontology (CHAO) [31] including both simple and complex changes. Changes can be specified via logging of incremental changes or by a direct ontology version comparison (see PromptDiff later). The ontology evolution process itself has been extensively studied by Stojanovic et al. [43]. Oliver and colleagues [33] investigate the representation of typical changes in controlled medical terminologies including additions, renaming, hierarchical changes and merges. In contrast to these works, we do not consider the active modification of ontologies but focus on determining the changes between given versions.

Tools such as OBO-Edit [8] or OBO2OWL [28] enable the management and development of OBO ontologies. OBO-Edit contains a commandline tool obodiff [2] which can compare a pair of OBO files and reports all edits that are necessary to transform the first into the second file. Furthermore, information on changes of current ontologies (history tracking) is primarily limited to mailing lists and reports by the ontology distributors. For instance, the GO consortium summarizes changes on the Gene Ontology in a monthly report.[3] Most of the tools and systems only report on primitive changes, e.g., additions of concepts or relationships between succeeding versions of an ontology. In contrast **COnto–Diff** is able to detect more meaningful, complex changes and is not restricted to succeeding versions, i.e., the input versions may be from much different points in time.

Several studies have analyzed the evolution of life science ontologies and terminologies. Study [46] analyzes the evolution of GO with the help of simple change statistics on the number of concepts, relationships and paths. A more comprehensive study [17] analyzes the evolution of 16 life science ontologies and associated annotations. Ceusters et al. [6] have studied the evolution of SNOMED CT and found that the main changes are class modifications followed by class additions and deletions. They acknowledge the importance of a diff or history mechanism to analyze and enhance the quality of ontologies, e.g., by identifying mistakes of the past and avoiding them in the future.

[2] obodiff: http://oboedit.org/docs/html/obodiff.htm.
[3] GO Monthly Reports: http://www.geneontology.org/MonthlyReports/.

2.2. Ontology matching

Ontology matching is the process of determining a set of semantic correspondences (ontology mapping) between concepts of two ontologies O_1 and O_2. A manual matching by domain experts is very time-consuming and for large ontologies almost infeasible. Thus, many (semi-) automatic matching algorithms have been developed for ontology matching (see [9,37,38] for surveys). Common match approaches utilize the linguistic and structural similarity of ontology concepts; some approaches also consider the similarity of ontology instances. State-of-the art match systems such as COMA++ [2], Falcon [20] or SAMBO [25] combine multiple matchers within a match strategy to achieve better match quality. In general, automatically determined mappings are not perfect but should be verified and revised by human experts (semi-automatic matching). The revision process benefits from a collaborative involvement of multiple experts to speed-up mapping creation and to increase the included knowledge [27,47].

Previous work on matching life science ontologies showed that linguistic matching methods based on the similarity of concept names and synonyms produce very good results [13,14]. This work applies such approaches to enable a largely automatic ontology matching. We also exploit the fact that matching two versions of the same ontology is facilitated by the large portion of shared identifiers between two versions.

2.3. Ontology change detection

Most related to this paper are previous approaches for detecting ontology changes. They can be classified into *incremental* and *direct* ones. Incremental approaches are based on a version log of changes. For instance, [35] differentiates between simple and composite changes which are defined declaratively using a Change Definition Language (CDL). Based on an available change log, the approach aims at detecting additional implicit changes by evaluating the definition of specified change operations. However, such approaches require access to a version log of an ontology which is often not possible. Furthermore, the occurrence of redundant changes in a log makes change detection between two versions difficult. Hence, we consider a direct comparison of ontology versions to determine the diff as more versatile.

The best known approach of this kind is PromptDiff [32] that is part of the Protégé ontology management suite [30]. It uses several heuristic matchers (e.g., single unmatched sibling, unmatched inverse slots, or same type/name) to detect changes between two ontology versions. The algorithm works iteratively and applies the heuristic matchers as long as no more changes are found. The approach supports the detection of several basic as well as complex changes including concept additions, deletions, splits and merges. The result is a difference table which lists the detected changes. Each row represents a change operation including its parameters. Changes on entire subtrees can be recognized with the PromptDiff plugin in the Protégé UI which allows users to visualize tree-level changes [31], i.e., one can notice if all classes of a subtree have changed in the same way, e.g., all were added or deleted. We will provide a more detailed comparison between PromptDiff and **COnto–Diff** in the evaluation (see Section 7.4).

The OntoView system [24] focuses on versioning of RDF-based ontologies. It can detect simple changes such as label modifications as well as logical definition changes, e.g., changes on subClassOf relationships or domain/range properties. The algorithm is based on RDF triples and uses a graph representation as well as a set of IF–THEN statements to detect changes between versions. More complex changes such as merges or splits are not supported.

The authors of [34] describe an approach to detect high-level changes such as moveClass in RDF/S knowledge bases. The pro-

posed framework distinguishes between basic, composite and heuristic changes which are expressed in a formal language. The main algorithm focuses on the detection of basic and composite changes while the detection of heuristic changes such as rename, merge, or split is considered as optional. The starting point is a so-called low-level delta containing RDF triples that have been added/deleted between input versions V1 and V2. Changes are described by (1) required added RDF triples, (2) required deleted RDF triples and (3) a set of conditions that need to be fulfilled. The detection algorithm first uses the low-level delta and the defined change descriptions to generate a set of possible changes between V1 and V2. Afterwards the algorithm iteratively selects change operations that meet the conditions and reduces the low-level delta set accordingly. The algorithm first tries to detect composite and then basic changes. Changes on entire subgraphs cannot be recognized by the approach.

In contrast to these previous change detection approaches we adopt a two-phase Diff approach that starts with a match processing to determine corresponding concepts in the input ontology versions. We then apply a rule-based approach to determine complex ontology changes. Ontology matching is not performed in OntoView while PromptDiff employs matching in its heuristic matchers. [34] utilizes ontology matching only as an optional step but not in the main algorithm. Compared to all other approaches we distinguish between matching and diff computation and can thus flexibly adapt our algorithm to work with different ontologies or to utilize enhanced matching techniques. The proposed rule-based Diff computation supports a large set of change operations and can easily be adapted to deal with further types of changes.

3. Models and problem statement

We first introduce the assumed ontology model and describe the change operations that may be applied during ontology evolution. We then define the notions of match and evolution mappings. Finally, we state the specific problem we address with the **COnto–Diff** approach.

3.1. Ontology model and versions

An ontology $O = (C, A, R)$ consists of concepts C having associated attributes of A. The concepts are interrelated by directed relationships of R. Each ontology concept has a special attribute called *accession* which is used to unambiguously identify a concept within the ontology. Concepts can have optional concept attributes $a = (a_{concept}, a_{name}, a_{value}) \in A$ which semantically describe the concept in more detail. For instance, life science ontologies often provide synonyms for a concept and use an 'obsolete' attribute to mark outdated concepts.

R consists of a set of directed relationships $r = (r_{source}, r_{type}, r_{target})$ of type r_{type} interconnecting concepts r_{source} and r_{target}. The most important relationship type in life science ontologies is 'is_a' describing a subsumption relation between two concepts. Another common relationship type is 'part_of' denoting an inclusion (parthood) relation. The ontology concepts connected by 'is_a' and 'part_of' relationships form a directed acyclic graph (DAG). Between two concepts c_1 and c_2 we allow at most one 'is_a' or one 'part_of' relationship, i.e., two coexistent relationships (c_1, is_a, c_2) and (c_1, part_of, c_2) are not allowed. However, concepts may have additional domain-specific directed relationships [41] such as the types 'positively_regulates' or 'negatively_regulates' used in the Gene Ontology.

Our ontology model is motivated by life science ontologies that typically have no directly associated instance data. Instead sources such as Ensembl [10] or SwissProt [5] use ontology concepts to uniformly describe the properties of their objects (annotation). The model is comparable to the OBO format [7] used to develop and distribute many life science ontologies [29,40]. Recent studies [28,45] showed how the OBO format is related to Semantic Web languages such as RDF/S and OWL and the findings also hold for our ontology model. RDF/S and OWL ontologies can be transformed into our model as follows: OWL classes correspond to our concepts, RDF/S 'subClassOf' to 'is_a' relationships, and RDF/S labels to the concept names. RDF/S properties can be converted into attributes, e.g., synonyms, or further relationships such as 'part_of'. Thus, we can also determine diffs for OWL ontologies. However, not all OWL constructs can be mapped into appropriate OBO constructs (e.g., property restrictions) so that we currently do not support the full OWL language.

An ontology version $O_v = (C_v, A_v, R_v)$ of version v represents a snapshot of an ontology at a specific point in time. The elements of O_v are assumed to be valid until a new ontology version is released. Ontology providers distribute new releases at regular time intervals or whenever a significant number of changes has been incorporated. For instance, the GO Consortium daily releases a new version of the popular Gene Ontology.

3.2. Change operations

We consider two sets of change operations for ontology evolution: basic changes (set B_{op}) and complex changes (set C_{op}). The presented operations are supported by our current design and implementation. While the sets of possible changes are already comprehensive our approach is flexible and customizable by supporting the addition of further change operations to deal with specific requirements.

Table 1 gives an overview of the changes that are currently supported by **COnto–Diff**. *Basic changes* are applied on a single concept, attribute or relationship and deal with either a map (change), addition or deletion resulting in nine operations displayed in the upper part of the table. All other changes are called complex changes and are shown in the middle/ lower part of the table. As we will see they are based on basic changes or other complex changes and thus specify changes at a higher level of abstraction. Some of the complex changes operate on single elements (denoted with lower case letters), e.g., *substitute*. Most of our complex changes refer to multiple ontology elements (sets denoted with upper case letters), e.g., *merge* or *addSubGraph*. For example, the merge of source concepts 'accessory cochlear nucleus' and 'anterior cochlear nucleus' into target concept 'cochlear ventral nucleus' in our running example can be described as

merge({accessory cochlear nucleus, anterior cochlear nucleus}, cochlear ventral nucleus).

Some of the changes such as *merge*, *split* and *move* are generally useful and not limited to life science ontologies. Complex changes w.r.t. entire subontologies such as *addSubGraph* are especially valuable for large life science ontologies to compactly describe the evolution. Finally, the changes *toObsolete* and *revokeObsolete* are useful to identify the concepts that should not be used anymore or for which the obsolete status has been withdrawn.

Complex change operations can be implemented by a series of basic change operations. For each complex change we maintain the simpler underlying changes (see end of Section 4.3 for more details). Furthermore, each change operation has an inverse that undoes the effect of the change. For instance, the inverse of *merge*(*Source_C*, *target_c*) is *split*(*target_c*, *Source_C*), i.e., a single source concept is split into multiple target concepts (for the inverses of all change operations see Table 1). This symmetry of

Table 1
COnto–Diff operations with descriptions and their inverses. The upper part shows all basic change operations (B_{op}) while all complex change operations (C_{op}) operating whether on single elements (denoted with lower case letters) or multiple elements (sets denoted with upper case letters) are displayed in the middle and lower part, respectively.

