University of Zurich[UZH]

IncVer - An Incremental Versioning System for OBO Ontologies

Master Thesis June 6, 2019

Felix Kieber
of Schaan, Liechtenstein

Student-ID: 08-731-945
felix.kieber@uzh.ch

Advisor: **Romana Pernischova**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

Writing a thesis is always a daunting task, but it would be much more challenging without the support of others, both professionally and emotionally. As such I would like to extend thanks to Professor Bernstein and the *Dynamic and Distributed Information Systems Group* for giving me the opportunity to write my master's thesis in their field of research. Special thanks goes to Romana Pernischova, my advisor, for giving me frequent, helpful feedback and input, and ample opportunity for questions and discussion, from beginning to the end. Finally, I would like to thank my parents and brothers for their patience and support during my work on this thesis and for picking up the slack walking the dog.

Zusammenfassung

Diese Masterarbeit gibt eine kurze Einführung und Übersicht zu den Forschungsfeldern der Evolution von Ontologien (*ontology evolution*) und Wirkungsanalyse von Veränderungen (*impact analysis*), untersucht COntoDiff, ein Tool zur Erkennung von Veränderungen zwischen Ontologie-Versionen und präsentiert eine Implementierung von IncVer, einem Programm zur Generierung von inkrementellen Versionen.

Die obengenannten Forschungsfelder befassen sich mit Veränderungen die an einer Ontologie durchgeführt werden. Deshalb sind verschiedene Momentaufnahmen, oder *Versionen* von grossem Interesse. Viele Ontologien stellen allerdings nur wenige Versionen zur Verfügung, wenn überhaupt, welche oftmals zeitlich weit auseinander liegen und oft hunderte bis tausende einzelne Veränderungen abdecken. Solch grosse Mengen lassen meist nur relativ grobe Einsichten zu Eigenschaften und Auswirkungen von Veränderungen zu.

IncVer erlaubt das Generieren von detaillierten Evolutionsdatensätzen. Das Programm liest dabei zwei Ontologie-Versionen, identifiziert und gruppiert Veränderungen und erstellt dann inkrementell eine Version für jede Gruppe von Veränderungen. IncVer baut auf COntoDiff auf und unterstützt bisher das OBO Ontologie Format. Ein Hauptaugenmerk liegt aber auf der Erweiterbarkeit der Software. Dazu wurde die IncVer-Architektur in drei separate Komponenten unterteilt, die zusammen eine Pipeline bilden. Sie besteht aus dem *Diff Calculator*, sowie den *Ordering* und *Applying* Komponenten. Ersterer ist zuständig für das Berechnen eines sogenannten *diff*s, einer Liste von Veränderungen zwischen zwei Versionen. Die folgende Komponente sortiert das resultierende diff und die *Applying* Komponente wendet die Änderungen schliesslich inkrementell an. Es wurde eine Grundimplementierung für alle drei Komponenten erstellt.

Um die Richtigkeit der Resultate zu verifizieren wurden drei Bedingungen formuliert, die erfüllt sein müssen damit die erstellten Versionen als richtig erachtet werden. Mit diesen Bedingungen, angewendet als Metriken, war es möglich vielversprechende Ergebnisse zu erzielen, die die Anwendbarkeit von IncVer in der Ontologieversionierung sowie die potenzielle Verwendbarkeit in den Forschungsfeldern Ontology Evolution und Impact Analysis demonstrieren.

Eine Jar Distribution von IncVer wurde erstellt, die die Grundimplementierung sowie die Evaluationsfunktionalität beinhaltet.

Abstract

This master thesis contains an introduction and overview on the field of ontology evolution and ontology versioning, an inspection of the ontology change detection tool COntoDiff and an implementation of the incremental version generation tool IncVer.

The fields of ontology evolution and impact analysis are interested in the changes that occur in an ontology. As such, snapshots in time, or *versions*, are of great interest to researchers. Many ontologies, however, provide only few versions, if at all, and these are often far apart in time and contain hundreds to thousands of changes. These large changes only allow rough analysis of their nature and impact.

IncVer is a tool which allows the generation of detailed evolution datasets, taking two input ontology versions and detecting and grouping the changes between these versions. Then, incremental versions are built, one per change action, building from the old version to the new version. IncVer is built on top of COntoDiff and so far supports the OBO ontology format, but is designed to be extensible at its core. In order to achieve this, the IncVer architecture is separated into three components forming a pipeline: The *Diff Calculator*, the *Ordering* and the *Applying* component, responsible for calculating a diff, sorting the resulting diff and applying the changes in that diff, respectively. A base implementation is provided for all three components.

To ensure correctness of the results, three conditions were formulated which need to be met for the generated versions to be considered correct. Applying these conditions as metrics, I was able to achieve promising results, demonstrating the applicability of IncVer to ontology versioning and its potential use to the fields of ontology evolution and impact analysis.

A Jar distribution of IncVer is provided, encapsulating the base implementation of the pipeline, as well as the evaluation functionality.

Table of Contents

1

Introduction

The concept of *Semantic Web* has been in use since the term has been coined in the seminal article by Berners-Lee et. al [2]. While it has undergone varying levels of hype and disinterest in the industry in its close to twenty years of existence, the academic interest remains unbroken.

The term *Semantic Web* encompasses a big array of topics, areas of research and fields of application, echoing the complexity and scope of the world wide web itself. A particular sphere of interest is the area of *ontology evolution*, which itself can be separated in various subtopics. Comprehensive overviews have been presented by Khattak et al. [12] and Zablith et al. [25]. In particular, Zablith et al. [25] provide a detailed survey of the various stages ontology evolution can be separated into. According to them, ontology evolution can be considered one of 11 stages of the broader *ontology change* process. They separate ontology evolution into five steps: **Detecting the Need for Evolution**, **Suggesting Changes**, **Validating Changes**, **Assessing Impact** and finally **Managing Changes** — the last two will be of particular interest to this thesis.

A common use case of ontologies, or knowledge graphs (KG) in general, is to run computations and queries on them, ranging from answering simple queries like: "When was Ernest Hemingway born?", to expensive operations like computing the logical closure of an ontology or executing functional analyses. Changes in the ontologies and KGs naturally lead to some of the results being invalidated. While Hemingway's date of birth will never change, a list of film adaptations of his books might. For complex and expensive operations it can be beneficial to know whether the change in the underlying ontology has significant influence on the result of said operations and research in this area of *impact analysis* is ongoing [19, 8].

To analyse the impact of changes in ontologies it is helpful to have different versions of real-world ontologies to capture real-world changes instead of synthetically produced ones; Pernischova [19] works with this approach, for instance. *Ontology versioning* is defined as "the ability to handle changes in ontologies by creating and managing different variants of it" by Klein and Fensel [13] and is its own area of research; various approaches have been suggested to capture or reconstruct [5], and represent and track [6, 9] changes between ontologies. In contrast to previous publications, which focus on ontology evolution only as "updating the ontology based on the required changes" and considering *ontology versioning* as a separate task, Zablith et al. [25] emphasize that ontology versioning is "intrinsically linked" to ontology evolution.

Unfortunately, for many, readily available ontologies only few versions other than the current ones exist, if at all. Those ontologies which maintain versions usually only produce a release every few months or in even bigger intervals. Because of this, the number of changes between versions is often very high and analysing change impact yields only very coarse results. This thesis looks to amend this situation by developing an approach to first splitting up the high number of changes between two versions into groups of smaller changes and then generating incremental versions based on these smaller groups of changes.

Most of the approaches towards change detection and classification are either very early publications such that no working implementation could be found, like [16], or remain mostly theoretical, as in [18]. An exception to this is *COntoDiff*, an approach presented by Hartung et al. [9]. *COntoDiff* refers to both the algorithm presented in the paper, as well as a Java implementation of the algorithm provided along with it[1]. *COntoDiff* first detects low-level changes between two input ontology versions and in a subsequent stage aggregates these changes into higher-level changes. Along with this, it provides a catalog of low- and high-level changes.

The goal of this thesis is thus to implement an incremental version generation system, or *incremental versioning system*, employing change actions as the basic steps between incremental versions, to aid in a more granular analysis of ontology change. After conducting a thorough literature research, assessing the state of the art in the area of *ontology evolution*, *COntoDiff* by Hartung et al. [9] was chosen as a foundation for the *incremental versioning system*. Using the existing *COntoDiff* implementation, a collection of change actions is generated, which occur between two input ontology versions. I devised a way in which these change actions can be ordered, such that they can be incrementally applied to the earlier or old input version, step by step building towards the later or new input version. The main challenge here lies in, firstly, ordering the change actions in such a manner that no unmet dependencies occur, like adding a connection to an entity that has not been added yet and, secondly, applying change actions correctly to an ontology. All these steps were integrated into an extensible software framework, named IncVer. Finally, the framework was tested on five different ontologies and evaluated by imposing conditions that were specifically devised for this purpose. Applying metrics based on said conditions, I was able to achieve promising results, demonstrating the applicability of IncVer on various ontologies.

The central contribution of this thesis is thus IncVer, a Java framework for incremental ontology version generation. Extensibility is a core concern of the framework and a component-based architecture allows for flexible customization of one or more parts of the execution pipeline. Along with the modular structure, a readily-usable base implementation for versioning OBO ontologies is provided as well; a JAR distribution for use in the Command Line is made available. Concretely, this thesis also contributes an approach to ordering change actions such that no dependency conflicts occur.

This thesis is structured as follows: after this introduction to the wider field of ontology evolution and ontology versioning in particular, the chapter proceeds with a brief theoretical introduction of relevant concepts and terms, such as the ontol-

[1] *https://dbs.uni-leipzig.de/de/research/projects/evolution_of_ontologies_and_mappings*

ogy models and the OBO format, before concluding with related work. After this follows an in-depth analysis of COntoDiff in Chapter 2, the chosen foundation for IncVer, introducing the catalog of *change actions* it defines, outlining its functionality and highlighting its limitations and their impact on the implementation of IncVer.

Chapter 3 is dedicated to the implementation of IncVer. It starts top-down with an introduction to the high-level architecture, followed by a segment listing the improvements and fixes made to COntoDiff in preparation for use in IncVer. The chapter then proceeds with implementation details of the framework, showcasing the base implementations for each stage in the IncVer pipeline.

Finally, in Chapter 4, I will establish three conditions for evaluating the results of IncVer. Subsequently I present and discuss the results of applying said conditions to the generated incremental versions of five different datasets. This thesis will then close by presenting limitations of the work done, potential future work and final conclusions.

1.1 Background Knowledge & Terminology

Before going into the specifics of COntoDiff and the practical work, it is helpful to first lay out some basic theory and terminology so as to prevent any confusion or misunderstanding due to subtle differences between the theoretical description of the *COntoDiff*-algorithm and its concrete implementation. Thus the section begins by briefly defining *diffs* in the context of ontology versioning, then I will introduce the ontology model used by the *COntoDiff*-algorithm, followed by a comparison to the *OBO flat file format*, which is used in the implementation.

1.1.1 Diffs

The list of changes between two ontology versions is referred to as *evolution mapping* or *diff*, borrowing the term from version control systems such as Git. The changes themselves are also, mostly synonymously, called *change actions* or *change operations*. Various models have been proposed to classify possible changes with a very detailed breakdown presented in [25], but a common approach is to seperate changes into *basic* or *low-level* and *complex* or *high-level* changes. In general, low-level changes are *atomic* changes, that represent a single, structural change, like updating a value or adding a new entity. High-level changes are composite, often semantic changes, grouping multiple operations, for instance deleting a person from an ontology, along with its properties such as age or gender. Various catalogs of change actions or *change languages* have been proposed on different levels of granularity and employing various formalisms, such as OWL, RDF or Generic formalisms. See [25] for more details.

