

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GRAPHICAL USER INTERFACE FOR COMPARING RDF MODELS

DIPLOMOVÁ PRÁCE

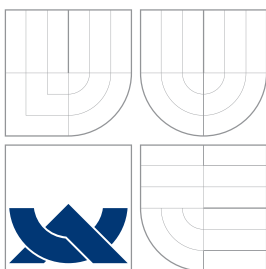
MASTER'S THESIS

AUTOR PRÁCE

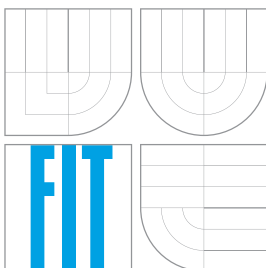
AUTHOR

Bc. MARTIN KRČMÁŘ

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

GRAFICKÉ UŽIVATELSKÉ ROZHRAŇÍ PRO POROVNÁVÁNÍ RDF MODELŮ

GRAPHICAL USER INTERFACE FOR COMPARING RDF MODELS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE

AUTHOR

Bc. MARTIN KRČMÁŘ

VEDOUcí PRÁCE

SUPERVISOR

doc. RNDr. PAVEL SMRŽ, Ph.D.

BRNO 2009

Abstrakt

Práce se zabývá vizualizací rozdílu mezi dvěma RDF modely. Jsou zde porovnávány existující nástroje a postupy, tyto nástroje jsou rozděleny do dvou kategorií. První kategorie reprezentuje textový přístup k zobrazování ontologií a potažmo jejich rozdílu, druhá zastupuje grafické metody. Na základě jejich zkoumání je navržen nový přístup. Konkrétně se jedná o nástroj, který podporuje paralelní textovou reprezentaci ontologií, která je obohacena o sémantické anotace (delta tags).

Klíčová slova

RDF model, Ontologie, Sémantický web, Zobrazování ontologií, Rozdíl ontologií

Abstract

This work elaborates on the visualization of the difference between two RDF models. Existing approaches and tools are compared, these tools are divided into two categories. The first category represents text based approach to ontology and its difference visualization, the second is a representative of graph based methods. On the basis of their comparison new approach is suggested. Specifically, tool that supports parallel (aligned) indented text layouts that are enriched with semantic annotations (delta tags) is described.

Keywords

RDF Model, Ontology, Semantic Web, Ontology Visualization, Ontology Difference

Citace

Martin Krčmář: Graphical User Interface for Comparing RDF Models, diplomová práce, Brno, FIT VUT v Brně, 2009

Graphical User Interface for Comparing RDF Models

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Smrže

.....

Martin Krčmář

May 25, 2009

Poděkování

I would like to thank my supervisor in Greece Yannis Tzitzikas for his support and ideas.

© Martin Krčmář, 2009.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1	Introduction	7
1.1	Thesis Outline	8
2	Ontology	9
2.1	What is Ontology?	9
2.2	Ontology Parts	9
2.3	RDF Models	10
2.4	OWL Models	12
3	SWKM	13
3.1	SWKM Platform	13
3.2	SWKM in our Application	14
4	Delta Visualization Approaches	15
4.1	<i>Text-based</i> Views	15
4.2	<i>Graph-based</i> Views	19
4.2.1	Node-link	19
4.2.2	Containment	20
4.2.3	3D	23
4.3	Summary	23
5	Our Approach	25
5.1	Brief Outline	25
5.2	Delta Tags: Motivation	26
5.3	Delta Tags	27
5.4	Global Neighborhood Metrics	29
5.4.1	Overall Class Similarity	30
5.4.2	Network Similarity	33
5.5	Parallel Tree Views and Delta Tags	33
5.5.1	Directed Acyclic Graph to Tree	34
5.5.2	Displaying Delta Tags	34
5.6	Complex Changes	35
6	Implementation	38
6.1	GUI Design	39
6.2	Architecture	39
6.3	Local Comparator	41
6.4	Lazy loading	42