Change operation	*Description*	*Inverse change operation*
addC(*c*)	Insertion of a new concept *c* in the changed ontology	*delC*(*c*)
delC(*c*)	Deletion of an existing concept *c* from the old ontology version	*addC*(*c*)
mapC(*c*1, *c*2)	Maps a concept *c*1 of the first ontology version to a concept *c*2 of the second version	*mapC*(*c*2, *c*1)
addR(*r*)	Insertion of a new relationship *r*	*delR*(*r*)
delR(*r*)	Deletion of an existing relationship *r*	*addR*(*r*)
mapR(*r*1, *r*2)	Maps a relationship *r*1 of the first ontology version to a (differently typed) relationship *r*2 of the second version	*mapR*(*r*2, *r*1)
addA(*a*)	Addition of an attribute *a*	*delA*(*a*)
delA(*a*)	Deletion of an existing attribute *a*	*addA*(*a*)
mapA(*a*1, *a*,2)	Maps an attribute *a*1 of the first to an attribute *a*2 of the second ontology version	*mapA*(*a*2, *a*1)
substitute(*c*1, *c*2)	Concept *c*1 is replaced by *c*2	*substitute*(*c*2, *c*1)
toObsolete(*c*)	Concept c becomes obsolete, i.e., it should not be used anymore	*revokeObsolete*(*c*)
revokeObsolete(*c*)	The obsolete status of c is revoked, i.e., it becomes active again	*toObsolete*(*c*)
move(*c*, *C_To*, *C_From*)	Moves a concept c and its subgraph from concept set *C_From* to the concept set *C_To*	*move*(*c*, *C_From*, *C_To*)
chgAttValue(*c*, *att_name*, *V_Old*, *V_New*)	Changes the set of values of the *att_name* attribute of a concept *c* from *V_Old* to *V_New*	*chgAttValue*(*c*, *att_name*, *V_New*, *V_Old*)
addLeaf(*c*, *C_Parents*)	Insertion of a leaf concept *c* below the concepts in *C_Parents*	*delLeaf*(*c*, *C_Parents*)
delLeaf(*c*, *C_Parents*)	Deletion of a leaf concept *c* situated below the concepts in *C_Parents*	*addLeaf*(*c*, *C_Parents*)
merge(*Source_C*, *target_c*)	Merges multiple source concepts *Source_O* into one target concept *target_c*	*split*(*target_c*, *Source_C*)
leafMerge(*Child_C*, *parent_c*)	Special merge that fuses all child concepts *Child_C* into their parent concept *parent_C*(*parent_c* becomes a leaf)	*leaf Split*(*parent_c*, *Child_C*)
split(*source_C*, *Target_C*)	Splits one source concept *source_c* into multiple target concepts *Target_C*	*merge*(*Target_C*, *source_c*)
leafSplit(*parent_c*, *Child_C*)	Special split that refines a leaf concept *parent_c* by multiple child concepts *Child_C*(*parent_c* becomes an inner concept)	*leaf Merge*(*Child_C*, *parent_c*)
addSubGraph(*c_root*, *C_Sub*)	Inserts a new subgraph with root *c_root* and concepts *C_Sub* connected by 'is_a' and 'part_of' relationships	*delSubGraph*(*c_root*, *C_Sub*)
delSubGraph(*c_root*, *C_Sub*)	Removes an existing subgraph with root *C_root* and concepts *C_Sub* connected by 'is_a' and 'part_of' relationships	*addSubGraphi*(*c_root*, *c_Sub*)

change operations allows us to derive for every change operation the associated inverse change operation.

3.3. Match and evolution mappings

We represent changes between two ontology versions O_{old} and O_{new} as a mapping. In model management [3,4] a mapping $map(O_{old}, O_{new})$ connects elements of an old ontology version O_{old} with elements of a new version O_{new}. We distinguish between a match mapping $match(O_{old}, O_{new})$ and an evolution mapping $diff(O_{old}, O_{new})$. Match mappings represent semantic correspondences between two ontology versions and thus interrelate unchanged elements as well as changed but corresponding (semantically equivalent or related) ontology elements. For our purpose, we only require simple match mappings consisting of correspondences interlinking two concepts each, i.e., $match(O_{old}, O_{new}) = \{matchC(c1, c2) | c1 \in O_{old}, c2 \in O_{new}\}$.

By contrast an evolution mapping highlights the differences and covers all changes that occurred between two ontology versions. Unchanged ontology elements included in a match mapping are not part of an evolution mapping. Diff mappings can contain all change operations as introduced in the previous section: $diff(O_{old}, O_{new}) = \{chgOp(e1, \ldots) | chgOp \in B_{Op} \cup C_{Op}\}$. The simplest kind of diff mapping, $diff_{basic}$, only contains basic change operations, i.e., map, add and delete operations: $diff_{basic}(O_{old}, O_{new}) = \{chgOp(e1, \ldots) | chgOp \in B_{Op}\}$. However, the main goal is to derive a semantically expressive diff specifying the occurred evolution by complex changes as much as possible. This final $diff_{compact}$ therefore should contain only the semantically most expressive change operations which are not part of any other change operation. As the name suggests, $diff_{compact}(O_{old}, O_{new})$ will generally have fewer operations than the corresponding $diff_{basic}$ since a complex change typically replaces several basic changes. We also want to determine the inverse evolution mapping that can be used to migrate O_{new} to O_{old}. We will use the inverse of the change operations in $diff(O_{old}, O_{new})$ to create the inverse mapping and show that it is equivalent to $diff(O_{new}, O_{old})$.

3.4. Problem statement

The problem that we investigate in this paper is the following. For two given ontology versions O_{old} and O_{new} of the same ontology and a match mapping $match(O_{old}, O_{new})$ the task is to compute the basic evolution mapping $diff_{basic}(O_{old}, O_{new})$ and a semantically expressive evolution mapping $diff_{compact}(O_{old}, O_{new})$ as well as their inverse mappings. The Diff algorithm should be able to recognize any defined change operation. The evolution mappings should be complete and minimal. In particular, they should contain all changes between the two input versions so that the new (old) ontology version can be constructed from the old (new) ontology version and the (inverse) diff evolution mapping. Furthermore, $diff_{basic}(O_{old}, O_{new})$ should only contain those basic changes that are required for a correct version migration. Analogously, $diff_{compact}(O_{old}, O_{new})$ should only contain change operations that are not included in any other complex change operation, i.e., the most compact set of changes between O_{old} and O_{new} w.r.t. the defined set of operations. The algorithms should also be scalable to large life science ontologies.

4. Change operation generating rules

The identification of basic and complex change operations is based on **C**hange **O**peration **G**enerating Rules (COG rules). Each rule is defined by a set of pre-conditions. If all pre-conditions are fulfilled, a sequence of resulting actions is applied to **create** new change operations or **eliminate** existing ones. Depending on the type of generated change operations we distinguish between (1) *Basic* and (2) *Complex* COG rules. We further use (3) *Aggregation* rules to iteratively determine more complex change operations for sets of ontology elements. The various rules will be used by the main Diff algorithm presented in Section 5.

In the following we describe the different types of rules in more detail and provide examples for illustration. Please consult Appendix A for the complete set of our current COG rules. In the rule def-

initions we denote single elements of an ontology with lower case letters ($a, b, \ldots \in O$), and element sets with upper case letters ($A, B, \ldots \subseteq O$). Each rule has an unique number also indicating the rule type rule ($b_1, b_2, \ldots$ for basic, $c_1, c_2, \ldots$ for complex, and $a_1, a_2, \ldots$ for aggregation COG rules). The rule numbers reflect dependencies between different rules that require a partial order in which rules have to be applied. For instance, rules (b_6) and (b_7) must be applied before (b_8) since they create change operations (*addR*, *delR*) which are needed by (b_8) to create *mapR* changes. In some rules, we test the equality (=sign) or inequality ($\neq$sign) of two elements. We regard two concepts as equal if they share the same identifier, i.e., they have exactly the same accession number or URI (disregarding namespace and versioning information though).

4.1. Basic COG rules

The basic COG rules (b-COG) primarily use information from the match mapping and the ontology versions to determine basic change operations. The following five b-COG rules are used to determine *addC*, *delC* and *mapC* change operations:

$$(b_1) \quad c \in O_{new} \wedge \nexists a(a \in O_{old} \wedge matchC(a,c)) \rightarrow \textbf{create}[addC(c)]$$

$$(b_2) \quad c \in O_{old} \wedge \nexists a(a \in O_{new} \wedge matchC(c,a)) \rightarrow \textbf{create}[delC(c)]$$

$$(b_3) \quad a \in O_{old} \wedge b \in O_{new} \wedge matchC(a,b) \wedge a \neq b \wedge \neg isObsolete(a) \wedge \neg isObsolete(b) \rightarrow \textbf{create}[mapC(a,b)]$$

$$(b_4) \quad a \in O_{old}, O_{new} \wedge matchC(a,a) \wedge \exists b(b \in O_{new} \wedge matchC(a,b) \wedge a \neq b) \wedge \neg isObsolete(a) \wedge \neg isObsolete(b) \rightarrow \textbf{create}[mapC(a,a)]$$

$$(b_5) \quad a \in O_{old}, O_{new} \wedge matchC(a,a) \wedge \exists b(b \in O_{old} \wedge matchC(b,a) \wedge a \neq b) \wedge \neg isObsolete(a) \wedge \neg isObsolete(b) \rightarrow \textbf{create}[mapC(a,a)]$$

In our running example (b_1) determines concept additions (*addC*) such as for 'trochlear nucleus' and 'trigeminal nucleus'. (b_3) creates *mapC* changes that map between changed concepts, e.g., *mapC* (accessory cochlear nucleus, cochlear ventral nucleus). Furthermore, (b_4) and (b_5) look for concepts that have multiple matches to others and create corresponding *mapC* changes. The existence of multiple correspondences implies a changed semantics for the concept that is expressed in the evolution mapping. Table A.1 lists six further b-COG rules to determine relationship- and attribute-level changes.

4.2. Complex COG rules

Complex COG rules (c-COG) are non-recursive and determine the complex changes based on either basic change operations or other complex changes. Complex changes on sets of elements are generally derived in two steps. We first apply c-COG rules to create complex changes on single ontology elements and then use an additional aggregation step (using aggregation rules) to combine several element changes into complex changes on set-valued parameters.

For example, for the complex merge operation we first derive partial merge operations on a single input element using the following c-COG rule:

$$(c_8) \quad a, b \in O_{old} \wedge c \in O_{new} \wedge mapC(a,c) \wedge mapC(b,c) \wedge a \neq b \wedge \nexists d(d \in O_{new} \wedge mapC(a,d) \wedge c \neq d) \wedge \nexists e(e \in O_{new} \wedge mapC(b,e) \wedge c \neq e) \rightarrow \textbf{create}[merge(\{a\},c), merge(\{b\},c)], \textbf{eliminate}[mapC(a,c), mapC(b,c)]$$

The left side of the rule ensures that there exist at least two different source concepts a and $b (a \neq b)$ mapping to the same target concept c, and that a and b have no further maps to other target concepts d and e, respectively. If these pre-conditions are fulfilled we create two element-level merge change operations one from concept a into concept c and one from b into c, the corresponding basic changes *mapC*(a, c) and *mapC*(b, c) are eliminated.

For our example *mapC*(accessory cochlear nucleus, cochlear ventral nucleus) and *mapC*(anterior cochlear nucleus, cochlear ventral nucleus) would produce change operations *merge*({accessory cochlear nucleus}, cochlear ventral nucleus) and *merge*({anterior cochlear nucleus}, cochlear ventral nucleus).

4.3. Aggregation COG rules

Aggregation COG rules (a-COG) are used to determine all affected elements in set-valued complex change operations. Particularly, several related element-level (or multi-valued) change operations can be aggregated into a combined change operation for a more compact representation. Furthermore, redundant element-level change operations can be eliminated since they are now covered in an aggregated change operation. a-COG rules are recursive to incrementally aggregate elements for a particular change operation.

For instance, the a-COG rule for *merge* looks as follows:

$$(a_5) \quad c \in O_{new} \wedge A, B \subseteq O_{old} \wedge merge(A,c) \wedge merge(B,c) \wedge A \neq B \rightarrow \textbf{create}[merge(A \cup B, c)], \textbf{eliminate}[merge(A,c), merge(B,c)]$$

The rule specifies that two existing merge operations on concept sets A and B for the same target concept c can be combined into a merge on the union $A \cup B$ into c. Since the merge from A into c and from B into c are now covered by *merge*($A \cup B$, c) we eliminate the two previous ones. By iteratively applying the rule we can increasingly aggregate the input sets of the operations until no further aggregation is possible. In our example we would create *merge*({accessory cochlear nucleus, anterior cochlear nucleus}, cochlear ventral nucleus) and eliminate *merge*({accessory cochlear nucleus}, cochlear ventral nucleus) and *merge*({anterior cochlear nucleus}, cochlear ventral nucleus).

As described in Section 3.2, each complex change operation can be implemented by a series of basic change operations. When applying a rule we keep track of the simpler change operations and affected ontology components underlying a complex change operation. For instance, our merge operation can be implemented by the two basic *mapC* operations *mapC*(accessory cochlear nucleus, cochlear ventral nucleus) and *mapC*(anterior cochlear nucleus, cochlear ventral nucleus). Users can thus be provided with complex changes for an overview but also with detailed change information if needed.