1.1.2 Ontology Model

So far, nothing specific has been said on what an ontology looks like. There are various ontology formats, the most important of which in all likelihood is the *Web*

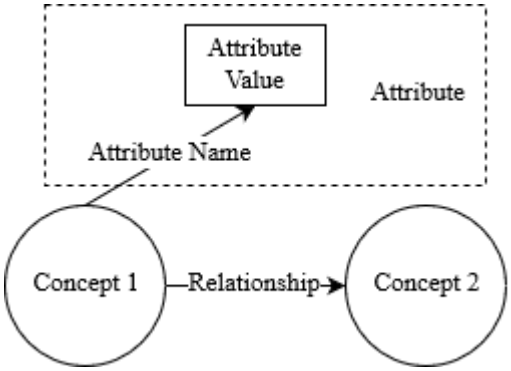

Figure 1.1: The CAR Ontology Model

Ontology Language, specifically *OWL 2*, developed and maintained by the W3C[2] (see [1]). Being a highly expressive ontology language, it is part of the W3C's semantic web stack and supports various serialization syntaxes such as RDF/XML, Functional Syntax or Turtle. Another format that is heavily used in the area of life sciences is the *OBO Flat File Format* or simply OBO format, which has its own syntax, specified in [11]). OBO is a strict subset of OWL 2.

COntoDiff opts for a relatively simple ontology model, which I will, for lack of an established term, simply call the *CAR* model, the reason for which will be apparent shortly. This model is strongly related to the *OBO ontology format* and directly maps to it. This ontology model consists of three types of elements, detailed in the following paragraphs. See Figure 1.1 for a visual representation of the model. In addition, Listing 1.1 shows a term stanza from the gene ontology. While explaining the parts of the CAR model, I will point out the corresponding lines in the listing.

Concepts are the entities in a CAR model, more specifically *classes*. Seeing a CAR-ontology as a graph structure, they are the vertices. Concepts are similar to, but less complex than classes in OWL 2. They are identified by an *id*, also referred to as *accession number* (mostly in the context of OBO). Concepts correspond closely to *terms* in the OBO-format, which must have at least an *id* and a *name* (a *label* in OWL terminoloy). The whole term stanza in Listing 1.1 corresponds to a term, and is uniquely identified by the id in line 2.

Attributes consist of an *attribute name* and an *attribute value* and are associated with a *concept*. As the name states, they denote certain properties that a concept can have. The data type of an attribute value is not further specified, but for our purposes are primitive data types, such as *string*, or *boolean*. Within a graph structure, attributes can be seen as a vertex-edge pair that connects a value to a concept using an attribute name, or tag. Attributes map to *tag-value* pairs of *terms* in OBO. Lines 2 through 7 represent attributes of the term. Strictly speaking, the id is also an attribute of the term.

Relationships connect a *source concept* to a *target concept* and have a name, but no associated attributes. In the context of a graph structure, relationships are directed, named edges. They correspond to *relationship* and *is_a* tags of terms in

[2] *https://www.w3.org/*

OBO. It is worth pointing out the distinction between instance-level relationships, which connect concepts, and schema-level relationship definitions which provide the blueprint for relationships. Analogous to the idea of classes vs. instances in object oriented programming, relationships are instances of relationship definitions on the schema level. The CAR model has no notion of relationship definitions — OBO, however, does. I will go into this difference in a later section. In Listing 1.1 lines 8 and 9 denote the outgoing relationships of the term. On line 8 is a *is_a* relationship connecting to `SO:0000143`, which is the most important relationship in many GO ontologies and hence has its own tag. Line 9 shows a generic relationship tag, in this case the term is a *part_of* term `SO:0000149`.

Listing 1.1: An OBO Term from the Gene Ontology

```
1 [Term]
2 id: SO:0000007
3 name: read_pair
4 def: "A pair of sequencing reads in which..." [SO:ls]
5 subset: SOFA
6 synonym: "read-pair" EXACT []
7 property_value: finalized_on "08-04-2018" xsd:string
8 is_a: SO:0000143 ! assembly_component
9 relationship: part_of SO:0000149 ! contig
```

1.1.3 Relation to OBO Format

It is important to note that the CAR model is only an informal model. Given that [9] originates from the area of life sciences and the gene ontology, it seems likely that the model was defined simply as a slightly more generic version of the OBO format. This is further supported by the fact that the implementation of COntoDiff works with the OBO file format.

As noted above, *concepts*, *attributes* and *relationships* closely correspond to elements of the OBO format. There are however some key differences between the CAR model and the OBO format, which are particularly relevant when looking at the limitations presented by COntoDiff.

Attributes vs. Tags: Attributes in the CAR model are a modification of tag-value pairs of terms in OBO (why I add the "of terms" part will be explained in the next subsection). In theory there can be an attribute for anything, that is, any attribute name is possible. In contrast, OBO is restricted to a predefined set of tags, that is, attribute names. This can be worked around however, with the `property_value` tag. This tag consists of a property name, a value and a data type, functioning as a way to specify custom properties. For an example, see Listing 1.1, line 7.

While the attribute values in the CAR models can be considered single-valued, the tag-value pairs in OBO have a predefined cardinality and often allow and in some cases even require multiple values. Since attributes are not explicitly defined as using primitive data type values, they could also make use of composite data

types. However, the actual implementation parses values as single strings and thus, for all intents and purposes, can be considered single-valued primitive data types.

Relationships and Typedefs: The distinction between relationship instances and schema-level definitions has been mentioned earlier in 1.1.2. Relationships in the CAR model stand on their own and have no associated attributes or structure except their name and connected concepts. The OBO format, however, supports *typedefs*, which are basically schema level definitions of relationships. Along with their name and id, typedefs can also have tag-value pairs, i.e. associated properties — this is also the reason why I emphasized that attributes correspond to tag-value pairs of terms.

1.2 Related Work

The first notable contribution to change detection between ontology versions is PromptDiff [16], introducing the notion of a *structural diff*, capturing the structure of a ontology as opposed to an arbitrary text serialization. This approach has been expanded on by Tury and Bieliková [23]. They introduce the distinction between changes on the schema or *structural* level, consisting of elements such as classes or relationship domains, and the level of individuals or *content*, meaning instances of classes. Both works employ heuristics in their algorithms for change detection.

COntoDiff [9] is both an algorithm and an implementation of that algorithm for change detection between versions of life science ontologies. They employ some heuristics described in the previously mentioned papers and produce an evolution mapping consisting of a comprehensive catalog of low- and high-level change actions. Other contributions to change detection exist of varying levels of granularity in change actions and supported formalisms, such as SemVersion [24], recording low-level changes based on RDF and RDFS. Kontchakov et al. [14] provide a low-level formal framework for diffs of DL-Lite (description logics) ontologies. Finally, Papavassiliou et al. [18] focus on high-level changes in RDF/S ontologies. Zablith et al. [25] compiled a comprehensive overview of change detection approaches and the process of ontology evolution as a whole.

Frommhold et al. [6] provide an approach to RDF versioning, as a foundation for a RDF version control system, addressing issues such as blank nodes or the possibility of unperceived manipulation. They also define a vocabulary to describe changes in an RDF dataset. Papakonstantinou et al. [17] introduce the *Semantic Publishing Versioning Benchmark* to evaluate the performance of ontology versioning system. The DBpedia Wayback Machine [5] is a semantic web pendant to the Internet Archive Wayback Machine[3], capable of reconstructing past iterations of Wikipedia articles converted to RDF versions.

[3] *https://archive.org/web/*

2

COntoDiff — An Analysis

The implementation of COntoDiff as presented in [9] (henceforth simply referred to as COntoDiff) forms the basis of the work in this thesis. As such, it warrants a closer inspection, disseminating what it does and, just as importantly, what it does not do. Section 2.1 looks at the capabilities of COntoDiff, detailing the stages it undergoes and listing the low- and high-level changes it supports. Section 2.2 then examines the shortcomings of COntoDiff, before the chapter closes with Section 2.3 detailing the consequences for the design and implementation of IncVer.

2.1 Inspection of COntoDiff

COntoDiff is the implementation of the *COntoDiff*-algorithm as it is laid out in [9]. A high-level recap of the algorithm will follow, as part of its inspection but for a more in-depth look, refer to the paper.

2.1.1 Stages

From a high-level point of view, the process of COntoDiff can be separated into three stages, where the output of each stage is used as input for the next stage, with the last stage producing the final *complex diff*. A set of so-called *Change Operation Generating Rules* (COG rules) determines the intermediate and final diffs of this process, generated at each stage and consisting of basic (low-level) and complex (high-evel) change operations. There is a set of COG rules for each stage, which are applied to the stage-input and produce the output. For a detailed listing of all COG-rules, refer to [9]. These diffs consist of *change actions*, which describe a self-contined change between two ontology versions. A note on terminology: *low-level* and *basic change (actions)* are used interchangeably, as are *high-level* and *complex*.

Basic Diff: This stage takes two ontologies, the *old* and *new* version, as input and calculates a *basic diff*. Formally, the basic COG rules (b-COG) are applied to the input ontologies in this stage. In practice, the process is less clearly structured and first determines newly added and removed concepts, attributes and relationships. Furthermore, with minimal use of heuristics similar to those introduced in [16], some mappings between old and new elements are determined. Unlike the next two stages, this stage is hard-coded and not customizable or extensible without directly editing the source code. The result is a diff consisting of nine possible low-level change actions:

- Add-, Delete-, MapConcept
- Add-, Delete-, MapAttribute
- Add-, Delete-, MapRelationship

As can be seen, there is an *add*, *delete* and *map* action for the three element types *concept*, *attribute* and *relationship*. Furthermore, note, that the *map* actions only record elements that have been modified — elements that remained wholly unchanged are not explicitly included in the diff.

Initial Complex Diff: In this stage, the complex COG rules (c-COG) are applied to the basic diff from the previous stage along with the input ontologies. These rules are specified in an XML file and could in theory even be extended or modified. They look for predefined "patterns", i.e. constellations of low-level change actions and ontology elements. An example of such a c-COG rule, specifically C_8, can be seen in Formula 2.1.

$$\begin{aligned} & a, b \in O_{old} \wedge c \in O_{new} \wedge mapC(a, c) \wedge mapC(b, c) \wedge a \neq b \\ & \wedge \nexists d(d \in O_{new} \wedge mapC(a, d) \wedge c \neq d) \\ & \wedge \nexists e(e \in O_{new} \wedge mapC(b, e) \wedge c \neq e) \\ \rightarrow & create[merge(\{a\}, c), merge(\{b\}, c)], \\ & eliminate[mapC(a, c), mapC(b, c)] \end{aligned} \tag{2.1}$$

This rule checks if there are two distinct concepts from O_{old} that map to the same concept in O_{new}, that also do not map to any other concept in O_{new}. If so, the high-level *merge* actions are created and the corresponding input change actions ($mapC$) are eliminated. It is important to highlight that two separate *merge* actions are generated, instead of one $merge(\{a, b\}, c)$. This aggregation is actually done in the next and final step, since more than two elements can be aggregated.

Aggregation: In this final stage, the set of aggregation COG rules (a-COG) are repeatedly applied to the result of the previous stages, until no new changes are generated. This is to allow the aggregation of any number of elements. Staying with the example of the high-level action *merge*, Formula 2.2 describes the rule which aggregates *merges* that merge into the same concept c. Note that lower-case letters refer to concepts, while upper-case letters denote sets of concepts. Applying 2.2 repeatedly results in one change action $merge(X, t)$ where $X = \{a\} \cup \{b\} \cup \{c\} \cup ...$ for all concepts that merge into t.

$$\begin{aligned} & c \in O_{new} \wedge A, B \subseteq O_{old} \wedge merge(A, c) \wedge merge(B, c) \wedge A \neq B \\ \rightarrow & create[merge(A \cup B, c)], \\ & eliminate[merge(A, c), merge(B, c)] \end{aligned} \tag{2.2}$$

The result of this stage is the final, *complex* or *compact diff*, containing the highest-level actions, that is, all actions that do not belong to any other complex change action. On one hand, not all low-level actions need to belong to a higher-level action (e.g. *addAttribute* can appear on its own), while on the other hand, high-level change actions can be aggregated into other high-level change actions (*addLeaf* can belong to *addSubGraph*). In practice, COntoDiff keeps a list of all basic and complex change actions, as well as a mapping between them.