7	Experimental Evaluation	43
7.1	Subjects	43
7.2	Evaluation tests	43
7.3	Compared tools	44
7.4	Tasks and questions employed	44
7.5	Results and discussion	45
7.5.1	User satisfaction	47
7.5.2	Task completion	47
7.5.3	Feedback	48
7.5.4	Results conclusion	48
8	Further research	49
8.1	<i>text-based</i> views with local <i>graph-based</i> views	49
8.2	Navigation among changes	50
8.3	SWKM Middleware	50
8.4	Manipulating ontologies using Triples	51
8.5	Instances	52
8.6	Fish-eye view	52
9	Conclusion	54
A	Pre Study Questions	57
A.1	Few questions before the experiment	57
B	Post Study Questions	58
B.1	Questions to answer after the main study	58
C	User Study Questionnaire	59
C.1	First for each question check one of the following possibilities	59
C.2	Study questions	59
D	User manual	61
D.1	User manual	61
D.2	Prepared Example Project	61
D.3	Create new project	62
D.4	Load an ontology	62
D.5	Display ontology model	62
D.6	Before comparing two ontologies	63
D.7	Display Ontology Difference	63
D.7.1	Comparison Tab	63
D.7.2	Comparison Control Panel	63
D.7.3	Parallel Aligned Views	63
D.7.4	Similarity Metrics	64
D.7.5	Overall similarity progress bar	65

List of Figures

2.1	RDF example	11
2.2	This figure demonstrates the structure and relationships between different ontology models mentioned and described in this chapter. You may observe that both models are based on XML (Extensible Markup Language) and especially the fact that the OWL language family extends the RDF schema.	12
3.1	Package structure showing the placement of SWKM model API in our application. In the bottom layer there is the SWKM Model API. Objects from this package are wrapped by new RDF Model API which is used by the application.	14
4.1	Ontoview comparison result.	16
4.2	Comparison of two RDFs files using Vimdiff	17
4.3	Comparison of two versions of the newspaper ontology using Protégé. The result is displayed using Table view.	18
4.4	Comparison of two versions of the newspaper ontology using Protégé. On the left, the newer ontology was loaded and then compared with the older one. On the right, first the older one was loaded and then compared with the newer.	18
4.5	Radial tree drawing technique example. Starts with the root node in the center. Children of the root are placed in the inner-most ring. The angle occupied by a child is proportional to space required by that node. Space required is recursively defined to be equal to sum of space required by its children plus some border space.	20
4.6	Comparison of two versions of the newspaper ontology using Protégé. In this figure the PromptViz plug-in is used to display the difference using Treemap. For example, the red rectangle in the bottom left corner is representing the class <i>Director</i> that has been deleted in the second version of the newspaper ontology. The highlighted rectangle represents the class <i>Columnist</i> , in the second version, this class is located in three positions in the class hierarchy, this is indicated by connections to other highlighted classes. Also this rectangle is marked by blue, due to the fact, it has been added as a subclass of the <i>External employee</i> class.	22
4.7	CropCircles tree visualization technique used for displaying ontologies. . . .	23
5.1	Class <i>Person</i> with changes. The left part of this Figure represents first version of an ontology containig this class and the right side its newer version with obvious changes.	26

5.2	In this figure you can observe the similarity metrics computed for the class <i>Columnist</i> from the <i>newspaper</i> ontology. You can see the <i>suprs</i> similarity value 0.750, in the first version this class had three super classes but in the second version it has four. Also in the first version there were two instances and one of them has been deleted and a new one added, that is why the instances similarity is 0.333. You can notice two new metrics, namely <i>overall</i> and <i>average</i> , they will be explained later.	29
5.3	In this figure you can see the similarity metrics computed for property called <i>responsible_for</i> of the class <i>Editor</i> again from the <i>newspaper</i> ontology. There you can observe the <i>instances</i> similarity and its value 0.625, which indicates instances referenced by this property in different instances have changed quite lot.	30
5.4	Computing the <i>overall</i> similarity metric illustration. Various colored areas mirror various ontology parts whose similarity computation is covered with different similarity metrics.	32
5.5	Example of a class with more than one direct super class. In this case it is the class <i>Editor</i> and the corresponding tooltip informing the user about all its super classes.	35
5.6	Example of difference visualization without identifying moved classes. This is one of the extremes that could happen, the whole class hierarchy from the first version has been wrapped by a new class.	36
5.7	Example of displaying aligned temporary moved classes. As you can see, the class <i>General_Employee</i> has been moved to different position in the class hierarchy in the second version. But to provide the user all the useful similarity metrics, we display this class aligned with the original class at the same line. Notice the similarity metric <i>Super classes</i> and its value 0.000, indicating these two classes have no super classes in common, which is obvious. In the first version the class was a top level class with no super classes, on the other hand in the second version the class is located somewhere deeper in the class hierarchy, so it has to have at least one super class, thus the similarity according to super classes equals zero.	37
6.1	The final application is presented in this figure. On the left side you may notice the project manager with two opened projects (the red icon indicates currently active project), on the right side there are projects's frames with different tabs. In the <i>Newspaper</i> frame, the comparison view is currently opened, in the <i>MovieDatabse</i> frame the ontology viewer is opened.	40
6.2	Package structure of the application	41
7.1	In this figure there is a chart representing final rating gained from the post study questionnaire. There are three clusters, each supplies particular tool compared. The first one is our application (RDFDiffVizz), then PromptDiff and PromptViz plug-in. Each color mirrors one subject and points gained from this subject.	46
8.1	Local <i>graph-based</i> visualization. This is visualization focused on the concept <i>Manager</i> from the <i>newspaper</i> ontology.	50