5. Diff computation

In this section we present **COnto–Diff** to generate a diff evolution mapping. We first give an overview of the approach and discuss how to obtain the match mapping needed as input. In Section 5.3 we describe the algorithm to determine the basic evo-

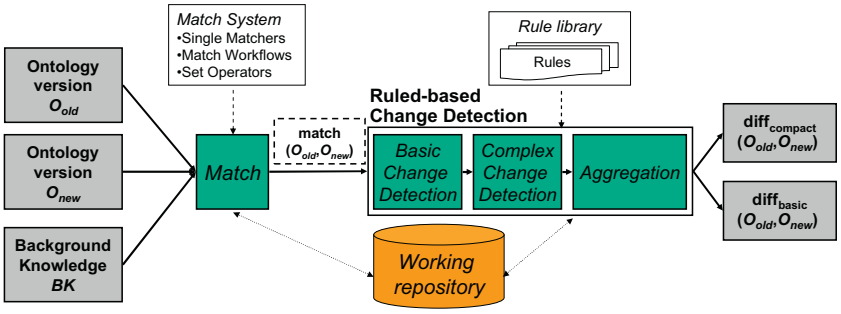

Fig. 2. Schematic overview of **COnto–Diff**.

lution mapping $diff_{basic}$ as well as the final evolution mapping $diff_{compact}$.

5.1. Overview

The main phases of **COnto–Diff** are illustrated in Fig. 2. The input are two versions of the same ontology (O_{old}, O_{new}). Optionally, background knowledge (e.g., dictionaries) can be provided for matching the ontologies. The result is an expressive diff evolution mapping $diff_{compact}(O_{old}, O_{new})$ and a corresponding $diff_{basic}(O_{old}, O_{new})$. Optionally we determine the inverse evolution mapping (Section 6.2). The complete algorithm operates on a working repository which stores the ontology versions as well as intermediate and final mappings.

The first phase is a *Matching* of the two ontology versions to identify common as well as modified but corresponding ontology elements. The result is a match mapping $match(O_{old}, O_{new})$ consisting of a set of *matchC* correspondences. The following steps use this result and are completely automatic. They utilize COG rules to determine the diff evolution mapping according to Algorithm 1. It sequentially applies the different kinds of COG rules for finding basic changes (*Basic Change Detection*), complex changes (*Complex Change Detection*), and aggregated complex changes (*Aggregation*).

5.2. Matching phase

The matching phase uses both input versions as well as optional background knowledge to compute a match mapping. Matching ontology versions is typically much easier than matching independent ontologies. This is due to the fact that a new version evolves from the older version and hence a larger part of the old version usually remains unchanged. To achieve a largely automatic ontology matching we use our GOMMA system that provides state-of-the-art match capabilities and has been successfully applied to life science ontologies [22]. The match strategy applied in this work is further explained in Section 7.1. As already mentioned, matching is generally semi-automatic, i.e., a domain expert should verify proposed match correspondences and correct them if necessary. Our Diff algorithm assumes that the match step provides the correct and complete set of correspondences.

The result of the matching phase for our motivating example in Fig. 1 is the following. All white categories exhibiting the same label in the old and new version do match as well as concept pairs connected by a dashed line.

5.3. Rule-based change detection

Algorithm 1 shows the overall procedure implementing the rule-based generation of diff evolution mappings. Its input are two ontology versions O_1 and O_2, a match mapping M between O_1 and O_2 as well as the list of COG rules R. The result contains two diff evolution mappings namely a basic diff evolution mapping $diff_{basic}(O_1, O_2)$ and a semantically enriched one, $diff_{compact}(O_1, O_2)$. The algorithm has three main steps to apply the three kinds of COG rules in order to generate the respective changes.

Algorithm 1. diffEvolMapGen

Input: two ontology versions O_1 and O_2, match mapping $M = match(O_1, O_2)$, list of rules $R = [R_{b-COG}, R_{c-COG}, R_{a-COG}]$,
Output: diff evolution mappings $diff_{basic}(O_1, O_2)$, $diff_{compact}(O_1, O_2)$

1 $diff_{basic}(O_1, O_2) \leftarrow$ `diffBasicGen`(O_1, O_2, M, R_{b-COG});
2 $D \leftarrow diff_{basic}(O_1, O_2)$;
3 **foreach** $r \in R_{c-COG}$ **do**
4 $D \leftarrow$ `applyRule` (D, r);
5 **end**
6 $diff_{compact}(O_1, O_2) \leftarrow$ `applyAggRules`(D, R_{a-COG});
7 **return** $[diff_{basic}(O_1, O_2), diff_{compact}(O_1, O_2)]$;

Step 1 calls a procedure *diffBasicGen* (see Algorithm 2) to generate the basic diff evolution mapping $diff_{basic}(O_1, O_2)$ based on match mapping M and the list of b-COG rules (R_{b-COG}). The b-COG rules need only to be applied once (*applyBasicRule*) in the pre-defined order (see numbering in Table A.1). We will use $diff_{basic}$ for ontology migration purposes (see Section 6.2).

Algorithm 2. diffBasicGen

Input: two ontology versions O_1 and O_2, match mapping $M = match(O_1, O_2)$, list of b-COG rules R_{b-COG}
Output: basic diff evolution mapping $D = diff_{basic}(O_1, O_2)$

1 $D \leftarrow empty$;
2 **foreach** $r \in R_{b-COG}$ **do**
3 $D \leftarrow$ `applyBasicRule` (D, r, O_1, O_2, M);
4 **end**
5 **return** D;

The processing of c-COG rules (steps 2–5 of Algorithm 1) is similar to the processing of b-COG rules in that each rule needs to be applied only once in the predetermined order. Rule processing starts with the basic diff evolution mapping and iteratively enriches the mapping with complex changes and the elimination of basic ones.

The *Aggregation* step requires to apply the a-COG rules multiple times to recursively aggregate set-valued change operations. This functionality is realized by the *applyAggRules* procedure (Algorithm 3) called in step 6 of Algorithm 1.

Algorithm 3. applyAggRules

Input: diff evolution mapping D, list of a-COG rules R_{a-COG}
Output: diff evolution mapping D'
1 $D' \leftarrow D$;
2 **repeat**
3 $D \leftarrow D'$;
4 **foreach** $r \in R_{a-COG}$ **do**
5 $D' \leftarrow$ `applyRule` (D', r);
6 **end**
7 **until** $D = D'$;
8 **return** D';

Algorithm 3 accepts an intermediate diff evolution mapping D and an ordered list of a-COG rules R_{a-COG} as input. In each iteration (repeat-until loop) the rules of R_{a-COG} are applied in their predefined order (see numbering in Table A.3). Thus, we can apply a-COG rules multiple times (once per iteration) to recursively detect and aggregate multiple change operations. The application of one rule (*applyRule*) modifies the temporary evolution mapping D' according to the rule's resulting actions (create, eliminate). We apply rules as long as new changes are inferred and the temporary mapping changes ($D \neq D'$).

Tables 2–4 contain the complete results of running the algorithm *diffEvolMapGen* for the running example. The first column shows the rule by which the change operation in column two was derived. Column three represents the rule by which a change operation was eliminated. Grey-shaded change operations had been created but later eliminated due to their coverage by a more complex change operation. The basic diff evolution mapping consists of the changes displayed in Table 2. All white-shaded changes in Tables 2–4 form the final (compact) diff evolution mapping which cannot further be compacted w.r.t. the used set of rules. As a result, we note that the basic

Table 2
Applying b-COG rules on motivating example.

Created by rule	*Created change operation*	*Eliminated by rule*
b_1	$addC$(trochlear nucleus)	c_6
b_1	$addC$(trigeminal nucleus)	
b_1	$addC$(brain stem white matter)	c_{12}
b_1	$addC$(cerebellar peduncle)	c_{12}
b_1	$addC$(crura cerebi)	c_6
b_1	$addC$(posterior commissure)	c_6
b_1	$addC$(superior cerebellar peduncle)	c_6
b_1	$addC$(middle cerebellar peduncle)	c_6
b_1	$addC$(inferior cerebellar peduncle)	c_6
b_3	$mapC$(accessory cochlear nucleus,cochlear ventral nucleus)	c_8
b_3	$mapC$(anterior cochlear nucleus,cochlear ventral nucleus)	c_8
b_6	$addR$(trochlear nucleus,is_a,brainstem nucleus)	c_6
b_6	$addR$(trigeminal nucleus,is_a,brainstem nucleus)	
b_6	$addR$(trigeminal motor nucleus,is_a,trigeminal nucleus)	c_2
b_6	$addR$(trigeminal sensory nucleus,is_a,trigeminal nucleus)	c_2
b_6	$addR$(brainstem white matter,part_of,brainstem)	
b_6	$addR$(cerebellar peduncle,is_a,brainstem white matter)	a_7
b_6	$addR$(crura cerebri,is_a,brainstem white matter)	c_6
b_6	$addR$(posterior commissure,is_a,brainstem white matter)	c_6
b_6	$addR$(superior cerebellar peduncle,is_a,cerebellar peduncle)	c_6
b_6	$addR$(middle cerebellar peduncle,is_a,cerebellar peduncle)	c_6
b_6	$addR$(inferior cerebellar peduncle,is_a,cerebellar peduncle)	c_6
b_7	$delR$(trigeminal motor nucleus,is_a,brainstem nucleus)	c_2
b_7	$delR$(trigeminal sensory nucleus,is_a,brainstem nucleus)	c_2
b_7	$delR$(accessory cochlear nucleus,is_a,brainstem nucleus)	
b_7	$delR$(anterior cochlear nucleus,is_a,brainstem nucleus)	

Table 3
Applying c-COG rules on motivating example.

Created by rule	*Created change operation*	*Eliminated by rule*
c_2	*move*(trigeminal motor nucleus,{brainstem nucleus},{trigeminal nucleus})	
c_2	*move*(trigeminal sensory nucleus,{brainstem nucleus},{trigeminal nucleus})	
c_6	*addLeaf*(trochlear nucleus,{brainstem nucleus})	
c_6	*addLeaf*(crura cerebri,{brainstem white matter})	c_{12}
c_6	*addLeaf*(posterior commissure,{brainstem white matter})	c_{12}
c_6	*addLeaf*(superior cerebellar peduncle,{cerebellar peduncle})	c_{12}
c_6	*addLeaf*(middle cerebellar peduncle,{cerebellar peduncle})	c_{12}
c_6	*addLeaf*(inferior cerebellar peduncle,{cerebellar peduncle})	c_{12}
c_8	*merge*({accessory cochlear nucleus},cochlear ventral nucleus)	a_5
c_8	*merge*({anterior cochlear nucleus},cochlear ventral nucleus)	a_5
c_{12}	*addSubGraph*(brainstem white matter,{crura cerebri})	a_{10}
c_{12}	*addSubGraph*(brainstem white matter,{posterior commissure})	a_{10}
c_{12}	*addSubGraph*(cerebellar peduncle,{superior cerebellar peduncle})	a_9
c_{12}	*addSubGraph*(cerebellar peduncle,{middle cerebellar peduncle})	a_9
c_{12}	*addSubGraph*(cerebellar peduncle,{inferior cerebellar peduncle})	a_9

Table 4
Applying a-COG rules on motivating example.

Created by rule	*Created change operation*	*Eliminated by rule*
a_5	*merge*({accessory cochlear nucleus,anterior cochlear nucleus}, cochlear ventral nucleus)	
a_9	*addSubGraph*(brain stem white matter,{cerebellar peduncle, superior cerebellar peduncle})	a_{10}
a_9	*addSubGraph*(brain stem white matter,{cerebellar peduncle, middle cerebellar peduncle})	a_{10}
a_9	*addSubGraph*(brain stem white matter,{cerebellar peduncle, inferior cerebellar peduncle})	a_{10}
a_{10}	*addSubGraph*(brain stem white matter,{crura cerebri,posterior commissure, cerebellar peduncle,inferior cerebellar peduncle, middle cerebellar peduncle,superior cerebellar peduncle})	

evolution mapping comprises 26 basic change operations while the semantically equivalent compact mapping contains merely 10 changes. Appendix B illustrates the generation of a complex change in more detail, namely the addition of the 'brainstem white matter' subgraph in the running example (right-hand side in Fig. 1).

For the correctness proofs of the proposed algorithm we refer to Appendix C.

6. Applications: migration of ontology versions and annotations

An important application of diff evolution mappings is the migration of ontology versions. In this Section we will first show how we can use the diff evolution mapping to migrate from an old to a new (changed) ontology version (Section 6.1). We then outline how we can use the inverse diff evolution mappings to migrate also in the backward direction (Section 6.2). Furthermore, we discuss how an evolution mapping can be used to adapt dependent data in particular annotations and ontology mappings (Section 6.3).