Change Action	Status	Notes
substitute	supported	Replace a concept by another concept
toObsolete	supported	Set `is_obsolete` attribute to *true*
revokeObsolete	supported	Inverse of *toObsolete*
move	supported	Move a concept and its subgraph from one concept to another. De facto "re-wires" a relationship to a new target concept
chgAttValue	supported	
addLeaf	supported	
delLeaf	supported	
addInner	new	
delInner	new	
merge	supported	Merge two or more concepts into a single one
split	supported	Split a concept into two or more concepts
addSubGraph	supported	Aggregation of *delInner* and *delLeaf* actions
delSubGraph	missing	
leafMerge	missing	specific case of *merge*
leafSplit	missing	specific case of *split*

Table 2.1: Supported High-Level Change Actions

2.1.2 List of Supported High-Level Change Actions

Table 2.1 lists all the high-level change actions that are currently supported by COntoDiff and, by extension, also by IncVer, marked as either *supported* or *new* in the *Status* column. Change actions marked as *new* are supported but not described in [9].

Furthermore, Table 2.1 also lists high-level change actions that are described in [9] but are not implemented in the available version of COntoDiff, marked as *missing* in the column *Status*. The reasons for this are not clear. One explanation could be, that these change actions simply did not occur or only rarely. This is supported by the evaluation conducted in [9], which lists no occurrences of *addSubGraph* and *leafMerge* change actions in the evaluated datasets. *leafSplits* did occur often, but no *splits* so perhaps the specific *leaf-* cases were removed to have uniform *split* and *merge* actions, which are more general versions of the *leaf-* versions. Another explanation for the lack of a *delSubGraph* change action may be technical difficulties. However, these are all speculations.

2.2 Limitations of COntoDiff

COntoDiff unfortunately also lacks some functionality in regards to the goal of this thesis, some of which has already been hinted at in the previous section. This section will list the main limitations of COntoDiff relevant to the implementation of IncVer.

2.2.1 No Real OWL Support

COntoDiff is claimed to support a subset of OWL. In practice however, this statement can be disregarded. To understand why, it is worth elaborating on the relation between OWL and OBO. Tirmizi et al. [22] take a look at how these two formats relate to each other and determines that OBO is a strict subset of *OWL DL*, that is, that every expression in OBO can also be expressed in OWL, however many OWL constructs cannot be expressed in OBO. In addition to this analysis, the paper provides a mapping from OBO to OWL.

While many mappings are readily apparent (*term* → *owl:Class*, *is_a-tag* → *rdfs:subClassOf*), some others are not quite so obvious. For instance, the OBO *relationship* tag is a relation between classes, not between instances. As such, it is mapped to an OWL construct using *rdfs:subClassOf* and *property restrictions* (*owl:Restriction*) to model this relation.

To relate this back to the topic at hand: using this mapping, only OWL documents of a very specific structure can be converted to OBO — in practice these are only documents which have been converted from OBO to OWL before with this mapping. Unfortunately, COntoDiff only parses these aforementioned structures in an OWL document and does not understand general OWL axioms. As such, for any practical purpose, COntoDiff does not support OWL.

2.2.2 Limited Set of Tags Supported

There is a discrepancy between arbitrary attributes that are supported by the CAR model in theory, the fixed set of tags that OBO defines and the attributes/tags that are actually recognized by COntoDiff. Specifically, COntoDiff only parses the following tags:

- id
- name
- is_obsolete
- xref
- def
- alt_id
- exact/broad/related/narrow_synonym
- synonym
- is_a
- relationship

While these tags are arguably among the most common ones, considering that the OBO spec defines more than 20 possible tags, COntoDiff is far from covering the full OBO spec.

2.2.3 Primitive Tag Value Parsing

In addition to the limited set of tags that are supported, the parsing of these values is also rather rudimentary. Particularly noteworthy is the handling of *annotations*,

or lack thereof. Annotations are additional, often optional arguments that provide further information on the attribute value. For instance, many tags in OBO support an optional trailing list of so-called *dbxrefs*, which are generally references to other datasets or wikis relevant to the tag, often taking the form of hyperlinks. COntoDiff does not parse these, so in the case that only a dbxref of an attribute changes, COntoDiff would not detect this change. However, this case probably occurs not very often, if at all.

Another noteworthy omission is the *scope* of a synonym tag. In OBO, synonyms can be of different types, or scope: EXACT, BROAD, NARROW or the default RELATED, which is assumed if no scope is specified. While COntoDiff initially parses the scope, it is not carried over to the change action data and is missing in the diff.

2.2.4 No Parsing of Typedefs

Another rather substantial omission is that *Typedef*-stanzas, specifying schema-level definitions of relationships, are wholly ignored by the COntoDiff OBO parser. Consequently, changes to relationship definitions, such as changes in domain or range, or relationship properties like transitivity or reflexivity, or worse, Typedef additions and removals are not captured at all.

2.2.5 No DeleteSubGraph Change Action

As seen in Table 2.1, there are some change actions that are mentioned in the paper but not implemented in COntoDiff. While the actions *leafSplit* and *leafMerge* are captured by the general *split* and *merge* actions respectively, there is no analogue for *delSubGraph*. This is particularly unexpected since its inverse operation, *addSubGraph* is supported.

It is unclear why these operations have not been implemented. Perhaps the developers came across unforeseen technical issues or the change actions were deemed unnecessary after further development. Uunfortunately no further documentation on this could be found.

2.3 Consequences for IncVer

The previous section featured an explanation of relevant missing features of COntoDiff. This section will now discuss how these insights inform the design and scope of the IncVer implementation. Overall, these limitations were split into two categories: those missing features that could reasonably be implemented via extensions or workarounds, and those that were considered outside the scope of this thesis and thus were accepted into the specification of IncVer.

2.3.1 Implemented Features

The limited support of OBO tags was relatively straightforward to address and has been improved by extending the OBO parser to accept any attribute. This does not

mean, however, that IncVer accepts any tag, since it is still limited to the set of tags defined in the OBO spec — this can however be worked around by using the *property_value* tag, for which support has also been added.

The incomplete parsing of tag values has been improved. Concretely, many attributes are being looked up in the new version input ontology to catch any *dbxrefs* associated with those values. The same process is used to determine the scope of synonyms. However, the full specification of OBO tag-values is still not being captured and improvements in this regard are very much desired. This will be reserved for future work.

Finally, support for Typedefs has been added as well. This is being done separately from the COntoDiff OBO parsing since it was not possible to add without largely rewriting the existing code and extending the *COntoDiff* algorithm. As such, Tyepdefs are parsed independently from the new and old version input ontologies and a very basic diff is calculated, including added, removed and changed Typedefs. New Typedefs are then included the first time a relationship of that type is added. Remaining Typedef updates (removals, changes and remaining additions) are then added after all other change actions have been applied.

2.3.2 Missing Features Accepted into Specification

A rather minor concession is the decision to forego an implementation of the *delSubGraph* change action. Such an implementation would likely be possible without diving very deep into the COntoDiff implementation as it should, in theory, be possible to specify the change action via the XML configuration file, mentioned in 2.1.1. Ultimately however, the availability of this change action was not as high a priority as other features, as IncVer is still fully functional without it.

In contrast, the lack of support for OWL in COntoDiff represents a larger shift in scope from the original vision. Since the *COntoDiff* algorithm operates strictly on the CAR model, it is not possible to include support for general OWL without significantly extending the algorithm to the point where this could constitute its own research project. Hence, it was decided to focus on the original goal of implement a functioning version of an incremental versioning system on the basis of the existing COntoDiff implementation.

At the same time, this factor informed the emphasis on a modular architecture, which will be discussed thoroughly in Chapter 3. The idea behind this is that at a later point, the *COntoDiff* algorithm may be replaced by a different diff algorithm, with its own format and change actions or equivalent.

3

IncVer Implementation

This chapter is concerned with the design and implementation of IncVer, the incremental version generator for ontologies. Following a top-down approach, the chapter begins with an introduction to the overall architecture of the system, detailing its pipeline architecture consisting of three components, along with general information. Then follows an overview of the changes made to COntoDiff, as it serves as the first component in the pipeline. The subsequent sections contain a look at implementation details of the remaining components that make up IncVer, as well as the pipeline that ties them together.

3.1 High-level Architecture

Initially, IncVer was conceived as general incremental versioning system built on top of COntoDiff. However, after inspecting COntoDiff and determining its functionality and limitations, detailed in Sections 2.1 and 2.2, it became clear that support for general ontologies, including OWL, is not possible solely on the basis of COntoDiff. Given these circumstances, the current architecture of *IncVer* was developed.

The goal behind the architecture is to provide a working implementation employing COntoDiff, but also provide an extensible framework that can support other types of ontologies such as OWL at a later time. Thus, a modular approach was chosen, consisting of three components: the *Diff Calculator*, the *Ordering* component and the *Applying* component, with a pipeline combining these elements in sequential fashion. See Figure 3.1 for a schematic visualization of the architecture with application flow and components. For all three components working implementations were built as part of this thesis which are listed in italics and parentheses in the figure, but the architecture is chosen such that they can be exchanged for different implementations, catering to different needs and ontologies.

Diff Calculator: The frameworks begins by using the *Diff Calculator* to calculate a *diff*. In the current implementation, COntoDiff is used for this step. Later on, this could be exchanged by a diff system for OWL ontologies, for instance. The resulting diff, i.e. a collection of change actions, is passed on to the next component.

Ordering: The *Ordering* component takes the (arbitrarily ordered) diff and applies a specific ordering to it. The simplest conceivable implementation would just shuffle the diff randomly, or, slightly more sophisticated, order the collection by

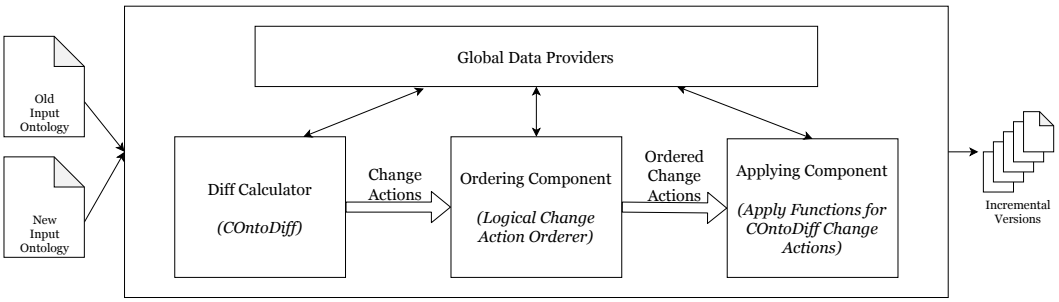

Figure 3.1: The IncVer Architecture, its Components and Base Implementations

change action. The implementation in IncVer chooses an approach that is a bit more involved and will be explained in detail in Section 3.3. Again, this component can later be exchanged for a custom *Ordering* implementation, catering to specific needs. The result of this step is a *sorted collection of change actions*.

Applying: In the final step, the *Applying* component takes the sorted change actions and builds the incremental versions, by applying the change actions, in order, to an evolving ontology, starting with the *old input version*. Proceeding this way, an ontology output file is created after each step. The changes are applied by calling what I term **apply functions**. For every change action that the *Diff Calculator* of choice can generate, a corresponding *apply function* needs to be implemented. This *apply function* takes a concrete change action along with an ontology and applies the change action to it accordingly. For IncVer an *apply function* has been implemented for all change actions supported by COntoDiff, as described in Section 2.1.