8.2	This figure shows possible exploitation of the SWKM Diff service in our application. Notice the middle information column and digits aligned on its left side. These digits represent actual number of change operations assigned to particular nodes (classes). The class <i>Columnist</i> has one new sub class added, this is mirrored with the tooltip window showing this specific operation. .	51
8.3	This figure demonstrates how the problem of displaying huge number of instances could be solved.	52
8.4	Example of fish-eye view technique applied to ordinary tree view (for instance, JTree component as used in our application).	53
D.1	In this Figure one active and one inactive project example is demonstrated. The red icon designates the currently active project. Under the active project there are two loaded ontologies, both set to be compared.	62
D.2		64
D.3	This progress bar reflects the overall similarity metrics in the case of classes, in the case of properties and instances it reflects the average similarity metrics.	65

List of Tables

5.1	Applicability of Metrics	27
-----	------------------------------------	----

Chapter 1

Introduction

Ontologies provide shared conceptualization of a domain by defining the concepts in the domain and describing how those concepts are related to each other. However, most domains of discourse are not static, but evolve as the understanding of the domain grows. As a consequence, the ability to compute the differences that exist between two ontologies is an important step to cope with the evolving nature of the Semantic Web. Several methods how to compute and subsequently visualize such differences could be found, therefore the fundamental objective of this work is not to come up with a new technique for computing the difference but to design new approach to visualize it.

In general *deltas* can be exploited for ontology distinction representation and what is more for reducing the amount of data that need to be exchanged and managed over the network hence they are useful for building advanced Semantic Web synchronization and versioning services. The visualization of *deltas* can aid humans in:

- Understanding the evolution history of one ontology.
- Migrating metadata to newer versions of ontologies.
- Creating mappings between different ontologies.
- For building a merge tool.

In this work we confine ourselves to ontologies represented in RDFS. Four comparison functions for RDF models were introduced and analysed in [23]. Specifically, in that work *deltas* are represented as sets of change operations, namely sets of *add* (addition) and *del* (deletion) triple operations that can be used to transform one knowledge base K to another denoted K' . Such set of change operations, although essential for achieving synchronization services, are not elegant for the human eye to decode and understand. The main challenge of our work is to build a tool capable of visualizing *deltas* that is both user friendly and effective.

There are several tools designed for comparing RDF models already implemented including SemVersion¹, PromptDiff² plug-in for Protégé³, OntoView or SWKM Diff service⁴.

¹<http://semanticweb.org/wiki/SemVersion>

²<http://protege.stanford.edu/plugins/prompt/PromptDiff.html>

³<http://protege.stanford.edu>

⁴<http://athena.ics.forth.gr:9090/SWKM/>

OntoView[9], PromptDiff and the Protégé plug-in called Prompt-Viz⁵ also enable to visualize the differences. The first two visualize the differences using *text-based* views (more on this in Chapter 4). The last one, PromptViz [13], combines the Treemap layout technique for graph visualization with a zoomable user interface to provide enhanced cognitive support for understanding the differences.

This work deals with comparison of aforementioned techniques and discusses their advantages and disadvantages. Subsequently, we introduce a new approach that combines and extends these techniques. Specifically, the contribution of our work lies in providing:

- Parallel aligned views to support better orientation in the resulting ontology comparison 5.5.
- Ontology visualization based on indented lists as they turned out to be one of the most suitable methods for this purpose according to several surveys [10].
- Provide the users additional information (delta tags 5.2) to help them better understand the essence of ontology modifications.
- Suggest other new features that could help to enrich our approach and give ideas for further research.

To test our ideas and suggestions we designed and implemented new tool called RDFSdiffVizz which supports the aforementioned functions. One whole chapter of this work deals with implementation details and another chapter regards the evaluation study conducted in order to verify correctness and usefulness of our approach.