6.1. Basic version migration

We can migrate an old version O_1 to the changed version O_2 by applying the basic diff evolution mapping $diff_{basic}(O_1, O_2)$ on O_1. This approach results in an in-place ontology version migration that retains the unchanged ontology elements. Changes are thus limited to the ontology parts affected by the evolution supporting an efficient migration.

Algorithm 4 (*ontVersionMig*) implements the migration of ontology version O_1 to O_1' based on the basic diff evolution mapping $diff_{basic}(O_1, O_2)$.

Algorithm 4. ontVersionMig

Input: ontology version O_1, basic diff evolution mapping $D = diff_{basic}(O_1, O_2)$
Output: migrated ontology version O_1'
1 $O_1' \leftarrow O_1$;
2 *performOrder* ← [delA, delR, delC, mapC, mapA, mapR, addC, addA, addR];
3 **foreach** *chgOp* ∈ *performOrder* **do**
4 $O_1' \leftarrow \texttt{perform}(D.getChgOp(chgOp), O_1')$;
5 **end**
6 **return** O_1';

It executes the basic change operations of $diff_{basic}$ in a predefined order (*performOrder*). For the deletions, we first remove concept attributes. We then remove the concepts from the ontology structure and finally eliminate themselves. For the map changes we first need to substitute concepts (*mapC*), afterwards *mapR* and *mapA* can be executed, e.g., we update a changed attribute value or relationship type. Finally, for additions we first add the concept and then its attributes and relationships. We show the correctness of *ontVersionMig* in Appendix D.1.

6.2. Inverse diff mappings

Inverse diff evolution mappings can be applied to undo the changes in an evolution, i.e., we want to switch back from a changed ontology version to the old one. Our change model allows an easy way to determine an inverse diff evolution mapping because every change operation has an unique inverse change operation as introduced in Section 3 and listed in Table 1. Hence, we can simply replace every change operation by its inverse change operation to obtain the inverse mapping. The inverse of $diff_{basic}(O_1, O_2)$ gives us a basic evolution mapping that, using the algorithm *ontVersionMig*, can be used to correctly migrate from O_2 to O_1. We show the correctness of the inverse diff mapping in Appendix D.2.

To evaluate and verify the proposed algorithms we can apply a roundtrip migration between two ontology versions as illustrated in Fig. 3. We first migrate version O_1 to a changed version O_1' as follows: $O_1' = ontVersionMig(O_1, diff_{basic}(O_1, O_2))$. Due to the correctness of the migration algorithm, O_1' should equal O_2. Second, we determine the inverse of $diff_{basic}(O_1, O_2)$, i.e., $diff_{basic}(O_2, O_1)$, and use it to migrate O_1': $O_1'' = ontVersionMig(O_1', diff_{basic}(O_2, O_1))$. The resulting ontology O_1'' should equal O_1 due to the correctness of inverse mappings and their migration. We evaluate the roundtrip migration for real-world ontologies in Section 7.3.

6.3. Migration of annotations and dependent mappings

The generated evolution mappings are also useful to migrate annotations or ontology mappings affected by ontology changes. For instance, collaborating annotation curators can make use of **COnto–Diff** evolution mappings between their current ontology version and the newest release. Each curator would then be able to recognize whether or not changes occurred in his area of expertise and can (semi-) automatically migrate affected annotations. In general, additive changes such as *addSubGraph* or *addLeaf* may not affect existing annotations so that there is no immediate need to update annotations. However, other changes such as *merge*, *split*, *substitute* or *delSubGraph* require an update of affected annotations. The adaptation should be performed in a semi-automatic manner to limit the amount of manual effort but should still give curators the chance to make manual decisions. The necessary steps for adaptation depend on the type of change and include the following:

- *merge*, *leafMerge*: annotations of the source concepts of the old ontology version need to be migrated to the merged (target) concept of the new ontology version,
- *substitute*: affected annotations from the source concept need to be migrated (associated) to the new target concept,
- *delSubGraph*, *delLeaf*: an expert should be consulted to determine whether the annotations of the deleted concepts should also be deleted or whether they can be migrated to alternate concepts such as the parents of deleted concepts,
- *toObsolete*: an expert should be consulted to determine whether the annotations of the obsolete concepts should also be marked as obsolete or migrated to an alternate concept, e.g., parent concept,
- *split*, *leafSplit*: an expert should be consulted to determine to which new target concepts (selected from the ones listed in the change operation) one should migrate the annotations of a split source concept.

When annotations can be adapted in different ways the final decision can be reached in a collaborative way. For example, if there are multiple alternatives for an obsolete concept, a team of curators could vote which of the alternatives would be the best solution to migrate an annotation. We have already realized a first annotation migration functionality in our OnEX tool [16] covering *merge*, *toObsolete* and *delC* changes. In Section 7.3 we evaluate how annotations were influenced by changes in the Gene Ontology. In the future we plan to extend this work by incorporating more complex change operations for annotation migration. In a similar way, we plan to use evolution mappings to adapt ontology mappings (see Section 8).

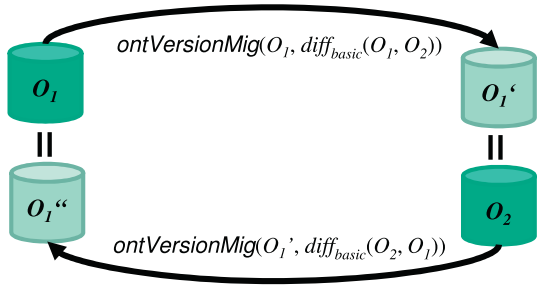

Fig. 3. Roundtrip migration of ontology versions using diff evolution mappings.

7. Evaluation experiments

We first describe the setup, in particular the tested ontologies and evolution scenarios. We then evaluate the basic and compact evolution mappings determined by **COnto–Diff**. We also analyze the use of the evolution mappings for a roundtrip migration of the ontologies and the adaptation of annotations. Finally, we compare the evolution mappings of **COnto–Diff** with those determined by PromptDiff.

7.1. Evaluation setup

We determine and evaluate evolution mappings for four large life science ontologies namely the Gene Ontology (GO) [12], the National Cancer Institute Thesaurus (NCIT) [39], the Mammalian Phenotype Ontology (MP) [42] and the Mouse Adult Gross Anatomy Ontology (MA) [19]. GO is widely used in bioinformatics for a uniform annotation of molecular-biological objects such as proteins or genes. NCIT is maintained at the National Cancer Institute and consists of 20 main categories which cover cancer-related topics such as drugs, tissues or anatomical structures. MP provides a set of standard terms for annotating mammalian phenotypic data

Table 5
Statistics for GO, NCIT, MP and MA scenarios.

$O_{old} - O_{new}$	$\lvert O_{old}\rvert(\lvert C\rvert, \lvert R\rvert)$	$\lvert O_{new}\rvert(\lvert C\rvert, \lvert R\rvert)$	$\lvert match\rvert$	$\lvert diff_{basic}\rvert$	$\lvert diff_{compact}\rvert$	*Ratio in %*
$GO_{2009-01} - GO_{2010-01}$	75,180 27,870 47,310	84,654 30,719 53,935	28,442	13,570	4304	31.7
$GO_{2010-01} - GO_{2011-01}$	84,654 30,719 53,935	93,366 33,271 60,095	31,151	13,691	4179	30.5
$NCIT_{2009-01} - NCIT_{2010-01}$	152,772 71,916 80,856	165,179 77,908 87,271	72,561	21,553	9360	43.4
$NCIT_{2010-01} - NCIT_{2011-01}$	165,179 77,908 87,271	184,818 87,396 97,422	79,275	25,065	8210	32.8
$MP_{2009-01} - MP_{2010-01}$	14,528 6807 7721	15,820 7430 8390	6971	1939	810	41.8
$MP_{2010-01} - MP_{2011-01}$	15,820 7430 8390	17,657 8230 9427	7576	2926	1072	36.6
$MA_{2009-01} - MA_{2010-01}$	6492 2873 3619	6611 2926 3685	2902	205	76	37.1
$MA_{2010-01} - MA_{2011-01}$	6611 2926 3685	6715 2968 3747	2945	168	62	36.9

while MA is a structured controlled vocabulary of the adult anatomy of the mouse. We consider two yearly evolution periods (three ontology versions) for each ontology: 2009 (2009–01 → 2010–01) and 2010 (2010–01 → 2011–01). The ontology versions are from the archives of GO,[4] NCIT [5] and OBO.[6] **COnto–Diff** is implemented in Java. We utilize a MySQL database as a working repository and for storing ontology versions as described in [23]. The experiments are conducted on a Windows XP desktop computer with an Intel Core 2 Duo CPU (2.66 GHz) and 4 GB of RAM.

Matching two ontology versions in the context of ontology evolution is facilitated by the large portion of unchanged concepts occuring in both versions. In the initial matching phase we thus apply the following automatic strategy. We utilize the fact that concepts of life science ontologies have unambiguous accession numbers and can thus easily identify most corresponding concepts in two ontology versions. Furthermore, we generate additional correspondences by checking if accessions or labels of concepts in the old version appear as synonyms of a concept in the new version. For instance, if the accession of a concept a appears as a synonym of concept b we would create the correspondence $matchC(a, b)$. We manually checked the match results especially correspondences between unequal concepts (correspondences like $matchC(a, a)$ are clear) and found that the automatically determined correspondences are correct and useful for our diff computation. In the comparison with PromptDiff (Section 7.4) we provide a specific match example that helped to find a merge change.

[4] GO: http://archive.geneontology.org/.
[5] NCIT: http://evs.nci.nih.gov/ftp1/NCI_Thesaurus/archive/.
[6] OBO: http://obo.cvs.sourceforge.net/viewvc/obo/obo/ontology/.

7.2. Basic vs. compact Diff evolution mappings

Table 5 lists details on the size of the considered ontology versions and the determined match and evolution mappings for the eight different evolution scenarios. For the ontology sizes we specify the number of concepts $|C|$ and relationships $|R|$. NCIT is the largest ontology followed by GO, MP and MA. The first three ontologies grew substantially in the two years (between 8% and 12% increase in each year) while MA only had a modest increase in size (2% per year).

The further columns show the sizes of $match$, $diff_{basic}$ and $diff_{compact}$, i.e., the number of match correspondences and the number of changes in the diff evolution mappings. For all investigated ontologies we could find a correspondence for each concept in the old ontology version, i.e., no ontology concept has been deleted but has at most been declared as obsolete. The number of match correspondences is even slightly higher than the number of concepts in the old versions since some source concepts are matched to several target concepts. The number of basic changes in $diff_{basic}$ is relatively high but much smaller than the number of correspondences underlining that the majority of concepts remained unchanged. For all scenarios the number of changes in $diff_{compact}$ could be substantially reduced with only between 30% and 43% of the number of changes in $diff_{basic}$ (see ratio in the last column of Table 5). The evolution mapping is also compact in comparison to the ontology sizes. For instance, $diff_{compact}(GO_{2010-01}, GO_{2011-01})$ is about a factor 20 smaller than the single versions $GO_{2010-01}$ and $GO_{2011-01}$. For MA the compactness of the diff is even higher (up to a factor of 100) due to a lower change intensity. These observations show that ontologies remain largely unchanged during evolution (many more

Table 6
Distribution of change operations. *add*, *del* and *map* represent all remaining basic changes for concepts (*mapC*, *addC*, *delC*) and relationships (*mapR*, *addR*, *delR*) in $diff_{compact}$.

	GO		NCIT		MP		MA	
	2009	2010	2009	2010	2009	2010	2009	2010
add	1375	1157	186	334	99	130	12	12
del	238	267	81	79	94	125	7	10
map	44	45	0	0	0	0	5	0
addLeaf	646	478	4,250	4,129	231	243	20	14
merge	89	113	28	28	6	17	4	0
leafSplit	239	182	161	318	65	58	7	6
substitute	1	3	1	0	6	41	0	0
move	1185	1526	4341	2387	177	305	18	18
toObsolete	63	44	67	189	56	31	0	0
addSubGraph	424	364	245	746	76	122	3	2
$\sum$	4304	4179	9360	8210	810	1072	76	62

match correspondences than basic changes). Furthermore, despite significant increases in ontology size our algorithm is able to find compact evolution mappings that we will now analyze in more detail.