The observant reader may already have gleaned from these descriptions, that the individual components are not wholly independent of each other. Most apparently, the set of apply functions in the *Applying* step is determined by the set of change actions that the *Diff Calculator* supports. Additionally, depending on the set of change actions, some dependencies may occur between them (e.g. `A` must be applied before `B`) which means that an arbitrary ordering might be invalid. This is, in fact, the case for COntoDiff, which is also the reason why the base implementation of the *Ordering* component in IncVer is more complex. Details on this point will follow in Section 3.3.

3.2 Changes to COntoDiff

This section features a brief look at the changes that were made to the COntoDiff implementation.

SQLite support: The original implementation used *MySQL* as a backing storage to store change action data and mappings from low- to high-level actions. *MySQL* is, however, a rather heavyweight SQL implementation, that needs to be installed on a host machine and have a daemon running to be accessible. Hence, COntoDiff was migrated to support *SQLite*, a very lightweight SQL implementation that operates in-memory and stores databases to files. *SQLite* does not need to be installed and can be used by simply including a programming library. This makes COntoDiff,

and by extension IncVer, much more portable since the SQL implementation is contained within the software distribution.

Generic Attribute Parser: As mentioned previously in Section 2.2.2, the original attribute parsing in COntoDiff was very limited, only supporting a small set of tags. This functionality was extended and any tag can be parsed now: COntoDiff still checks for known tags, but if no match is found, the extension parses a line according to the pattern: `<tag-name>: <tag-value>`. In addition, whitespaces and end-of-line comments, demarcated with an exclamation mark, are trimmed.

Bugfixes: Finally, while working with COntoDiff, some bugs were encountered and subsequently fixed. Among these were, for instance, typos or faulty regexes.

3.3 Ordering Implementation

The *Ordering* component is, in essence, a sorting function that operates on a unsorted collection of change actions and produces an ordered sequence of change actions. This definition is deliberately left very general to allow for high flexibility. The only implicit condition is that the order of change action produces valid incremental ontology versions.

Later implementations may, for instance, implement a filtering Ordering component that not only sorts the change action but removes certain change actions, or even adds new change actions. In this way, the Ordering component can be thought of as a kind of *transformation* function. In fact, the base implementation in IncVer makes use of this freedom, as we will see later in this section.

3.3.1 Atomic and Composite Change Actions and Application Order

The change actions which COntoDiff generates can be separated in two groups, the *atomic* change actions and the *composite* change actions. These terms are roughly equivalent to basic/complex and low-level/high-level, but are more descriptive for the context of this section.

Atomic change actions are those generated by the basic diff in COntoDiff and consist of *add*, *delete* and *map* operations for *concepts*, *attributes* and *relationships* each, yielding a total of nine atomic change actions. Some of these, like `add-` and `delAttribute` can appear independently in the final, compact diff, others, like `mapAttribute` or `delConcept` are always aggregated into a composite change action.

Composite change actions are the high-level change actions that are created by combining and aggregating atomic changes or other composite changes. The expression *composite* is chosen to emphasize the fact, that they are made up of other change actions. Ultimately, every composite change action of atomic and nested composite change actions can be unravelled into a flat collection of atomic change actions.

This introduces the question of how to sort these atomic change actions, such that no dependencies occur. A composite change action as a whole must be self-contained, i.e. not leave the ontology which it is applied to in an inconsistent state,

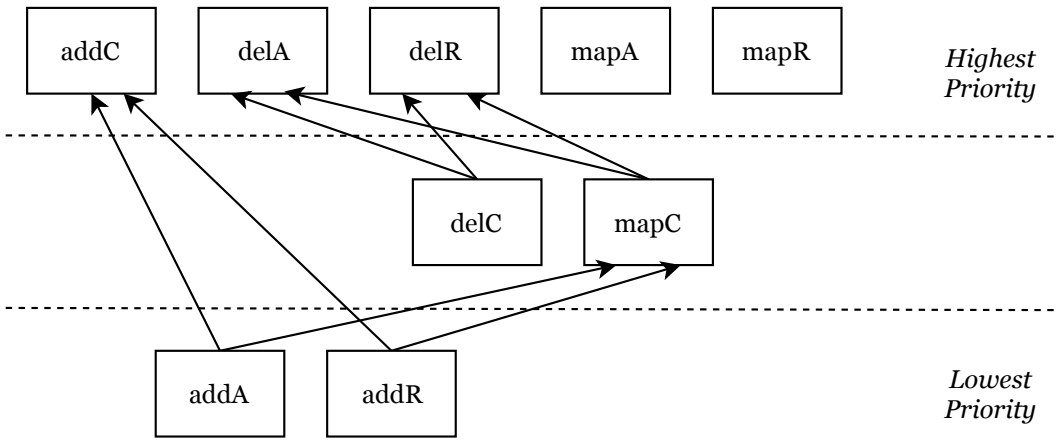

Figure 3.2: The Atomic Change Action Ordering to Avoid Dependencies

like having dangling relationships or attributes that point to concepts that do not exist. As such, when all atomic change actions which make up that composite change action are applied, the ontology must be in a consistent state. However, depending on the implementation of the ontology and the *apply function*, the order in which these atomic change actions are applied also matters. For instance, the `addAttribute` apply function in IncVer does not support adding of attributes to concepts that do not exist. The implementation side of this issue is handled in the *Applying* component for IncVer, but thematically it fits into the topic of ordering.

There is a straightforward solution to this problem: Given the set of atomic change actions, consisting of the nine change actions mentioned above, a global partial order can be defined over them such that no dependencies among them can occur, forming a partially ordered set or *poset*. The ordering is shown in Figure 3.2.

The graph in Figure 3.2 is to be read top to bottom, with change actions placed high coming before those placed below them. The ordering between elements on the same level does not matter and can be chosen arbitrarily — hence, a partial ordering. Arrows indicate that the source may depend on the target change action. The idea behind constructing this order is to place change actions as high as possible, avoiding dependencies on the same level.

Several observations allowed this poset to be constructed. First of all, change actions can never depend on change actions of the same type, hence, there are no loops in our graph. Otherwise it would not be possible to sort atomic change actions based on their type. Furthermore, attributes and relationships associated with concepts must always be removed before the concept they belong to is removed. By the same token, concepts must always be added before associated attributes and relationships are added. The inverse of these two cases — a concept is removed and an attribute/relationship is added to it, and a concept is added and an attribute/relationship is removed from it — logically cannot occur, assuming two valid input ontologies. Finally, as a combined case of the previous point, when a concept is mapped, deletions on it must occur before, and additions to it must be made after it is mapped. This leaves the change actions `mapA` and `mapR`; these have no dependencies and depend on no other change action type, since both `map`s imply, that

their associated concepts exist in both the old and new version and thus are neither added nor deleted.

3.3.2 Dependencies among High Level Change Actions

Initial work on IncVer was done under the assumption that high level change actions generated by COntoDiff were always independent of each other, such that they could be applied in any order. Accordingly, a simple random shuffling function was chosen as the first *Ordering* component to be implemented. However, an initial working prototype soon revealed, that the assumption of independence was false.

I determined, that certain change actions require other change actions to be applied before them, otherwise they fail because certain preconditions (e.g. the existence of a concept) were not met. Specifically, the following case was observed: While applying an `addSubGraph` change action, an exception was thrown because application of an `addRelationship` change action had failed. `addRelationship` was an atomic change action of the `addSubGraph` composite change action and attempted to connect a concept `A` to a concept `B` with relationship `r`. However, concept `B` was missing. The atomic change actions were applied following the sort order described in 3.3.1, which means that the addition of `B` was not part of said `addSubGraph` change action. Further inspection of all change actions in the corresponding diff revealed, that concept `B` was added in a different `addSubGraph` change action.

This might indicate an issue with the rule formulated for `addSubGraph`, but instead of searching for a potential issue deep within the COntoDiff implementation, I opted for a more general solution. This approach is also able to handle other kinds of dependencies that might be encountered and is described in the next subsection. However, inspection of the dependencies revealed another issue: cyclic dependencies can occur. That is, in its simplest form, change action `A` depends on change action `B`, and *vice versa*. More generally, when seeing change actions and their dependencies as a nodes and edges of a graph, respectively, there are cycles between the change actions in question. However, this graph visualization points us in the direction of a solution, as will be discussed in the next section.

3.3.3 Logical Change Action Orderer

The *Logical Change Action Orderer* is the base implementation of the *Ordering* component in IncVer and an answer to the issues explained in the previous section. In essence, it produces a mostly arbitrary ordering, only taking dependency order into account. That is, it guarantees that dependent change actions come after the change actions they depend on, and only that.

A valid ordering can, however, not always be achieved simply by ordering the change actions correctly, due to the previously mentioned possibility of cycles. With cycles, there *is no correct ordering*: Consider, for instance, the interdependent change actions `A` and `B`.`B` depends on `A`, so `A` must come before `B`, but `A` also depends on `B`, thus `B` must come before `A` and so on.

This is why the *logical change action orderer* makes use of the possibility to transform input change actions, as mentioned in Section 3.3. In particular, it determines

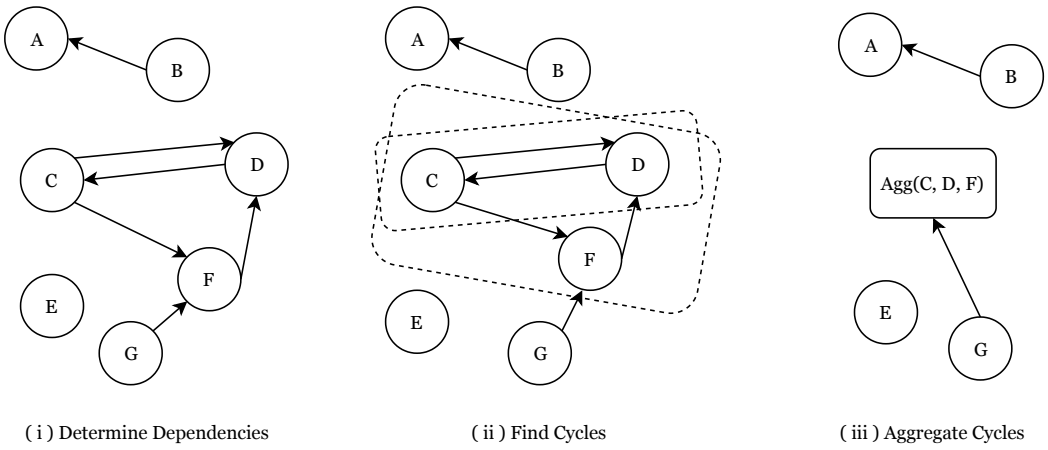

Figure 3.3: Steps of the Logical Change Action Orderer

the cyclic dependencies and aggregates these *dependency clusters* into an aggregate change action, implemented as `AggregateChanges`. Note the distinction between *composite change action*, which is the category of change actions that contain other change actions, and `AggregateChanges`, which is a concrete composite change action, specifically to aggregate other change actions which form cyclic dependencies.

Figure 3.3 shows the steps that the orderer undertakes to determine a valid, aggregated order of change actions. It starts by modelling the change actions provided by the diff as nodes in a graph. Then, in step **(i)**, dependencies among the change actions are determined. In the base implementation, this is done by an algorithm that inspects composite change actions for `addRelationship` atoms, that connect a concept that is in another composite change action, that "reach over", in a manner of speaking. If such instances are found, a dependency is added, modelled as a directed edge, where the target node depends on the source node. Said type of dependency was the only type of dependency among change actions that was encountered during testing. However, since the approach is very generic, this part can be extended with further heuristics to determine other potential dependencies, if required.

Step **(ii)** consist of finding cycles in the graph generated by the previous step, or *dependency clusters*. A distinction is to be made between simple cycles, which are a closed path along edges and nodes with *no repeating nodes*, except the start-end node, and complex cycles which *may have repeating nodes*. Consider the constellation in Figure 3.4. They form two simple cycles, namely $S_{simple,1} = \{A, B, C\}$ and $S_{simple,2} = \{B, D\}$, but only one complex cycle $S_{complex} = \{A, B, C, D\}$. For the purpose of dependency aggregation, we are interested in complex cycles. To stay with the example, we would want to aggregate A, B, C and D into a single `AggregateChanges` change action.