To analyze the determined evolution mappings, we show in Table 6 the number of basic and complex change operations within the compact evolution mappings. We group all remaining basic changes into the three groups *add*, *del* and *map*. The large ontology growth is reflected in a high number of information extending operations such as *addLeaf*, *leafSplit* as well as subgraph additions. For instance, the largest subgraph (GO:0070887 – 'cellular response to chemical stimulus') added in 2009 encompasses 126 new concepts; in 2010 the largest added subgraph on 'renal system development' (GO:0072001) consisted of 168 concepts. There have been additions of large subgraphs in NCIT as well, e.g., 'MicroRNA_Gene' (C80699) with 285 concepts in 2009 or 'Gastrointestinal_System_Cancer_TNM_Finding_v7' (C89710) with 359 concepts in 2010. Furthermore, our algorithm was able to assign all *mapC* changes (no *mapC* changes in $diff_{compact}$) to a complex change such as *merge* or *substitute*. *mapR* changes were especially identified in GO where new relationship types such as 'negatively_regulates' or 'positively_regulates' have been introduced. Thus, existing relationships have been modified to include these new types and to enhance the semantics of GO.

Note that no concepts have been deleted in the ontologies under investigation (the delete frequencies in Table 6 are only *delR* changes and contain no *delC* changes). Outdated concepts are merely marked as obsolete but the outdated information is retained for compatibility reasons. Setting concepts to obsolete has frequently been performed in GO, NCIT and MP. For GO and MP we also observe a relative high number of *merge* changes. For example, in 2010 the four GO concepts GO:0001622 ('super conserved receptor expressed in brain receptor activity'), GO:0001623 ('Mas proto-oncogene receptor activity'), GO:0001624 ('RDC1 receptor activity') and GO:0001625 ('Epstein-Barr Virus-induced receptor activity') have been merged into GO:0004930 ('G-protein coupled receptor activity'). Interestingly, in NCIT some concepts have become obsolete and merged into another concept at the same time, e.g., 'Mammoplasty' (C51614) is obsolete since August 2009 but was also merged into 'Breast_Reconstruction' (C15354). Using our c-COG rule (c_{14}) we could identify such co-occurring modifications and prefer *merge* over *toObsolete* for the final diff result. Move changes, i.e., the rearrangement of concepts within the ontology have occurred for all investigated evolution scenarios.

Table 7
Roundtrip migration results for all investigated scenarios. The intersection ($O_1 \cap O_1''$) contains exactly the same elements as the union ($O_1 \cup O_1''$). For visualization, we show only set-sizes.

diff(O_1, O_2)	$\lvert O_1\rvert$	$\lvert O_1 \cap O_1''\rvert$	$\lvert O_1 \cap O_1''\rvert$
$GO_{2009-01} - GO_{2010-01}$	75,180	75,180	75,180
$GO_{2010-01} - GO_{2011-01}$	84,654	84,654	84,654
$NCIT_{2009-01} - NCIT_{2010-01}$	152,772	152,772	152,772
$NCIT_{2010-01} - NCIT_{2011-01}$	165,179	165,179	165,179
$MP_{2009-01} - MP_{2010-01}$	14,528	14,528	14,528
$MP_{2010-01} - MP_{2011-01}$	15,820	15,820	15,820
$MA_{2009-01} - MA_{2010-01}$	6492	6492	6492
$MA_{2010-01} - MA_{2011-01}$	6611	6611	6611

Table 8
Influence of ontology evolution on the Uniprot-GOA Human annotation set of version 63. The table shows the number of all and influencing changes occurred in GO between May 2008 and January 2011 as well as the number of affected annotations (changes with no impact are not displayed).

Change	Occurence	With impact	Affected annotations
merge	209	38	125
leaf Split	445	144	707
toObsolete	292	115	197
	946	297	1029

7.3. Ontology version and annotation migration

To evaluate the correctness of the determined evolution mappings we performed the roundtrip migration experiment described in Section 6.2. For comparing O_1 with O_1'' we utilize their representation as sets of concepts and relationships as introduced in Section 3.1. By testing $O_1 \cup O_1'' = O_1 \cap O_1''$ we evaluate whether the migrated ontology version O_1'' contains exactly the same elements (concepts, relationships) as O_1, indicating the completeness of the evolution mapping and its inverse mapping. The results are presented in Table 7. The first column shows the number of elements in the source version ($|O_1|$), the last two columns indicate the number of elements in the intersection and union of O_1 with O_1''. We can verify that O_1 and O_1'' contain exactly the same elements for all scenarios confirming that the determined evolution mappings allow for the correct migration of ontologies and that **COnto–Diff** has been correctly implemented.

To analyze the migration of annotations we use the manually-curated Uniprot-GOA Human annotations in version 63 from May 2008.[7] The annotations are based on the GO version from May 2008 and we compute the evolution mapping $diff_{compact}(GO_{2008-05}, GO_{2011-01})$ to study changes w.r.t. the GO version of January 2011. The results in Table 8 show the change frequency for three complex changes and how many annotations were affected by these changes. Overall we observe that more than 1,000 annotations are affected and that about every third change operation influenced at least one annotation. For example, we determined 38 merge operations affecting 125 annotations. We could use this mapping to automatically update the annotations with the new ontology concepts.

7.4. Comparison with PromptDiff

We now compare the results obtained with **COnto–Diff** with those determined by PromptDiff [32]. For this analysis, we evaluate the MA diff evolution mapping for 2009. We loaded the two versions $MA_{2009-01}/MA_{2010-01}$ into the current version of Protégé and executed PromptDiff to get the difference table. PromptDiff uses a somewhat different set of change operations, in particular they

[7] Uniprot-GOA archive: ftp://ftp.ebi.ac.uk/pub/databases/GO/goa/old/HUMAN/.

Table 9
Quantitative statistics of the COnto–Diff–PromptDiff comparison for the MA 2009 scenario. The table shows the number of detected changes (occ.) as well as information about their manually verified correctness (corr.). Ticks denote correct results while crosses signal wrong change operations.

COnto–Diff			PromptDiff		
Change	occ.	corr.	corr.	occ.	Change
Addition of concepts					
addC	5	✓	✓	59	add
addleaf	20	✓			
addSubgraph	3	✓			
leafSplit	7	✓			
Structural changes					
addR	7	✓	✓	15	add
delR	7	✓	✓	15	delete
mapR	5	✓			
move	18	✓	✓	15	change
Revision of concepts					
merge	4	✓	✗	2	delete
			✗	2	map
Overall	76			108	

distinguish between *add*, *delete*, *merge*, *split*, *map* and *change* modifications. For a better comparison, we group our changes and those of PromptDiff into three categories namely 'Addition of concepts', 'Structural changes' and 'Revision of concepts' as shown in Table 9.

PromptDiff generates a total of 108 changes, while **COnto–Diff** determines only 76 complex change operations. Many changes are similar in the two diff results but there are also substantial differences caused by the different sets of changes and different detection approaches. While PromptDiff has found 59 concept additions, **COnto–Diff** classifies and aggregates those into more expressive changes like *leafSplit* or *addSubGraph*. For instance, the addition of a new subgraph MA:0002901 ('aorta wall') representing knowledge about the wall of the aorta and its different layers was identified. Another example is the split of MA:0001722 ('decidua') into the two more fine-grained leaf concepts MA:0002905 ('decidua basalis') and MA:0002906 ('decidua capsularis'). These examples show why there are fewer change operations in the **COnto–Diff** result and that these changes are more precise and understandable than just listing individual additions.

With respect to 'Structural changes', both tools identify the same concepts involved. However, PromptDiff sometimes does not aggregate basic changes into more complex ones, e.g., it often reports one *add* and one *delete* change instead of a *move*. Particularly, **COnto–Diff** also detects the 15 relationship changes found by PromptDiff either as a *mapR* or as a *move* change. **COnto–Diff** detects eight additional *mapR*/ *move* changes which are classified as relationship additions and deletions by PromptDiff. Thus, the diff of PromptDiff contains 16 more structural changes (8 additions, 8 deletions) resulting in a less compact diff representation.

The most significant differences occur for changes of the last category 'Revision of concepts'. **COnto–Diff** determines four correct merges, including a merge of the apparently redundant concept MA:0002440 ('smooth muscle tissue') into the retained concept MA:0000166 ('smooth muscle tissue') as well as the merge of MA:0002832 ('ventricle trabecula carnea') into MA:0002831 ('trabecula carnea'). PromptDiff does not recognize any merge of concepts but detects two concept maps and two deletions. In particular, PromptDiff maps MA:0002440 to MA:0002930 ('posteromedial cortical amygdaloid nucleus') and MA:0002832 to MA:0002922 ('basomedial amygdaloid nucleus') which are apparently both wrong. PromptDiff also recognizes two wrong deletions of MA:0000056 and MA:0001485. By contrast, **COnto–Diff** correctly finds that these concepts are involved in merge operations, namely MA:0000056 ('fat') is merged into MA:0000009 ('adipose tissue') and MA:0001485 ('incisive bone') into MA:0001493 ('premaxilla'). This is made possible by our match strategy using accession numbers and synonyms of ontology concepts. For example, the concept MA:0000009 in the new version contains a synonym MA:0000056 so that the match step generates a correspondence *matchC*(MA:0000056, MA:0000009) which together with the trivial *matchC*(MA:0000009, MA:000009) is later rewritten into a merge. We suspect that the PromptDiff problems are due to its match approach. It seems that PromptDiff first applies an exact matching of concept accessions/labels and then tries to perform a fuzzy lexical matching of unmatched concepts. It therefore does not create correspondences to already matched concepts leading to wrong correspondences.

In summary, we observe that **COnto–Diff** can determine a more expressive and more compact diff result than PromptDiff and that PromptDiff determines wrong changes, probably because of limitations in the applied matching.

8. Discussion and future work

We presented a new rule-based approach **COnto–Diff** to determine an expressive and invertible diff evolution mapping between two versions of the same ontology. The diff evolution mapping covers basic and complex changes. The approach is based on an initial matching between the ontology versions and utilizes Change Operation Generating Rules (COG rules) to find the basic as well as complex change operations. The rules also specify which simpler changes are replaced by more expressive changes. The evaluation on large life science ontologies showed that our Diff approach generates semantically expressive and compact evolution mappings. We also showed that the determined evolution mappings allow the correct migration of old into new ontology version or vice versa. They can also be used to semi-automatically adapt annotations and ontology mappings after ontology modifications. We expect the expressive evolution mappings to be useful for ontology developers and users such as curators, especially for large ontologies.

A key advantage of the proposed **COnto–Diff** approach over previous solutions such as PromptDiff is its high modularity and flexibility. First, **COnto–Diff** separates matching from diff computation and can thus leverage customized match strategies, e.g., to deal with domain-specific ontology characteristics such as the use of synonyms in life science ontologies. Using this strategy we are able to automatically compute diffs for all OBO-based ontologies available in the OBO Foundry or in BioPortal. Second, the rule-based approach supports an easy extension of **COnto–Diff** to identify additional kinds of changes. We could thus incorporate

specific changes to deal with obsolete concepts. The comparative evaluation with PromptDiff (see Section 7.4) showed that **COnto–Diff** can determine more compact and more correct diff evolution mappings due to its use of tailored matching techniques and the ability to correctly identify complex changes such as merges. We made the proposed diff functionality already available within a web tool [15]. Thus, users can compute and analyze evolution mappings for different life science ontologies online or via a web service interface.

There are still some limitations of **COnto–Diff** and its usability for further kinds of ontologies. First, the matching step which we could apply automatically for life science ontologies in this paper may require human intervention, e.g., for correcting/ revising determined correspondences. Especially if we cannot make use of unambiguous concept accession numbers, we need to apply further matching techniques to find correspondences. Second, to support further changes corresponding COG rules must be defined correctly, i.e., domain knowledge about the evolution of ontologies needs to be formally defined in rules to detect the changes. A designer of such rules needs to keep possible effects in mind, i.e., one must be aware of the consequences by introducing defective rules or manipulating their order. Third, **COnto–Diff** does not yet cover OWL constructs such as disjunctive classes or property restrictions. Fourth, **COnto–Diff** is currently not integrated in an ontology editor such Protégé, however it would be possible to provide **COnto–Diff** as a plugin so that it can be used in editors as well.