As an implementation detail, the graph library used (*JGraphT*[1]), otherwise very potent, unfortunately only supports detection of simple cycles. Hence, an additional step is required to determine complex cycles. For any set of change actions which form a complex cycle, there are actually infinite possible paths that represent a

[1] https://jgrapht.org/

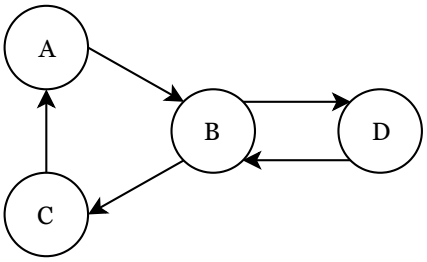

Figure 3.4: Simple vs. Complex Cycles

complex cycle, since the path can just continue to repeat itself. Thus, more precisely, the sets of change actions that are part of the same complex cycle are required — the specific path along this cycle is irrelevant for our purposes. Equally, we are only interested in the set of nodes that are part of a simple cycles, not the path itself.

The observation that the path and thus the order in which the nodes are walked does not matter is important. Instead, simple cycles are looked at as sets of nodes. Combined with the realization, that *dependency clusters* consist of simple cycles that overlap, which in this context means to have *shared nodes*, we can formulate an algorithm to collect the simple cycle sets into *dependency clusters*: Given all sets of nodes that form a simple cycle, repeatedly merge two overlapping sets — replace the two sets by their union – until no overlapping sets remain. The resulting sets are the *dependency clusters*.

Finally, as step **(iii)**, all dependency clusters are aggregated into an `Aggregate-Changes` change action containing the original change actions. In Figure 3.3 these would be change actions *C*, *D* and *F*. Note that edges coming into or leading out of the cluster must be maintained and, if the merge of nodes leads to two edges pointing from the same source node to the same target node, collapsed.

The resulting graph is a *directed acyclic graph*, which always has at least one *topological ordering*. A *topological ordering* or *topological sort* is a ordering of the nodes in a graph, such that if there is an edge (i, j), then $i < j$ (for theoretical details on this, refer to [21]). This is in fact exactly what we require as output of the *Ordering* component: *The Logical Change Action Orderer produces an ordering of the input change actions which matches the topological sort of a graph constructed of the change actions as vertices and dependencies among them as edges, with potential cycles collapsed into aggregate change actions.* Note that this is not possible if a graph has cycles, since if we have edges $(i, j), (j, i)$, a topological sort would imply that $i < j \wedge j < i$ which is obviously false.

3.4 Applying Implementation

The final *Applying* component is a collection of *apply functions* for all change actions that the *Diff Calculator* supports. Strictly speaking, the set of change actions is determined by the *Ordering* component, as it is in our case, because the *Logical Change Action Orderer* introduces an aggregation change action, but for simplicity's sake I will refer to it as the COntoDiff change actions. The purpose of this part

of IncVer is to take an initial ontology version — the old input version — and "evolve" it by incrementally applying the ordered change actions produced by the *Ordering* component, thus creating the desired incremental versions until the new input version is reached.

3.4.1 Implementation Details

No software artifact of a certain size can created in reasonable time without the use of external libraries, which handle out-of-scope tasks. Two notable libraries are used extensively throughout the code and had influence on some of the design of IncVer. They will be briefly described in the following paragraphs.

OWL API[2]: The OWL API was presented first by Horridge and Bechhofer [10] and developed as part of the *Wonderweb Project*[3]. It is one of the most commonly used APIs for working with OWL 2 and is implemented in Java. It consists of the API, an in-memory reference implementation of the API and parsing and rendering capabilities for the most common serialization formats for OWL, such as RDF/XML, OWL/XML, OWL Functional Syntax or Turtle, among others. In addition it provides a reasoner interface for external OWL reasoners, which is supported by many reasoners, such as FaCT++[4] or HermiT[5]. The current version as of this writing is version 5 but its adoption in various libraries is not yet complete, as in the ROBOT library discussed next. IncVer thus uses version 4.

The OWL API will be used as an internal representation of the ontologies. Consequently, all apply functions have been written to operate on `OWLOntology` objects, as provided by the API. The reason for this choice is twofold: Firstly, using the highly expressive OWL ontology format as a basis facilitates extending the pipeline to other ontology formats than OBO. Secondly, the ROBOT library, explained below, also uses OWL internally.

ROBOT OBO Library[6]: ROBOT is both a command line tool for working with OBO ontologies, as well as a programming library which supports manipulating said ontologies. The tool provides functions such as converting between various formats like OBO, RDF/XML and OWL Functional Syntax, or generating simple, axiom-based diffs between two ontologies, both of which were used heavily during development and testing. The main use for IncVer lies in its capability to parse and render OBO ontologies. As mentioned above, ROBOT uses the OWL API internally to represent ontologies. This means that it converts the OBO format to OWL, according to similar rules as the ones presented by Tirmizi et al. [22].

3.4.2 General Apply Functions

In its generic form, an *apply function* takes an ontology and a change action as input, and applies that change action to the ontology. The apply function may assume that the ontology is in a consistent state such that the change can be applied, i.e.

[2] *https://github.com/owlcs/owlapi*
[3] *http://www.cs.ox.ac.uk/ian.horrocks/Projects/wonderweb.html*
[4] *http://owl.cs.manchester.ac.uk/tools/fact/*
[5] *http://www.hermit-reasoner.com/*
[6] *http://robot.obolibrary.org/*

it is not supposed to add an attribute to a concept that does not exist — it is the responsibility of the *Ordering* component to ensure a valid ordering of change actions.

As mentioned before, for IncVer an apply function for every change action generated by COntoDiff, as well as the `AggregateChanges` introduced by the *Logical Change Action Orderer* has been implemented. These can be split into low-level and high-level, analogous to the change actions themselves. The low-level apply functions apply the corresponding low-level change actions (*add/del/map C/A/R*) to the ontology accordingly.

Many of the high-level apply functions operate differently. Seeing as high-level or composite change actions are basically a collection of low-level change actions, they can simply be applied by flattening them to their low-level change actions, ordering these according to the atomic change action ordering discussed in Section 3.3.1, and delegating the application of the low-level change actions to the low-level apply functions. This also handles the challenge, that some composite change action types can contain arbitrarily many (or few, down to zero) change actions of different types, atomic and composite. `delLeaf` for instance may contain just a `delC` and one `delR` change action, but also could contain multiple `delA`s, as well as `delR` change actions. Furthermore the low-level apply functions need not know whether they are part of a high-level change action or details of its implementation.

This covers the general pattern of how the apply functions are implemented, and there is little else to say on their implementation. However, three apply function implementations are worthy of some spotlight:

MapC: The `mapC` apply function is special in that it may have to delete a concept, add a concept, or do nothing at all. The `mapC` change action can occur in `merge` and `split` actions. Because of this, three cases are possible for a mapping of concept `A` to concept `B`: **(1)** `A` equals `B`, i.e. both concepts are the same and `mapC` needs to do nothing, **(2)** the *merge* case, where `A` is not in the new version, being merged into `B` and `mapC` must remove `A`, and finally **(3)** the *split* case, where `B` is created from a split of `A` and must be added by `mapC`.

However, only the first case can be detected from within the `mapC` change action; apply functions do not know whether they are called alone or as part of a high-level apply function (like `merge/split`). To solve this issue, a workaround has been implemented. The determining factor whether a concept in a map operation is to be added or removed is whether it is also mapped *from* or *to*, respectively. That is, given `mapC(A,B)`, if and only if `B` is also mapped *from*, it need not be added. Otherwise it has to be added. Analogous reasoning holds for A and its removal. With this knowledge, two lookup sets are created by iterating through all `mapC` change actions generated by the diff. One set holds all concepts that are mapped *to*, and one holds the concepts that are mapped *from*. Using this set-lookup strategy, we achieve linear complexity $O(n)$ with n equals the number of `mapC` actions, since we only have to iterate through them once, and looking up values in a set has $O(1)$ complexity.

Attributes: This concerns the `add-`, `map-` and, to a lesser degree, the `delete-Attribute` apply functions. The issue here goes back to the attribute parsers and change actions implemented in COntoDiff and has already been touched upon in

Section 2.3.1. Firstly, *synonym scope* is not stored in the change actions, the apply functions only know to add a new *synonym* attribute with its value, but not what scope the synonym has. Similarly, *dbxrefs* are not stored either. To avoid having to rewrite parts of COntoDiff, in order for it to pass on these values, the apply functions instead look up the attributes in the *new input version*.

addRelationship: The noteworthy point here is not the addition of a relationship itself, but the fact that COntoDiff does not parse *typedefs*. This has received mention in Section 2.3.1 along with an approach to solving this. However, the issue here is a larger one and warrants some discussion: *typedefs* are a part of an OBO ontology and ignoring them is no minor omission. While relationships are covered on the instance level and thus, one could implicitly infer typedefs from whether a relationship occurs in the ontology or not, a lot of schema information, such as *domain*, *range* or *hierarchy* are not retained. Moreover, there are ontologies which contain typedefs but no corresponding relationship instances, so this implicit approach fails.

Unfortunately, this problem is embedded in *COntoDiff* and large changes to it would have to be made to amend this. Though no analysis has been conducted on this, it stands to reason that, similar to the other elements in the CAR model, a typedef can be *added*, *deleted* or *mapped*. In addition, typedefs are closely connected to their relationship instances: for a valid ontology, relationships should only occur when a corresponding typedef is defined. The inverse does not hold, as mentioned before — an OBO ontology can have a typedef but no instance of it.

From this, we can deduce some relations between operations on typedefs and operations on relationship instances: Relationship instances can be added at the earliest at the same time as adding the corresponding typedef, not before that. Likewise, relationship instances must be removed at the latest at the same time as their typedef is deleted. Finally, when a typedef is mapped, *all* its relationship instances must be changed accordingly.

It is apparent that without tracking the typedefs, these connections cannot be captured. Instead, a compromise solution was built, as implementing such a system would have easily exceeded the scope of this thesis. I implemented a simple component that parses typedefs of the old and new input ontology versions and then calculates a very basic diff. This diff tracks *additions*, *deletions* and *mappings*. The mapping part is however very basic, matching typedefs with the same *id* are matched. There is no facility to capture more sophisticated matchings based on patterns, for cases where the *id* is changed. Relating this all back to the `addRelationship` apply function: this simple diff is stored for lookup, and whenever it is called, `addRelationship` checks if the relationship has a corresponding typedef that is newly added, and if it is the first time an instance of that relationship is added, the typedef is added as well. Additionally, as a final step after all change actions have been applied, the remaining typedef changes (unapplied additions, removals and mappings) are applied. This approach takes inspiration from the connections made in the previous paragraph.

3.5 Implementation Pipeline and Providers

Finally, all these parts must be combined. This is what the *pipeline* is for. It is responsible for taking the input versions, feeding them to the *Diff Calculator*, passing the result on to the *Ordering* component, and finally coordinating the *Applying* component to iteratively apply change actions and generating incremental versions. Additionally, the pipeline takes care of initializing the components and providers, which will be elaborated on in the next section.

3.5.1 Global Data Providers

As has been explained in previous sections, there are some parts within the pipeline that depend on information from a different part — for instance the `mapC` apply function, which needs access to the *mapped-to* and *-from* concepts, which is calculated on the diff generated by the diff calculator. For this purpose, several so-called *providers* have been implemented, which calculate and subsequently provide certain information. A listing of these will follow, along with a brief explanation of what their specific task is.

DiffDataProvider: Provides global access to data concerning the diff that was calculated between the old and new input versions. It provides read-access to basic and complex change actions, but also some lookup functionality, used in apply functions, like whether a concept is mapped to or from (see Section 3.4.2).