In future work, we plan to address these issues as well as the following topics. We want to apply **COnto–Diff** to additional ontologies and domains. We also want to investigate in more detail the adaptation of ontology mappings and of ontology instances.

Acknowledgments

We thank all anonymous reviewers for their useful comments to improve the paper. This work is supported by the German Research Foundation (DFG), Grant RA 497/18-1 (“Evolution of Ontologies and Mappings”).

Appendix A. Summary of COG rules

The following Tables A.1–A.3 list all COG rules (b-COG, c-COG, a-COG) that allow the determination of all change operations introduced in Section 3.2.

Appendix B. Example walkthrough for Diff algorithm

In the following we illustrate an example walkthrough to determine the subgraph addition in our running example (see Fig. 1). For this subgraph example the b-COG addition rule (b_1) would detect seven concept additions: *addC*(brainstem white matter), *addC*(crura cerebri), *addC*(cerebellar peduncle), *addC*(superior cerebellar peduncle), *addC*(middle cerebellar peduncle), *addC*(inferior cerebellar peduncle) and *addC*(posterior commissure). For detecting subgraph additions the following c-COG rules are applied:

$$(c_6)\quad a,r \in O_{new} \wedge addC(a) \wedge addR(r) \wedge a = r_{source} \wedge \nexists s(s \in O_{new} \wedge addR(s) \wedge a = s_{target}) \wedge r_{type}, s_{type} \in \{'is\_a', 'part\_of'\} \rightarrow \textbf{create}[addLeaf(a, \{r_{target}\})], \textbf{eliminate}[addC(a), addR(r)]$$

$$(c_{10})\quad a,b \in O_{new} \wedge B \subseteq O_{new} \wedge addC(a) \wedge addLeaf(b,B) \wedge a \in B \rightarrow \textbf{create}[addSubGraph(a,\{b\})], \textbf{eliminate}[addC(a), addLeaf(b,B)]$$

The first rule (c_6) is used to detect leaf concept additions. Rule (c_{10}) is based on the results of (c_6) and infers subgraph additions connecting a newly added concept a and a leaf concept b rooted at a. In our example ‘crura cerebri’, ‘superior cerebellar peduncle’, ‘middle cerebellar peduncle’, ‘inferior cerebellar peduncle’ and ‘posterior commissure’ are classified as leaf concept additions. Afterwards rule (c_{10}) infers *addSubGraph*(brainstem white matter, {crura cerebri}), *addSubGraph*(brainstem white matter, {posterior commissure}), *addSubGraph*(cerebellar peduncle, {superior cerebellar peduncle}), *addSubGraph*(cerebellar peduncle, {middle cerebellar peduncle}) and *addSubGraph*(cerebellar peduncle, {inferior cerebellar peduncle}). We then can apply the following a-COG rules:

$$(a_9)\quad a,b,r \in O_{new} \wedge A \subseteq O_{new} \wedge addSubGraph(a,A) \wedge addC(b) \wedge addR(r) \wedge r_{source} = a \wedge r_{target} = b \wedge r_{type} \in \{'is\_a', 'part\_of'\} \rightarrow \textbf{create}[addSubGraph(b, \{a\} \cup A)], \textbf{eliminate}[addSubGraph(a,A), addC(b), addR(r)]$$

$$(a_{10})\quad a \in O_{new} \wedge A,B \subseteq O_{new} \wedge addSubGraph(a,A) \wedge addSubGraph(a,B) \wedge A \neq B \rightarrow \textbf{create}[addSubGraph(a, A \cup B)], \textbf{eliminate}[addSubGraph(a,A), addSubGraph(a,B)]$$

Particularly, (a_9) recursively aggregates added concepts into larger subgraphs. If multiple subgraph additions with the same root exist, we can aggregate these into one by fusing their sub concepts (a_{10}). In our example (a_9) would detect three changes: *addSubGraph*(brainstem white matter,{cerebellar peduncle, superior cerebellar peduncle}), *addSubGraph*(brainstem white matter,{cerebellar peduncle, middle cerebellar peduncle}) and *addSubGraph*(brainstem white matter,{cerebellar peduncle, inferior cerebellar peduncle}) which are finally aggregated into *addSubGraph*(brainstem white matter, {crura cerebri, posterior commissure, cerebellar peduncle, superior cerebellar peduncle, middle cerebellar peduncle, inferior cerebellar peduncle}) by (a_{10}).

Appendix C. Correctness of Diff algorithm

We will show that the proposed algorithm for generating diff evolution mappings (see Section 5.3) is correct, in particular that it generates all changes and that it terminates. We first show that the generation of the basic diff evolution mapping is complete, i.e., determines all basic changes between two input ontology versions O_{old} and O_{new}. We focus on concept changes; the correctness proof for relationship and attribute changes is analogous.

Theorem 1. *The b-COG rules applied in diffBasicGen generate a complete basic diff evolution mapping* $diff_{basic}(O_{old}, O_{new})$ *containing*

(a) all concept additions (*addC*) between O_{old} and O_{new},
(b) all concept deletions (*delC*) between O_{old} and O_{new},
(c) all concept changes (*mapC*) including concepts that map to multiple concepts in the other ontology version.

To prove the theorem, we refer to the five b-COG rules (b_1–b_5) introduced in Section 4.1 and applied in *diffBasicGen*. The rules distinguish between concepts that match with at least one concept in the other ontology version and those that do not match. For all non-matching concepts of O_{new} b-COG rule (b_1) generates *addC* change operations. b-COG rule (b_2) generates concept deletions (*delC*) for all non-matching concepts of O_{old}. Matching concepts occur in correspondences $matchC(a, b) \in match(O_{old}, O_{new})$ and are processed by b-COG rules (b_3), (b_4) and (b_5). Rule (b_3) creates a $mapC(a, b)$ change if a and b are unequal ($a \neq b$). For ($a = b$), rules (b_4) and (b_5) ensure that we only create a *mapC* change if the concept is involved in further correspondences, i.e., has not remained

Table A.1
List of all b-COG rules.

ID	Rule
b_1	Creation of a concept addition: $c \in O_{new} \wedge \nexists a(a \in O_{old} \wedge matchC(a, c))$ $\rightarrow$**create**[$addC(c)$]
b_2	Creation of a concept addition: $c \in O_{old} \wedge \nexists a(a \in O_{new} \wedge matchC(c, a))$ $\rightarrow$**create**[$delC(c)$]
b_3	Creation of a concept map between different concepts based on a match correspondence: $a \in O_{old} \wedge b \in O_{new} \wedge matchC(a, b) \wedge a \neq b$ $\wedge\neg isObsolete(a) \wedge \neg isObsolete(b)$ $\rightarrow$**create**[$mapC(a, b)$]
b_4	Creation of a concept map between equal concepts if source is involved in multiple match correspondences: $a \in O_{old}, O_{new} \wedge matchC(a, a) \wedge \exists b(b \in O_{new} \wedge matchC(a, b) \wedge a \neq b)$ $\wedge\neg isObsolete(a) \wedge \neg isObsolete(b)$ $\rightarrow$**create**[$mapC(a, a)$]
b_5	Creation of a concept map between equal concepts if source is involved in multiple match correspondences: $a \in O_{old}, O_{new} \wedge matchC(a, a) \wedge \exists b(b \in O_{new} \wedge matchC(b, a) \wedge a \neq b)$ $\wedge\neg isObsolete(a) \wedge \neg isObsolete(b)$ $\rightarrow$**create**[$mapC(a, a)$]
b_6	Creation of a relationship addition: $r \in O_{new} \wedge r \notin O_{old}$ $\rightarrow$**create**[$addR(r)$]
b_7	Creation of a relationship deletion: $r \in O_{old} \wedge r \notin O_{new}$ $\rightarrow$**create**[$delR(r)$]
b_8	Creation of a relationship map if only the relationship type between two concepts changed: $r \in O_{old} \wedge s \in O_{new} \wedge delR(r) \wedge addR(s) \wedge r_{source} = s_{source} \wedge$ $r_{target} = s_{target} \wedge r_{type} \neq s_{type}$ $\rightarrow$**create**[$mapR(r, s)$],**eliminate**[$delR(r)$,$addR(s)$]
b_9	Creation of an attribute addition: $p \in O_{new} \wedge p \notin O_{old}$ $\rightarrow$**create**[$addA(p)$]
b_{10}	Creation of an attribute deletion: $p \in O_{old} \wedge p \notin O_{new}$ $\rightarrow$**create**[$delA(p)$]
b_{11}	Creation of an attribute map if value has changed: $p \in O_{old} \wedge q \in O_{new} \wedge delA(p) \wedge addA(q) \wedge p_{concept} = q_{concept} \wedge$ $p_{name} = q_{name} \wedge p_{value} \neq q_{value}$ $\rightarrow$**create**[$mapA(p, q)$],**eliminate**[$delA(p)$,$addA(q)$]

the same. Hence, all $matchC(a, a)$ connecting unchanged concepts are not included in $diff_{basic}$. In summary, $diff_{basic}$ reflects all basic changes but does not relate unchanged ontology parts.

Theorem 2. *The c-COG and a-COG rules applied in the second part of diffEvolMapGen terminate and generate a complete and the most compact diff evolution mapping $diff_{compact}(O_{old}, O_{new})$ w.r.t. our defined change operation set and rules.*

We first show that algorithm *diffEvolMapGen* terminates. This is mainly ensured by two facts. First, all rules operate on a finite number of ontology elements in O_{old}/O_{new} and do not create new ontology elements. Second, the evaluation of all rules terminates since we apply them only once or as long as the mapping changes. The application of c-COG rules terminates since they are non-recursive and are applied only once based on a pre-defined order. a-COG rules are recursive but always reduce the number of change operations by aggregating ontology elements. Particularly, each rule uses at least two input change operations which are fused into one; the input change operations are eliminated. This steady reduction of change operations terminates when the most aggregated change operations have been found.

To prove the completeness of the generated $diff_{compact}$ we have to show that this mapping covers all changes. This is ensured by the completeness of the input ($diff_{basic}$) and since our rules cover all possible changes and the rule execution always terminates.

The derivation of the most compact evolution mapping is ensured by the design of our rules. In particular, we can show for every change operation that it will be rewritten by the algorithm to at most one complex change operation in a unique way. This is because the preconditions using a particular kind of change operation in different rules are mutually exclusive. Together with the termination and completeness properties this confluence behavior ensures that we determine the most compact evolution mapping. We will illustrate the confluence only for the non-trivial *mapC* change operation, the proof for the other basic change operations is analogous. For the map of a concept a into another concept b($mapC(a, b)$) four variants may occur:

1. $mapC(a, b)$ belongs to a substitute, i.e., a is exactly replaced by b
2. $mapC(a, b)$ belongs to a merge of multiple concepts A into b
3. $mapC(a, b)$ belongs to a split of a into multiple concepts B
4. $mapC(a, b)$ cannot be rewritten

From the c-COG rules only rules (c_1), (c_8) and (c_9) can consume *mapC* changes. The preconditions in the three rules are mutually exclusive, i.e., exactly one of them or no rule is applicable. If *mapC* connects two concepts a and b which are not involved in any other

Table A.2
List of all c-COG rules.

ID	Rule
c_1	Creation of a substitute if exactly two different concepts map: $a \in O_{old} \wedge b \in O_{new} \wedge mapC(a, b) \wedge a \neq b \wedge$ $\nexists c(c \in O_{new} \wedge mapC(a, c) \wedge b \neq c) \wedge \nexists d(d \in O_{old} \wedge mapC(d, b) \wedge a \neq d)$ →**create**[*substitute*(*a*, *b*)],**eliminate**[*mapC*(*a*, *b*)]
c_2	Creation of a move if a concept's is_a/part of relationship changed: $r \in O_{old} \wedge s \in O_{new} \wedge delR(r) \wedge addR(s) \wedge r_{source} = s_{source} \wedge r_{target} \neq s_{target}$ $\wedge r_{type} = s_{type} \wedge r_{type}, s_{type} \in \{'is\_a', 'part\_of'\}$ →**create**[*move*(r_{source},{r_{target}},{s_{target}})], **eliminate**[*delR*(*r*),*addR*(*s*)]
c_3	Creation of an attribute value change: $p \in O_{old} \wedge q \in O_{new} \wedge mapA(p, q)$ →**create**[*chgAttValue*($p_{concept}$, p_{name},{p_{value}},{q_{value}})], **elimitnate**[*mapA*(*p*, *q*)]
c_4	Creation of toObsolete if a concept's status changed from false to true: $c \in O_{new} \wedge chgAttValue(c, n, V, W) \wedge n =$ 'obsolete' $\wedge$ 'false' $\in V \wedge$ 'true' $\in W$ →**create**[*toObsolete*(*c*)],**eliminate**[*chgAttValue*(*c*, *n*, *V*, *W*)]
c_5	Creation of revokeObsolete if a concept's status changed from true to false: $c \in O_{new} \wedge chgAttValue(c, n, V, W) \wedge n =$ 'obsolete' $\wedge$ 'true' $\in V \wedge$ 'false' $\in W$ →**create**[*revokeObsolete*(*c*)],**eliminate**[*chgAttValue*(*c*, *n*, *V*, *W*)]
c_6	Creation of a leaf addition if a concept with no children was added: $a, r \in O_{new} \wedge addC(a) \wedge addR(r) \wedge a = r_{source} \wedge r_{type} \in \{'is\_a', 'part\_of'\} \wedge$ $\nexists s(s \in O_{new} \wedge addR(s) \wedge a = s_{target} \wedge s_{type} \in \{'is\_a', 'part\_of'\})$ →**create**[*addLeaf*(*a*,{r_{target}})],**eliminate**[*addC*(*a*),*addR*(*r*)]
c_7	Creation of a leaf deletion if a concept with no children was deleted: $a, r \in O_{old} \wedge delC(a) \wedge delR(r) \wedge a = r_{source} \wedge r_{type} \in \{'is\_a', 'part\_of'\} \wedge$ $\nexists s(s \in O_{old} \wedge delR(s) \wedge a = s_{target} \wedge s_{type} \in \{'is\_a', 'part\_of'\})$ →**create**[*delLeaf*(*a*,{r_{target}})],**eliminate**[*delC*(*a*),*delR*(*r*)]
c_8	Creation of a concept merge if multiple maps to one target exist: $a, b, \in O_{old} \wedge c \in O_{new} \wedge mapC(a, c) \wedge mapC(b, c) \wedge a \neq b \wedge$ $\nexists d(d \in O_{new} \wedge mapC(a, d) \wedge c \neq d) \wedge \nexists e(e \in O_{new} \wedge mapC(b, e) \wedge c \neq e)$ →**create**[*merge*({*a*}, *c*)],**eliminate**[*mapC*(*a*, *c*),*mapC*(*b*, *c*)]
c_9	Creation of a merge from a leaf concept into its parent: $a, b, \in O_{old} \wedge A \subseteq O_{old} \wedge c \in O_{new} \wedge mapC(b, c) \wedge delLeaf(a, A) \wedge b \neq A \wedge$ $\nexists r(r \in O_{new} \wedge r_{target} = c \wedge r_{type} \in \{'is\_a', 'part\_of'\})$ →**create**[*leafMerge*({*a*}, *c*)],**eliminate**[*mapC*(*b*, *c*),*delLeaf*(*a*, *A*)]
c_{10}	Creation of a concept split if multiple maps from one source exist: $c \in O_{old} \wedge a, b \in O_{new} \wedge mapC(c, a) \wedge mapC(c, b) \wedge a \neq b \wedge$ $\nexists d(d \in O_{old} \wedge mapC(d, a) \wedge c \neq d) \wedge \nexists e(e \in O_{old} \wedge mapC(e, b) \wedge c \neq e)$ →**create**[*split*(*c*,{*a*}),*split*(*c*,{*b*})],**eliminate**[*mapC*(*c*, *a*),*mapC*(*c*, *b*)]
c_{11}	Creation of a split from a leaf concept into its children: $c \in O_{old} \wedge a, b \in O_{new} \wedge A \subseteq O_{new} \wedge mapC(c, b) \wedge addLeaf(a, A) \wedge b \in A \wedge$ $\nexists r(r \in O_{old} \wedge r_{target} = c \wedge r_{type} \in \{'is\_a', 'part\_of'\})$ →**create**[*leafSplit*(*c*,{*a*})],**eliminate**[*mapC*(*c*,*b*),*mapC*(*a*, *A*)]
c_{12}	Creation of a subgraph addition based on a leaf and inner concept addition: $a, b \in O_{new} \wedge B \subseteq O_{new} \wedge addC(a) \wedge addLeaf(b, B) \wedge a \in B$ →**create**[*addSubGraph*(*a*,{*b*})],**eliminate**[*addC*(*a*),*addLeaf*(*b*, *B*)]
c_{13}	Creation of a subgraph deletion based on a leaf and inner concept deletion: $a, b \in O_{old} \wedge B \subseteq O_{old} \wedge delC(a) \wedge delLeaf(b, B) \wedge a \in B$ →**create**[*delSubGraph*(*a*,{*b*})],**eliminate**[*delC*(*a*),*delLeaf*(*b*, *B*)]
c_{14}	Prioritization of merges over toObsolete changes if one and the same concept is affected: $a \in O_{old} \wedge A \subseteq O_{old} \wedge b \in O_{new} \wedge merge(A, b) \wedge toObsolete(a) \wedge a \in A$ →**eliminate**[*toObsolete*(*a*)]

mapC change, only rule (c_1) can be executed since (c_8) and (c_9) require a second *mapC* change operation involving either *a* or *b*. In this case *mapC*(*a*, *b*) is rewritten to a substitute (*substitute*(*a*, *b*)) which cannot be aggregated (no a-COG rule). If the target concept *b* of *mapC*(*a*, *b*) is involved in another *mapC* change *mapC*(*c*, *b*) it is possible that rule (c_8) can be executed. However, the precondition ensures that *a* as well as *c* must not be involved in another *mapC* change, e.g., *mapC*(*a*, *d*) or *mapC*(*b*, *d*). If this applies, both changes belong to a merge into the common target *b*. (c_8) would generate two partial merges, one from *a* into *b* and one from *c* into *b*. In the aggregation step (rule (a_5) for merge), these partial merges are unified in one common *merge* change operation, i.e., all *mapC* which belong to a merge are rewritten to one common *merge* operation. On the opposite, the source concept may be involved in multiple *mapC* changes, i.e., another *mapC* change *mapC*(*a*, *c*) exists. In this case, the split rule (c_9) could be executed if the targets *b* and *c* are not involved in any other *mapC* change operation. If the precondition is fulfilled we would derive two partial splits, one from *a* to *b* and one from *a* to *c*. In the aggregation phase, rule (a_6) then unifies all partial splits with the same source concept (in our case *a*). Hence, all *mapC* changes which belong to a common split change are rewritten to exactly one *split* change operation. The fourth variant occurs in situations where no exact decision can be reached. For instance, if *mapC*(*a*, *b*) exists, *a* is mapped to another target concept *c*(*mapC*(*a*, *c*)) and *b* is mapped to another source concept *d*(*mapC*(*d*, *b*)) we cannot clearly find out what happened. On the one hand, *a* and *d* could be merged into *b*. On the other hand, *a* could be split into *b* and *c*. Since we cannot automatically treat this case, we do not derive any complex change operation so that we keep the basic change operation.

The properties of our rule set guaranteeing termination and confluence need to be preserved when extending it for the support of additional change operations. In particular, new COG rules should not introduce cyclic dependencies between rules and only aggregation rules reducing the number of changes may be recursive. Furthermore, the use of a new change operation in multiple rules must use mutually exclusive preconditions for this change operation.

Appendix D. Correctness of version migration algorithms

D.1. Correctness of basic version migration

We will show the correctness of the *ontVersionMig* algorithm presented in Section 6.1.

Theorem 3. *Algorithm ontVersionMig is correct, i.e., for the basic diff evolution mapping* $diff_{basic}(O_1, O_2)$ *determined by algorithm diffBasicGen, it creates the new ontology version* O_2 *from the original version* O_1.

We prove the theorem for concept changes. We need to show that the generated ontology version is complete, i.e., it (1) contains all concepts and (2) contains no further/other concepts. First, it is easy to see that the algorithm removes deleted concepts indicated by *delC* changes so that they do not become part of O_2. Analogously, the concepts of the domain of *mapC* changes are eliminated and thus do not appear in O_2. Second, an unchanged concept *c* already available in O_1 should be present in O_2 as well. Particularly, such a *c* is not covered by any change operation of $diff_{basic}$ and thus remains in O_2. Finally, concepts specified by *addC* changes are added to the unchanged ontology part and thus become part of O_2. In the same way the range concepts of *mapC* changes in $diff_{basic}$ are inserted.

D.2. Correctness of inverse diff mappings

In the following we will prove the correctness of our inverse diff mappings.

Theorem 4. *The inverse of a basic diff evolution mapping* $diff_{basic}(O_1, O_2)$ *is correct, i.e., is identical to* $diff_{basic}(O_2, O_1)$.

We prove this theorem for concept changes. We will show that the inverse of every change operation in $diff_{basic}(O_1, O_2)$ is in $diff_{basic}(O_2, O_1)$ and also that $diff_{basic}(O_2, O_1)$ does not contain additional changes. The addition of a concept *c*(*addC*(*c*)) in $diff_{basic}(O_1, O_2)$ has *delC*(*c*) as inverse. Since *c* is not in O_1 but in O_2

Table A.3
List of all a-COG rules.

ID	Rule
a_1	Aggregation of move changes if they refer to the same concept: $c \in O_{new} \wedge A, C \subseteq O_{old} \wedge B, D \subseteq O_{new} \wedge move(c, A, B) \wedge move(c, C, D) \wedge (A \neq C \vee B \neq D)$ $\rightarrow$**create**$[move(c, A \cup C, B \cup D)]$,**eliminate**$[move(c, A, B), move(c, C, D)]$
a_2	Aggregation of chgAttValue changes if multiple values of an attribute changed: $c \in O_{new} \wedge chgAttValue(c, n, A, B) \wedge chgAttValue(c, n, C, D) \wedge (A \neq C \vee B \neq D)$ $\rightarrow$**create**$[chgAttValue(c, n, A \cup C, B \cup D)]$,**eliminate**$[chgAttValue(c, n, A, B), chgAttValue(c, n, C, D)]$
a_3	Aggregation of addLeaf changes if they refer to the same concept: $a \in O_{new} \wedge A, B \subseteq O_{new} \wedge addLeaf(a, A) \wedge addLeaf(a, B) \wedge A \neq B$ $\rightarrow$**create**$[addLeaf(a, A \cup B)]$,**eliminate**$[addLeaf(a, A), addLeaf(a, B)]$
a_4	Aggregation of delLeaf changes if they refer to the same concept: $a \in O_{old} \wedge A, B \subseteq O_{old} \wedge delLeaf(a, A) \wedge delLeaf(a, B) \wedge A \neq B$ $\rightarrow$**create**$[delLeaf(a, A \cup B)]$,**eliminate**$[delLeaf(a, A), delLeaf(a, B)]$
a_5	Aggregation of merges if they share the same target concept: $c \in O_{new} \wedge A, B \subseteq O_{old} \wedge merge(A, c) \wedge merge(B, c) \wedge A \neq B$ $\rightarrow$**create**$[merge(A \cup B, c)]$,**eliminate**$[merge(A, c), merge(B, c)]$
a_6	Aggregation of leaf merges if they share the same target concept: $c \in O_{new} \wedge A, B \subseteq O_{old} \wedge leafMerge(A, c) \wedge leafMerge(B, c) \wedge A \neq B$ $\rightarrow$**create**$[leafMerge(A \cup B, c)]$,**eliminate**$[leafMerge(A, c), leafMerge(B, c)]$
a_7	Aggregation of splits if they share the same source concept: $c \in O_{old} \wedge A, B \subseteq O_{new} \wedge split(c, A) \wedge split(c, B) \wedge A \neq B$ $\rightarrow$**create**$[split(c, A \cup B)]$,**eliminate**$[split(c, A), split(c, B)]$
a_8	Aggregation of leaf splits if they share the same source concept: $c \in O_{old} \wedge A, B \subseteq O_{new} \wedge leafSplit(c, A) \wedge leafSplit(c, B) \wedge A \neq B$ $\rightarrow$**create**$[leafSplit(c, A \cup B)]$,**eliminate**$[leafSplit(c, A), leafSplit(c, B)]$
a_9	Extension of an added subgraph if a relationship to another added concept exists: $a, b, r \in O_{new} \wedge A \subseteq O_{new} \wedge addSubGraph(a, A) \wedge addC(b) \wedge addR(r) \wedge$ $r_{source} = a \wedge r_{target} = b \wedge r_{type} \in \{'is\_a', 'part\_of'\}$ $\rightarrow$**create**$[addSubGraph(b, \{a\} \cup A)]$,**eliminate**$[addSubGraph(a, A), addC(b), addR(r)]$
a_{10}	Aggregation of two added subgraphs if they refer to the same root: $a \in O_{new} \wedge A, B \subseteq O_{new} \wedge addSubGraph(a, A) \wedge addSubGraph(a, B) \wedge A \neq B$ $\rightarrow$**create**$[addSubGraph(a, A \cup B)]$,**eliminate**$[addSubGraph(a, A), addSubGraph(a, B)]$
a_{11}	Aggregation of two added subgraphs if one of them is contained in the other: $a, b, r \in O_{new} \wedge A, B \subseteq O_{new} \wedge addSubGraph(a, A) \wedge addSubGraph(a, B) \wedge$ $addR(r) \wedge r_{source} = a \wedge (r_{target} = b \wedge r_{target} \in B) \wedge r_{type} \in \{'is\_a', 'part\_of'\}$ $\rightarrow$**create**$[addSubGraph(b, \{a\} \cup A \cup B)]$,**eliminate** $[addSubGraph(a, A), addSubGraph(b, B), addR(r)]$
a_{12}	Extension of a deleted subgraph if a relationship to another deleted concept exists: $a, b, r \in O_{old} \wedge A \subseteq O_{old} \wedge delSubGraph(a, A) \wedge delC(b) \wedge delR(r) \wedge$ $r_{source} = a \wedge r_{target} = b \wedge r_{type} \in \{'is\_a', 'part\_of'\}$ $\rightarrow$**create**$[delSubGraph(b, \{a\} \cup A)]$,**eliminate** $[delSubGraph(a, A), delC(b), delR(r)]$
a_{13}	Aggregation of two deleted subgraphs if they refer to the same root: $a \in O_{old} \wedge A, B \subseteq O_{old} \wedge delSubGraph(a, A) \wedge delSubGraph(a, B) \wedge A \neq B$ $\rightarrow$**create**$[delSubGraph(a, A \cup B)]$,**eliminate**$[delSubGraph(a, A), delSubGraph(a, B)]$
a_{14}	Aggregation of two deleted subgraphs if one of them is contained in the other: $a, b, r \in O_{old} \wedge A, B \subseteq O_{old} \wedge delSubGraph(a, A) \wedge delSubGraph(b, B) \wedge$ $delR(r) \wedge r_{source} = a \wedge (r_{target} = b \vee r_{target} \in B) \wedge r_{type} \in \{'is\_a', 'part\_of'\}$ $\rightarrow$**create**$[delSubGraph(b, \{a\} \cup A \cup B)]$, **eliminate**$[delSubGraph(a, A), delSubGraph(b, B), delR(r)]$