TypedefTracker: Is responsible for tracking which typedefs have been and are yet to be added to, removed from or modified in the ontology. Whenever the `addRelationship` apply function adds a relationship, it checks with the `TypedefTracker` whether this relationship belongs to a new typedef and is being added the first time, in which case the apply function will also add the typedef to the ontology, along with the relationship.

RobotOWLData & OWLUtil: Provide convenience functions and lookup for data and operations relevant to ROBOT and OWL respectively. Foremost, they provide full IRIs used in the OWL representation. The IRI for concepts and typedefs depends on the name of the input ontology, and thus can only be determined after it has been parsed. Furthermore, many OBO tags are denoted by specific IRIs, not all following the same pattern. Hence `RobotOWLData` provides a build-function for these. Finally, several convenience functions can generate various complex OWL axoims which are used to represent OBO constructs and would be tedious to be constructed manually in every apply function.

3.5.2 Command Line Interface

To make IncVer more readily usable, a command line interface (CLI) was implemented. The distributed Jar supports two commands, `pipeline` and `validate`. The former executes the pipeline as it is described in this chapter. The latter is a utility for evaluating the results generated by IncVer. It has been used for the evaluation in this thesis, which will be presented in-depth in Chapter 4 — this section shall only briefly demonstrate the usage of the command line tool.

Long	Short	Parameters	Description
`--save`	`-s`	`FILE`	Saves the calculated diff in serialized format to `FILE`
`--load`	`-l`	`FILE`	Loads a previously serialized diff from `FILE` instead of calculating it
`--prefix`	`-p`	`PREFIX`	Set the the prefix of generated incremental versions, resulting in the pattern `PREFIX.n.obo`, where `n` is the version number. Defaults to `incver.n.obo`
`--nocheck`	`-n`		Set this flag to disable the storage requirements check
`--diffonly`	`-d`		Diff only mode. Calculates diff without generating incremental versions. Requires the `-s` or `--save` flag

Table 3.1: Available Options and Flags for the Pipeline CLI Command

Pipeline: This command allows the execution of the main IncVer pipeline, which can be called by providing two input versions as well as specifying an output folder. An example call would be:

```
$ java -jar incver.jar pipeline old.obo new.obo incver-output/
```

This call would execute the `pipeline` on `old.obo` and `new.obo` as the old and new input versions respectively. The generated incremental versions would be saved into the folder `iv-output/`. By default, the pipeline calculates an estimate of how much disk space the resulting versions will approximately take up and the user must confirm to proceed with the generation, but this check can be disabled. Additionally, several flags and options, of the form `--flag` and `--option=param` respectively, can be specified to customize the execution. Table 3.5.2 shows the available options in detail.

Validate: With the `validate` command, the generated versions can be evaluated. This is done by applying three conditions, discussion of which will be reserved for Chapter 4. A sample call would be:

```
$ java -jar incver.jar validate old.obo new.obo incver-output/
```

As can be seen, the same arguments as with the `validate` call are passed, this time with a populated output folder containing incremental versions. In addition, the two options `--only`/`-o` and `--skip`/`-s` can be specified, taking the numbers of the conditions that are to be applied or skipped respectively as arguments. A valid parameter would be `--only=13`, which would only execute conditions 1 and 3. By default, the `validate` command executes all three conditions.

4

Evaluation and Discussion

While the main goal of this thesis is the realisation of a functioning and usable base implementation, an evaluation is of course in order to assert that the software does what it is supposed to do. I was able to receive promising results evaluating IncVer with multiple datasets and metrics.

This chapter will start by laying out how IncVer is evaluated, introducing three evaluation conditions. Following that, I will present some results: As part of the evaluation, five sample ontologies were selected and processed by IncVer. The resulting incremental versions were then subjected to said evaluation metrics. Finally, the chapter concludes by discussing the results and remarking on limitations and future work.

4.1 Evaluation Metrics

It is not the explicit goal of IncVer to correctly model the evolution history of two ontology versions, but to produce *valid* incremental versions. Valid in this context means that the incremental versions *could have reasonably* existed. Indeed, there is no way to determine whether the generated incremental versions and their order correspond to the actual evolution of the ontology without additional versioning info, such as edit logs. While the base implementation of IncVer does not support such functionality, it might be implemented in the future by ways of a custom *Orderer* component.

Three *conditions* were thus formulated which need to hold for the result of running IncVer to be considered valid. The conditions look at ontologies as collections of *OWL axioms*, which can colloquially be understood as *statements*, such as *"SO:0001 is a class"*, *"X has name Peter"* or *"parent_of is the inverse of child_of"*. The first two conditions check the incremental versions as a whole, in a sense, while the third condition analyses the individual steps between the incremental versions.

The conditions were formulated as strict logical statements that need to hold over the entirety of the output produced. Such an all-or-nothing approach is much too coarse for helpful analysis, since a single violation in thousands of axioms would result in a failure, without further information. Thus, for meaningful evaluation the conditions were converted to metrics that yield a numeric breakdown of each condition.

4.1.1 End-To-End Conditions

For old and new input versions O_{old}, O_{new} respectively and incremental versions $O_1, O_2, ...O_n$ with n = number of Change Actions:

$$\forall i : a \in O_i \implies a \in O_{old} \cup O_{new} \tag{4.1}$$

$$O_n = O_{new} \tag{4.2}$$

No Unknown Axioms: Condition (4.1) states that every axiom a in each incremental version O_i must also be in the old or the new input version. In essence, this condition ensures that *no unknown axioms* are introduced to incremental versions. It is also worth pointing out, that this condition may not reflect the actual development between the ontology versions, since it is easily possible that during its evolution an axiom is added and then removed again. This case cannot be captured by COntoDiff or the base implementation of IncVer.

As a metric, all axioms of each incremental version are checked to see if they are contained in the union of O_{old} and O_{new}. As a further layer of granularity, *containment* is split up into three cases, instead of the two *contained/not contained*. These are *full match*, *partial match* and *unknown*. A full match occurs when exactly the same axiom is in the incremental version as well as in one or both of the input versions. A partial match is a slight relaxation of the former case; it occurs when axioms with the same values, such as *subject* or *attribute name* are in the incremental version as well as in the input versions, but annotation values such as *dbxrefs* differ. This case is mainly motivated by the limited attribute parsing of COntoDiff as discussed in Section 2.2.3. If none of the previous two is the case, the axiom is classified as an unknown. As a final point: Since many axioms will be in every version, unchanged, each unique axiom is only considered once, as not to inflate the numbers.

Last Version Equals New Version: Condition (4.2) states that the final incremental version that is generated equals the new input version, that is, consists of the same axioms. The purpose of this condition is twofold: Firstly and obviously, with the final version, all change actions are applied and we should "land on" the new input version. Metaphorically speaking, this condition checks whether the path taken (the incremental versions) leads to the intended destination (the new input version). Secondly, and perhaps more importantly, this condition is the complement to (4.1), in that it ensures that no axioms are *missing*.

Converted into a metric, the final version O_n is compared with the new input version O_{new}. Here I differentiate between four cases: *full match*, *partial match*, *unknown* and *missing*. Full and partial matches work analogously to the previous condition. Unknown axioms are those that are in O_n but not in O_{new}, and missing axioms are the inverse case, axioms that are in O_{new} but not in O_n. As can be seen, the naming is chosen in relation to the expected, i.e. the new input version. The measurement taken is a count of all four cases and since there are only two versions compared, we need not consider duplicate axioms.

It is also worth pointing out, that *unknown* axioms generally are not expected to be wholly unknown axioms with previously unseen values, but incorrectly or incompletely parsed attribute values, strings or annotations. Similarly, *missing* axioms

are likely not missing completely but similar axioms with incorrect values. This is because even one wrong character in a definition string, for instance, would lead to the axiom not being recognized. Moreover, it is important to distinguish between unknowns in *Condition 1*, which are axioms that are unknown to the *old and new* input ontologies, and unknowns in *Condition 2*, which are axioms that are unknown only to the *new* input ontology.

An additional condition was considered, stating that every generated incremental version must be valid. Valid in this context means structurally valid. A violation would be for example an attribute being in the ontology without the concept it belongs to. However, after some investigation it became clear that, firstly, most of such violations are not possible in OBO — for instance, an attribute can only be written in its corresponding term stanza — and secondly, the `ROBOT` library only produces valid version — if a formatting error is present or an invalid value, then attempting to save the ontology produces an exception. Hence, a metric based on this condition would not be very meaningful.

4.1.2 Increment Step Condition

For a given sequence of change actions $C_1, C_2, ...C_n$, the resulting incremental versions $O_1, O_2, ...O_n$ and the change action set resulting from applying COntoDiff to two versions $CA_{i,j} = \texttt{COntoDiff}(O_i, O_j)$ require that:

$$CA_{i-1,i} = \{C_i\} \tag{4.3}$$

The third condition (4.3) states that when calculating the diff between two *consecutive* incremental versions O_{i-1} and O_i, the resulting diff must be a set only containing *one* change action, that action being the same as that which was applied to generate the second version. Essentially, this condition ensures that each subsequent version changed exactly by the change action that was applied to it, or simply, that the apply functions apply the correct changes and are applied correctly. In order to be able to do this after the fact an *incremental version log* is required, which keeps track of the change actions that were applied to each version, or potential errors that occurred when applying.

Hence, the IncVer pipeline keeps a log while generating the incremental versions and writes them to a file after applying all the changes. The log maps version file names to the change actions that were applied to generate that version, or error messages if an exception occurred while applying the change or writing it to file.

For the metric, a diff is calculated with COntoDiff, between every two consecutive versions of the generated incremental versions. The resulting compact diff is compared with the expected change action recorded in the version log. A distinction is made between six different cases:

Exact The diff produces one change action that is identical to the expected change action. This is the desired result.

Type The diff produces one change action that is of the same type (e.g. *delAttr*) as the expected change action but not exactly identical, differing in a value, for instance.

Dataset	Versions	Old Size	New Size	Versions	LLCA
Sequence types and features Ontology (SO)	2010-7 to 2010-11	17 110	17 796	404	962
Phenotype and Trait Ontology (PATO)	2018-3 to 2018-11	21 890	21 748	329	455
Plant Ontology	2017-2 to 2018-9	23 150	23 544	273	865
Human Disease Ontology (Non-Classified)	2019-3 to 2019-4	136 232	136 421	609	1070
PROtein Ontology (Non-Reasoned)	2013-2 (v32) to 2013-3 (v33)	364 250	365 024	290	1238

Table 4.1: Evaluation datasets, the old and new versions used, as well as their respective size in number of axioms. Additonally, the number of versions generated and low level change actions computed by COntoDiff is listed.

Unknown The diff produces one change action but it is of a different type than the expected change action.

Exact Contained The diff produces multiple change actions, one of them is identical to the expected change actions.

Type Contained The diff produces multiple change actions, at least one of them is the same type as the expected change action but none is identical.

Not Contained The diff produces multiple change actions, all of which are of a different type than the expected change action.

Furthermore, a note is made if an error occurred during the creation of an incremental versions, since those results are almost certain to produce an error.

4.2 Results

This section will briefly introduce the datasets used for evaluation. Then, the results of the evaluation will be presented, along with an explanation of how the evaluation was conducted.

4.2.1 Datasets

For the purpose of evaluation, five datasets ([4, 7, 3, 20, 15]) were selected. The only requirements for these datasets were that **(1)** they must be in the OBO format, and **(2)** there must exist at least two versions of it. Apart from that, an effort was made to select ontologies of differing sizes and sources.

Table 4.1 lists the datasets along with some characteristics. The size is measured in OWL axiom count and is a good indicator for file size. It is however only a very rough indicator for number of changes between versions. Hence, column *Versions* lists the number of incremental versions generated and column *LLCA* lists the number of low-level change actions calculated by COntoDiff. Low-level change actions were chosen as a more precise indicator for degree of change compared to high-level change action count, since that number can be misleading: one HLCA could be a `addSubGraph` with tens of component changes or just a single `delA`. For the number of high level change actions, please refer to Table 4.6.