b-COG rule (b_2) will create a $delC(c)$ change in $diff_{basic}(O_2, O_1)$. Analogously, (b_1) would create $addC(c)$ in $diff_{basic}(O_2, O_1)$ if c is in O_1 but not in O_2 which corresponds to a $delC(c)$ change (inverse of $addC(c)$) in $diff_{basic}(O_1, O_2)$. A $mapC(a, b)$ change in $diff_{basic}(O_1, O_2)$ has $mapC(b, a)$ as its inverse and, according to rules (b_3, b_4, b_5), requires a correspondence $matchC(a, b)$. Changing the domain and range leads to a $matchC(b, a)$ correspondence and thus to a $mapC(b, a)$ change in $diff_{basic}(O_2, O_1)$. The concept changes in $diff_{basic}(O_2, O_1)$ are only created by rules (b_1) to (b_5) like those of $diff_{basic}(O_1, O_2)$. Hence, there can be no further changes in addition to the changes in the inverse of $diff_{basic}(O_1, O_2)$.

References

[1] Aitken S, Chen Y, Bard J. OBO explorer: an editor for open biomedical ontologies in OWL. Bioinformatics 2008;24(3):443.

[2] Aumueller D, Do HH, Massmann S, Rahm E. Schema and ontology matching with COMA++. In: Proc of ACM SIGMOD. ACM; 2005. p. 906–8.

[3] Bernstein PA. Applying model management to classical meta data problems. In: Proceedings of conference on innovative database research (CIDR); 2003. p. 209–20.

[4] Bernstein PA, Melnik S. Model management 2.0: manipulating richer mappings. In: Proceedings of the 2007 ACM SIGMOD international conference on management of data; 2007. p. 1–12.

[5] Boeckmann B, Bairoch A, Apweiler R, Blatter MC, Estreicher A, Gasteiger E, et al. The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003. Nucl Acids Res 2003;31(1):365–70.

[6] Ceusters W, Spackman K, Smith B. Would SNOMED CT benefit from realism-based ontology evolution? In: AMIA annual symposium proceedings, vol. 2007; 2007. p. 105.

[7] Day-Richter J. The OBO flat file format specification, version 1.2. <http://www.geneontology.org/go.format.obo-1_2.shtml>.

[8] Day-Richter J, Harris M, Haendel M, Lewis S. OBO-edit – an ontology editor for biologists. Bioinformatics 2007;23(16):2198.

[9] Euzenat J, Shvaiko P. Ontology matching. New York: Springer-Verlag; 2007.

[10] Flicek P, Aken BL, Beal K, Ballester B, Caccamo M, Chen Y, et al. Ensembl 2008. Nucl Acids Res 2008;36(Database issue):D707–14.

[11] Flouris G, Manakanatas D, Kondylakis H, Plexousakis D, Antoniou G. Ontology change: classification and survey. Knowl Eng Rev 2008;23(2):117–52.

[12] Gene Ontology Consortium. The gene ontology project in 2008. Nucl Acids Res 2008;36(Database issue):D440–4.

[13] Ghazvinian A, Noy N, Musen M. Creating mappings for ontologies in biomedicine: simple methods work. In: Proc of AMIA annual symposium; 2009.
[14] Gross A, Hartung M, Kirsten T, Rahm E. Mapping composition for matching large life science ontologies. In 2nd Intl conference on biomedical ontology (ICBO); 2011.
[15] Hartung M, Gross A, Rahm E. CODEX: exploration of semantic changes between ontology versions. Bioinformatics 2012;26(6):895–6.
[16] Hartung M, Kirsten T, Gross A, Rahm E. OnEX: exploring changes in life science ontologies. BMC Bioinformatics 2009;10:250.
[17] Hartung M, Kirsten T, Rahm E. Analyzing the evolution of life science ontologies and mappings. In: Data integration in the life sciences (DILS); 2008. p. 11–27.
[18] Hartung M, Terwilliger J, Rahm E. Recent advances in schema and ontology evolution. In: Bellahsene Z, Bonifati A, Rahm E, editors. Schema matching and mapping. Springer; 2011. p. 149–90 [chapter 6].
[19] Hayamizu TF, Mangan M, Corradi JP, Kadin JA, Ringwald M. The adult mouse anatomical dictionary: a tool for annotating and integrating data. Genome Biol 2005;6(3):R29.
[20] Hu W, Qu Y. Falcon-AO: a practical ontology matching system. Web Semantics: Sci. Services Agents World Wide Web 2008;6(3):237–9.
[21] Huang D, Sherman B, Lempicki R. Bioinformatics enrichment tools: paths toward the comprehensive functional analysis of large gene lists. Nucl Acids Res 2009;37(1):1.
[22] Kirsten T, Gross A, Hartung M, Rahm E. Gomma: a component-based infrastructure for managing and analyzing life science ontologies and their evolution. J Biomed Semantics 2011;2:6.
[23] Kirsten T, Hartung M, Gross A, Rahm E. Efficient management of biomedical ontology versions. In: OTM workshops; 2009. p. 574–83.
[24] Klein M, Fensel D, Kiryakov A, Ognyanov D. Ontology versioning and change detection on the web. Knowl Eng Knowl Manage: Ontol Semantic Web 2002:247–59.
[25] Lambrix P, Tan H. Sambo – a system for aligning and merging biomedical ontologies. Web Semantics: Sci Services Agents World Wide Web 2006;4(3):196–206.
[26] Maere S, Heymans K, Kuiper M. Bingo: a cytoscape plugin to assess overrepresentation of gene ontology categories in biological networks. Bioinformatics 2005;21(16):3448.
[27] McCann R, Shen W, Doan A. Matching schemas in online communities: a web 2.0 approach. In: 24th International conference on data engineering; 2008. p. 110–9.
[28] Moreira D, Musen M. OBO to OWL: a protege OWL tab to read/save OBO ontologies. Bioinformatics 2007;23(14):1868.
[29] Noy NF, Shah N, Dai B, Dorf M, Griffith N, Jonquet C, et al. Bioportal: a web repository for biomedical ontologies and data resources. In: Proc of ISWC; 2008.
[30] NoyNF, Chugh A, Liu W, Musen MA. A framework for ontology evolution in collaborative environments. In: The semantic web – ISWC 2006, 5th international semantic web conference; 2006. p. 544–58.
[31] Noy NF, Kunnatur S, Klein M, Musen MA. Tracking changes during ontology evolution. In: Proc of ISWC; 2004. p. 259–73.
[32] Noy NF, Musen MA. PromptDiff: a fixed-point algorithm for comparing ontology versions. In: Proceedings of the national conference on artificial intelligence; 2002. p. 744–50.
[33] Oliver D, Shahar Y, Shortliffe E, Musen M. Representation of change in controlled medical terminologies. Artif Intel Med 1999;15(1):53–76.
[34] Papavassiliou V, Flouris G, Fundulaki I, Kotzinos D, Christophides V. On detecting high-level changes in RDF/S KBs. In: Prof of ISWC; 2009. p. 473–88.
[35] Plessers P, Troyer O. Ontology change detection using a version log. In: Proc of ISWC; 2005.
[36] Prüfer K, Muetzel B, Do H, Weiss G, Khaitovich P, Rahm E, et al. Func: a package for detecting significant associations between gene sets and ontological annotations. BMC Bioinformatics 2007;8(1):41.
[37] Rahm E. Towards large scale schema and ontology matching. In: Bellahsene Z, Bonifati A, Rahm E, editors. Schema matching and mapping. Springer; 2011. p. 3–27 [chapter 1].
[38] Rahm E, Bernstein PA. A survey of approaches to automatic schema matching. VLDB J 2001;10(4):334–50.
[39] Sioutos N, Coronado S, Haber MW, Hartel FW, Shaiu WL, Wright L. NCI thesaurus: a semantic model integrating cancer-related clinical and molecular information. J Biomed Inform 2007;40(1):30–43.
[40] Smith B, Ashburner M, Rosse C, Bard J, Bug W, Ceusters W, et al. The OBO foundry: coordinated evolution of ontologies to support biomedical data integration. Nat Biotechnol 2007;25(11):1251–5.
[41] Smith B, Ceusters W, Klagges B, Köhler J, Kumar A, Lomax J, et al. Relations in biomedical ontologies. Genome Biol 2005;6(5):R46.
[42] Smith C, Goldsmith C, Eppig J. The mammalian phenotype ontology as a tool for annotating, analyzing and comparing phenotypic information. Genome Biol 2004;6(1):R7.
[43] Stojanovic L, Maedche A, Motik B, Stojanovic N. User-driven ontology evolution management. Knowl Eng Knowl Manage: Ontol Semantic Web 2002:133–40.
[44] Sure Y, Erdmann M, Angele J, Staab S, Studer R, Wenke D. Ontoedit: collaborative ontology development for the semantic web. In: Proc of ISWC; 2002. p. 221–35.
[45] Tirmizi S, Aitken S, Moreira D, Mungall C, Sequeda J, Shah N, et al. Mapping between the obo and owl ontology languages. J Biomed Semantics 2011;2(Suppl 1):S3.
[46] Yang Z, Zhang D, Ye C. Ontology analysis on complexity and evolution based on conceptual model. In: Data integration in the life sciences. Springer; 2006. p. 216–23.
[47] Zhdanova AV, Shvaiko P. Community-driven ontology matching. In: ESWC; 2006. p. 34–49.