As can be seen, the datasets range from relatively small, like the *SO* with only 17 to 18 000 axioms, to an order of magnitude larger with the *Human Disease Ontology* and *PROtein Ontology* consisting of well over 100 000 and 300 000 axioms respectively. While testing, filesize was a another consideration: In combination with a high number of incremental versions, the resulting total size of all incremental versions can quickly grow large. In this regard, the result of the *Human Disease Ontology* took up the most disk space, reaching 2.84 GB in size, and *PATO* was the smallest with only 212 MB.

Finally, the rate of change varies strongly as well. I will measure rate of change as $\frac{\#\text{ LLCA}}{Time}$. The slowest to change of our datasets is the *Plant Ontology* with $\frac{865\text{ LLCA}}{19\text{ months}}$ or approximately $45.5\frac{LLCA}{month}$. On the other side of the spectrum, there is the *PROtein Ontology*, with 1238 change actions in a single month.

4.2.2 Condition 1

Table 4.2 lists the results of evaluating the datasets with *Condition 1*. For the most part the results are satisfying, with only few unknown axioms produced. *SO* yields only one partial match and one unknown axiom, and the *Plant* and *Human Disease* ontologies include no unknown axioms at all. Fewer errors on *SO* are to be expected, since that dataset was used during development. As such it is even more surprising that the results on the *Plant Ontology* and *HDO* are slightly better.

It is pertinent to compare the number of errors to the number of low level change action, for an impression on how well IncVer performs, since many axioms in an

Dataset	Full	Partial	Unknown	Total	LLCA
SO	17 918	1	1	17 920	962
PATO	22 018	0	0	22 018	455
Plant	23 684	0	12	23 696	865
HDO	136 673	0	0	136 673	1070
PROtein	365 011	0	50	365 061	1238
Total	565 304	1	63	560 368	4590

Table 4.2: Results of Evaluation with Condition 1, featuring the counts of Full and Partial matches, Unknown axioms and total count, along with the number of low-level change actions between old and new version

ontology are completely unchanged. In that regard, the performance is best on the *HDO* with 0 errors on 1070 low-level change actions, and the worst on *PRO* with $\frac{50 \text{ errors}}{1238 \text{ LLCA}}$ or $0.040^{errors}/_{LLCA}$.

Surprisingly, no partial matches occurred except a single one in *SO*. This indicates that the measures taken to address the issue of the restricted parsing capabilities in COntoDiff were somewhat effective, if not perfect as we will see in the next section.

4.2.3 Condition 2

The results of imposing *Condition 2* to the datasets can be seen in Table 4.3. It is readily apparent that more errors occurred for this condition, but it could be argued that it is also a stricter condition, since matches are only sought in the new input version. This captures additional issues, like an attribute not being deleted or a concept not being correctly mapped. Nevertheless, the number of errors is relatively small, with *PATO* again achieving a perfect performance.

Table 4.4 shows the error rates per dataset in $\frac{errors}{LLCA}$, calculated by summing the errors, once including and once without partial matches, and then dividing the sum

Dataset	Full	Partial	Unknown	Missed	Total	LLCA
SO	17 787	6	1	3	17 797	962
PATO	21 748	0	0	0	21 748	455
Plant	23 326	138	81	80	23 625	865
HDO	136 409	5	42	7	136 463	1070
PROtein	364 960	5	61	59	365 085	1238
Total	564 230	154	185	149	564 718	4590

Table 4.3: Evaluation Results for Condition 2 including the counts of Full and Partial matches, Unknown and Missed axioms and total count, along with the number of low-level change actions between old and new version

Dataset	w. Partial	w.o. Partial	Axioms	LLCA
SO	0.010	0.004	17 797	962
PATO	0	0	21 748	455
Plant	0.346	0.186	23 625	865
HDO	0.050	0.046	136 463	1070
PROtein	0.101	0.097	365 085	1238
Overall	0.106	0.073	564 718	4590

Table 4.4: Error Rates in Errors per LLCA for Condition 2, both including and excluding Partial Match cases as errors, along with the total number of axioms in the final generated versions and the count of low level change actions between old and new input version, detected by COntoDiff

Dataset	Exact	Type	Unkn.	Exact C	Type C	Not C	Aggr.	Total
SO	359	1	41	0	0	4	2	405
PATO	330	0	0	0	0	0	0	330
Plant	0	0	0	229	23	7	1	259
HDO	0	0	0	397	208	3	0	608
PROtein	166	4	22	0	0	1	0	193
Total	855	5	65	626	231	15	3	1795

Table 4.5: The results of applying Condition 3 to every consecutive pair of incremental versions, listing the counts of Exact, Type and Unknown cases, as well as Exact Contained, Type Contained and Not Contained. Additionally, the number of Aggregate change actions and the total number of compared pairs are listed.

by the number of low level change actions. This shows the *Plant Ontology* to produce the worst result, with a particularly high count of partial matches. This may indicate a prevalent use of annotations in that specific ontology. IncVer performs considerably better on the other datasets, with *PROtein* ranging at roughly 0.1 $^{errors}/_{LLCA}$ and the remaining ontologies having an error rate of 0.5 and below, and with an overall error rate of approximately 0.1 $^{errors}/_{LLCA}$ when counting partial matches and 0.73 when excluding them.

It is also important to point out that *unknown* and *missed* axioms are not independent of each other but are in many cases related. If, for instance, IncVer fails to correctly map an attribute, this will produce an unknown axiom, since the unchanged attribute is not in the new version, as well as a missing axiom, since the final version does not contain the correctly changed attribute. This may explain why the numbers of unknown and missed axioms lie very close to each other for four out of five datasets.

Finally, not all of the errors produced are caused by IncVer failing to apply change actions correctly. As mentioned previously, COntoDiff has only very limited parsing capabilities, and while part of this issue has been addressed by extending the parser, some limitations still apply. Changes to annotations for instance are completely ignored and other, multi-valued tags are only parsed partially. Given incomplete information, it is obvious that IncVer cannot fully reconstruct the target ontology version even with its best-effort approach.

4.2.4 Condition 3

The results of the evaluation with Condition 3 are presented in Table 4.5. The columns are counts according to the distinctions made earlier in Section 4.1.2, representing *Exact*, *Type* and *Unknown* cases, as well as *Exact Contained*, *Type Contained* and *Not Contained* cases, in that order. In addition, the number of *aggregate* change actions that were generated while producing the incremental versions and the total number of *generated* versions is listed. Both warrant some explanation.

COntoDiff does not produce *aggregate* change actions — they are introduces

Dataset	HLCA	Errored Versions
SO	405	0
PATO	330	0
Plant	275	16
HDO	610	2
PROtein	291	98
Total	1911	116

Table 4.6: Count of High Level Change Actions for every dataset, along with the number of incremental versions that produced an error.

by the *Logical Change Action Orderer* by merging dependency cycles, as detailed in Section 3.3.3. As such, running COntoDiff on the incremental versions will not produce an *aggregate* change actions but the individual actions that make up that *aggregate* change action and will appear as a *Not Contained* case in the evaluation. For the sake of correctness, the resulting counts are kept in the table, but the aggregate cases have all been inspected and the resulting change actions are indeed the expected ones. Thus, the aggregate count is listed as well, and can be deducted from number of *Not Contained* cases.

Additionally, Table 4.6 lists the high-level change action count *after aggregation* along with the number of incremental versions that produced an error applying the change and were subsequently not written to file. This is relevant to the data in Table 4.5 since that data is only calculated over the successfully generated versions. It is also an additional measure of performance, directly measuring the execution error rate of the apply functions. Inspection of the errors for the *PROtein Ontology* show that most of the errors are due to `chgAttValue` apply function not being able to find an axiom in the new input version involving an *xref* attribute.

4.3 Discussion

Condition 1 aims to assess the degree to which unknown or wrong axioms are introduced to the incremental versions. As expected, the numbers are low with only $0.014^{errors}/_{LLCA}$ overall. In addition, the unknown axioms are not wholly foreign axioms but values that have been parsed incorrectly or incompletely, or definitions and synonyms, who could not be looked up in the target ontology, again likely due to the limited parsing.

Results for the arguably stricter Condition 2 paint a similar picture, although the number of errors is slightly higher. Overall, approximately $0.07^{errors}/_{LLCA}$ occur if partial matches are accepted, and a slightly higher $0.1^{errors}/_{LLCA}$ if partial matches are considered errors. Two things are worthy of note: Firstly, there is some variance between the different datasets and there seems to be little to no correlation between size of ontology and error rate nor between low-level change action count and error rate, but no deeper statistical analysis has been conducted on this. The higher error count in some datasets may be due to more prevalent usage of annotation values and

less common term tags, which were tested less or not at all. Secondly, the counts for *Unknown* and *Missed* axioms are often related. If, for instance, a concept is mapped incorrectly or an attribute is not deleted, an *Unknown* (to the *new* version) axiom is present and the correct one is missing. Considering this, the results are acceptable, with the exception of the *Plant* ontology which has a noticeably higher rate of errors than the other ontologies, particularly when counting *Partials* as errors. In general, there are no grave errors, but rather axioms that are not quite correct.

The analysis and interpretation of the results of the third condition is less straightforward. For the *SO*, *PATO* and *PROtein Ontology*, the majority of change actions are indeed *exact* matches, showing good performance. While the *Plant* and *Human Disease* ontologies each have 0 exact matches, they have many *Exact Containment* matches, which indicates that the change actions did not completely fail. Indeed, inspection of these *Exact Containment* cases revealed a peculiar pattern: almost all generated diff sets contained the expected change action, but between *every* pair of consecutive versions, COntoDiff also detects several `mapC` change actions. Further inspection is required to fully understand this occurrence, but possible causes for this may lie in the `mapC` implementation or with COntoDiff. In particular, one possible explanation is that COntoDiff detects mappings between unchanged concepts and since these concepts are never modified, they are detected between every pair of ontologies.

Many *Unknowns* can be explained by partial application of *composite change actions*. For instance, the *move* action consists of deleting an old relation and adding a new one in its place. However, if an error occurs in the second step, the old relationship is still deleted, as the composite apply functions for the most part do not first ensure whether all component actions succeed, leading to a *delR* action being detected instead of *move*. I deem it feasible for such consistency-checking behaviour to be implemented if it is desired. For the example at hand: Whether it is preferable for the old axiom to be kept in case of partial errors, or be deleted depends wholly on the use case, ideally the complete composite change action should be applied.

Taken together, the results of all three validations paint a favourable picture. While IncVer is by no means without issues that need yet to be addressed, and which will be elaborated on in the following sections, it shows a lot of promise as a base implementation. In particular, the surprisingly high performance on some ontologies (*SO*, *PATO*, *PROtein*) shows that IncVer generally works well. This assessment is supported further by the fact that many of the errors that occurred can be traced back to one specific issue with the implementation, that can be feasibly addressed.

5

Limitations

This chapter briefly discusses some limitations of this thesis, given by its scope, time constraint as well as those which derive from the results of the evaluation.

While the results presented in Chapter 4 are encouraging and demonstrate that IncVer largely works, there are still some caveats. IncVer has some clearly defined shortcomings that need yet to be addressed, some of which go back to COntoDiff: not all attribute tags are yet supported, others are being only partially or incorrectly parsed. The workaround for some attribute types implemented in IncVer that consist of looking up attributes in the new input version amends this issue somewhat but does not manage to completely solve it.

Closer inspection of the validation results for condition in Section 4.2.4 identify further issues. The high count of errored versions in the *PROtein Ontology* shows that handling of the *xref* attribute is not yet good enough, and warrants a re-examination of the implementation of the `chgAttValue` apply function. Also, the generation of `mapC` change actions between incremental versions for the *Plant* and *Human Disease* ontologies is not fully understood yet and requires further investigation; it is possible, however, that this is not actually an error in IncVer, but instead caused by COntoDiff detecting some unchanged concepts as *mappings*, explaining why these `mapC` actions occur between all versions.

All in all, IncVer is expected to work well for many OBO ontologies, however, perfect results are not to be expected. Manual inspection of the results may be advisable. For automated inspection, the code implementing all three validations conducted in this chapter is also included in the JAR distribution and can be executed via the command line.

Going beyond the evaluation results, there are also some functional limitations. The most prominent one is IncVer being restricted to OBO ontologies. Initially, more broad support for other ontology formats, mainly OWL, was planned. However after learning that COntoDiff only supports OBO, the scope had to be adjusted. The support for *typedefs* is also very limited as has been laid out in Section 2.2.

As a final note on performance: the main bottleneck lies in the calculation of the compact diff in COntoDiff. During testing and evaluation over the datasets, the execution time mainly increased with diff size. Responsible for this seems to be primarily the aggregation step, which repeatedly iterates over the complete diff until no new aggregation is achieved. Obviously, writing more incremental version also takes more time. Despite this, executing IncVer over the datasets resulted in acceptable running times; as a reference value, generating the incremental versions

for the *HDO* which yielded 610 incremental versions taking up approximately 2.84 GB completed in under 5 minutes on an Intel Core i7-8550U business laptop with 16GB RAM and SSD. Calculating diffs and generating incremental version over larger datasets with many changes however will quickly reach infeasible levels of space and time requirements.

6

Future Work

The insights from building IncVer and the subsequent analysis of its results in Chapter 4 provides multiple avenues to pursue for future work. Most obvious is the addressing of the issues mentioned in Section 4.3 , including the re-examination of some apply functions, such as `mapC`, and improvements to the attribute handling (*xrefs* in particular) and annotations.

To inform the corrections and improvements that are required, more detailed validation results might be necessary. The `VERSIONLOG.csv` that is produced when IncVer is run could serve as a solid starting point, as it records exceptions that occur during creation of incremental versions. The validation reports, produced by running the `validate` command could also be updated to, perhaps optionally, provide more detailed information.

Another issue is the rudimentary support for *typedefs*. It should be considered only provisional and if more sophisticated tracking of typedef changes is required, the results of the base IncVer implementation may not be of sufficient quality. This may then require a different *Diff Calculator* than COntoDiff or a majorly revised version, which tracks typedefs.

Shifting from corrections and improvements to extensions: an implementation of pipeline components that support OWL ontologies would be a major contribution to the framework. Many concepts used in the base implementation, such as the dependency graph and cycle merging could be reused in an OWL counterpart.

Finally, an *Ordering* component that takes into account some version history information, such as existing intermediate versions on version control systems or edit history from ontology editors would go a long way in approximating real world ontology evolution.

7

Conclusions

This chapter marks the conclusion to this thesis, summarizing the work done and recapturing my findings and the results that were attained.

With IncVer, I implemented an incremental versioning tool which allows for the creation of detailed version datasets from two input ontology versions. These resulting incremental versions differ only by a few changes between each consecutive pair. Previously, researchers were limited to the releases of ontologies, which often lie apart several months or even longer and usually contain hundreds to thousands of changes between releases. IncVer makes it now possible to analyse change between ontology versions on a much more granular level and it is thus expected to be of particular use in the field of ontology evolution and impact analysis, where it is often desired to have small and detailed changes instead of large swathes of changes, in order to analyse the nature and impact of individual change actions or small groups of related changes.

The modular architecture of IncVer is by design very extensible. Users are able to customize every stage in its pipeline. The first component, the *Diff Calculator*, is responsible computing a diff between two input ontology versions. An updated version of COntoDiff was chosen as base implementation for this component, but many other diffing approaches could conceivable be plugged in. The resulting diff, a set of change actions, is passed on to the *Ordering* component, producing a sorted, possibly transformed sequence of change actions. The base implementation is the *Logical Change Action Orderer*, producing an arbitrary order, avoiding unmet dependencies between change actions. To achieve this, I developed an approach which represents the change actions as a dependency graph, merges cycles and then produces a topologically ordered sequence of change actions. It is possible to implement custom *Orderer* components which can be tailored to specific research requirements, such as modelling the evolution after real world patterns or taking edit histories into account. The final *Applying* component takes the ordered list of change actions and applies them incrementally to the old input version, evolving the ontology until it reaches the state of the new input version.

To ensure correctness, three conditions were imposed on the generated incremental versions, which had to be met for a result of running IncVer to be considered correct. They examine whether unknown axioms were introduced, required axioms were missing and whether the difference between two consecutive versions matches the change action that was supposed to be applied. With these conditions converted to more granular metrics, the incremental versions generated from five different

datasets were evaluated, yielding promising results. It is thus fair to say that this thesis was successful in providing a tool that allows for creation of detailed evolution datasets. This now enables further research on ontology evolution and its effects.

Nonetheless, issues remain; The results of IncVer are not completely error free and performance varies with the ontology used. The cause of most errors could however be narrowed down and at least some possible explanations could be formulated. Because of time constraints, they could not yet be fixed.

IncVer is so far limited to OBO ontologies, dictated by the discovery that COntoDiff in effect only supports the OBO format. Furthermore, it only recognized a fraction of the defined attribute tags, but this could be addressed by extending the OBO parser. A further improvement has been implemented, parsing typedefs separately, computing a very basic diff and adding new typedefs whenever the corresponding attribute is added for the first time or at the end.

The extensible nature of IncVer makes it a very flexible and promising tool for the area of ontology evolution and change analysis. Apart from addressing remaining issues, a potentially very useful extension would be the implementation of pipeline components that support the OWL format. Given the high expressivity of OWL and that it is possible to translate many ontology formats, including OBO, to OWL, this would go a long way in making the approach of IncVer usable on a large number of ontologies. Other improvements such as more sophisticated support for OBO typedefs or including historical information when ordering change actions would further increase the usefulness of IncVer.

References

[1] OWL 2 web ontology language document overview (second edition). Tech. rep., W3C, Dec 2012. *http://www.w3.org/TR/2012/REC-owl2-overview-20121211/*. Retrieved June 6, 2019.

[2] Berners-Lee, T., Hendler, J., Lassila, O., et al. The semantic web. *Scientific american 284*, 5 (2001), 28–37.

[3] Cooper, L., Walls, R. L., Elser, J., Gandolfo, M. A., Stevenson, D. W., Smith, B., Preece, J., Athreya, B., Mungall, C. J., Rensing, S., et al. The plant ontology as a tool for comparative plant anatomy and genomic analyses. *Plant and Cell Physiology 54*, 2 (2012), e1–e1.

[4] Eilbeck, K., Lewis, S. E., Mungall, C. J., Yandell, M., Stein, L., Durbin, R., and Ashburner, M. The sequence ontology: a tool for the unification of genome annotations. *Genome Biology 6*, 5 (Apr 2005), R44.

[5] Fernández, J. D., Schneider, P., and Umbrich, J. The dbpedia wayback machine. In *Proceedings of the 11th International Conference on Semantic Systems* (2015), ACM, pp. 192–195.

[6] Frommhold, M., Piris, R. N., Arndt, N., Tramp, S., Petersen, N., and Martin, M. Towards versioning of arbitrary RDF data. In *Proceedings of the 12th International Conference on Semantic Systems - SEMANTiCS 2016*, ACM Press, pp. 33–40.

[7] Gkoutos, G. Phenotype and trait ontology. *http://purl.obolibrary.org/obo/pato/releases/2018-03-28/pato.owl*.

[8] Gross, A., Hartung, M., Prüfer, K., Kelso, J., and Rahm, E. Impact of ontology evolution on functional analyses. *Bioinformatics 28*, 20 (2012), 2671–2677.

[9] Hartung, M., Gross, A., and Rahm, E. Conto–diff: generation of complex evolution mappings for life science ontologies. *Journal of biomedical informatics 46*, 1 (2013), 15–32.

[10] Horridge, M., and Bechhofer, S. The owl api: A java api for owl ontologies. *Semantic Web 2*, 1 (2011), 11–21.

[11] John, D.-R. The obo flat file format specification, version 1.2. *https://owlcollab.github.io/oboformat/doc/GO.format.obo-1_2.html*. Retrieved June 6, 2019.

[12] Khattak, A. M., Batool, R., Pervez, Z., Khan, A. M., and Lee, S. Ontology evolution and challenges. *J. Inf. Sci. Eng. 29*, 5 (2013), 851–871.

[13] Klein, M. C., and Fensel, D. Ontology versioning on the semantic web. In *SWWS* (2001), pp. 75–91.

[14] Kontchakov, R., Wolter, F., and Zakharyaschev, M. Can you tell the difference between dl-lite ontologies?. In *KR* (2008), pp. 285–295.

[15] Natale, D. A., Arighi, C. N., Blake, J. A., Bult, C. J., Christie, K. R., Cowart, J., D'Eustachio, P., Diehl, A. D., Drabkin, H. J., Helfer, O., et al. Protein ontology: a controlled structured network of protein entities. *Nucleic acids research 42*, D1 (2013), D415–D421.

[16] Noy, N. F., Musen, M. A., et al. Promptdiff: A fixed-point algorithm for comparing ontology versions. *AAAI/IAAI 2002* (2002), 744–750.

[17] Papakonstantinou, V., Fundulaki, I., and Flouris, G. Assessing linked data versioning systems: The semantic publishing versioning benchmark. *Emerging Topics in Semantic Technologies: ISWC 2018 Satellite Events 36* (2018), 219.

[18] Papavassiliou, V., Flouris, G., Fundulaki, I., Kotzinos, D., and Christophides, V. On detecting high-level changes in rdf/s kbs. In *International Semantic Web Conference* (2009), Springer, pp. 473–488.

[19] Pernischova, R. Impact of Changes on Operation over Knowledge Graphs. *Unpublished Master Thesis*, (Aug 2018)

[20] Schriml, L. M., Arze, C., Nadendla, S., Chang, Y.-W. W., Mazaitis, M., Felix, V., Feng, G., and Kibbe, W. A. Disease ontology: a backbone for disease semantic integration. *Nucleic acids research 40*, D1 (2011), D940–D946.

[21] Thulasiraman, K., and Swamy, M. *Graphs: Theory and Algorithms*. A Wiley interscience publication. Wiley, 1992.

[22] Tirmizi, S. H., Aitken, S., Moreira, D. A., Mungall, C., Sequeda, J., Shah, N. H., and Miranker, D. P. Mapping between the obo and owl ontology languages. *Journal of biomedical semantics 2*, 1 (2011), S3.

[23] Tury, M., and Bieliková, M. An approach to detection ontology changes. In *Workshop proceedings of the sixth international conference on Web engineering* (2006), ACM, p. 14.

[24] Völkel, M., and Groza, T. Semversion: An rdf-based ontology versioning system. In *Proceedings of the IADIS international conference WWW/Internet* (2006), vol. 2006, p. 44.

[25] Zablith, F., Antoniou, G., d'Aquin, M., Flouris, G., Kondylakis, H., Motta, E., Plexousakis, D., and Sabou, M. Ontology evolution: a process-centric survey. (2015), 45–75.

A

Appendix

A.1 Contents of the CD

- "Abstract.txt", an unformatted text file containing the abstract
- "Zusfsg.txt", an unformatted text file containg a German summary
- "Masterarbeit.pdf", a PDF file of this master thesis
- Executable Jar Distribution of IncVer
- Configuration files "ChangeActions.xml" and "Rule_OBO.xml"
- Java source code for IncVer
- Development documentation for IncVer
- Collection of synthetic datasets for testing IncVer
- Python script for out-of-order, line-based file comparison
- Ontologies used for evaluation
- CSV and text files containing evaluation results for evaluation ontologies
- LaTeX files of this thesis

List of Figures

List of Tables