1.1 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 gives brief introduction into terms used throughout this work, especially ontology conception is presented altogether with different formal models used to capture it. Chapter 3 describes the SWKM⁶ (Semantic Web Knowledge Middleware) platform and its features used in our application. Chapter 4 discusses related work and categorizes the existing approaches into two main categories according to their ontology representation concept. Both categories are then further examined in order to collect the best qualities from each and propose their utilization in our application. Chapter 5 is the main chapter and introduces and analyses our approach. New conceptions are suggested and inscribed such as *delta tags*, *global neighborhood* or *parallel tree views*. Chapter 6 denotes the implementation of our work. Its main purpose is not to give comprehensive elevation on all implementation details but rather a brief outline of used techniques and certain overall sight of application's architecture. Chapter 7 summarizes evaluation results gained from our evaluation study conducted in order to test our hypotheses. Description of experimental tests, compared tools, subjects involved or feedback polled from this study are summarized and presented in that chapter. Chapter 8 attends to further research issues since several ideas and proposals could not have been implemented due to time limitation. Finally, Chapter 9 concludes this thesis and resumes all its contribution.

⁵<http://www.thechiselgroup.org/promptviz>

⁶<http://athena.ics.forth.gr:9090/SWKM>

Chapter 2

Ontology

This chapter aims at giving brief introduction into several terms used throughout this thesis, especially ontology conception is presented altogether with two formal models used to capture it. The first model is RDF model described in Section 2.3, this is the one actually used and implemented in our application. Then to be comprehensive another model widely used, although not involved in our approach, is mentioned in Section 2.4.

2.1 What is Ontology?

Ontologies provide a formal specification of a domain of discourse and are becoming increasingly prevalent in the high tech world. According to [8] ontologies, as sets of concepts and their interrelations in a specific domain, have proven to be useful tool in the areas of digital libraries, the semantic web, and personalized information management. As a result, there is a growing need for effective ontology visualization for design, management and browsing. The term *ontology* is initially borrowed from philosophy, where an ontology is a systematic account of existence. Another useful definition can be found in [6]. Ontology is an explicit specification of a conceptualization. The term "conceptualization" is defined as an abstract, simplified view of the world, which needs to be represented for some purposes. It contains objects, concepts and their entities that are presumed to exist in some area of interest. And also the relations that hold among them.

Where can we find ontologies? They are widely used in the medical community and will provide the backbone of the Semantic Web. On the surface ontologies may appear to be like database schemas; however, ontologies are not a way of organizing a specific data set for efficient retrieval, but rather a reusable structure for data within a domain that is designed to capture all the inherent relationships and meta-data among the knowledge that will be stored in there.

2.2 Ontology Parts

Ontology can consist of several different parts. The following is based on the top level items of a frame-based knowledge model as described in [12]:

- **Classes** are collections of objects that have similar properties. Classes are arranged into a subclass-superclass hierarchy using either single or multiple inheritance. Each class has slots attached to it. Slots can be inherited by the subclasses. Later in our application these slots will be referenced as *properties*.

- **Slots** are named binary relations between a class and either another class or a primitive object (string or a number). Slots attached to class may be further constrained by facets.
- **Facets** are named temporary relations between a class, a slot and either another class or a primitive object. Facets may impose additional constraints on a slot attached to a class, such as the cardinality or value type of a slot.
- **Instances** are individual members of classes.

There are two primary methods that have been used to construct ontologies. The first one based on Description Logic, the second one on Frames. The main difference between these two systems is that as a subset of first order predicate logic, Description Logic includes the ability to automatically classify new concepts descriptions with respect to previously defined concepts and to check the consistency of declared statements which frees us from having to explicitly enter all the necessary information about new concept. The system will automatically complete any implied information. In the Frame based system all information must be explicitly defined.

2.3 RDF Models

RDF stands for Resource Description Framework and it is a W3C Recommendation¹. RDF specification provides a lightweight ontology system to support the exchange of knowledge on the Web. The RDF Specification is build on URI and XML technologies.

The following RDF description is taken from [2]. RDF was originally designed as a metadata data model. It has come to be used as a general method for conceptual description or modeling of information that is implemented in web resources; using a variety of syntax formats.

RDF data model does not vary a lot from classic conceptual modeling approaches such as Entity-Relationship or Class diagrams. It is based upon the idea of making statements about resources in the form of subject-predicate-object expressions (will be also refered as Triples later in this work). Resource is an object that can be pointed at by a URI (Uniform Resource Identifier). The subject in the triple denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. Here is a simple triple example: (Netherlands, hasCapital, Amsterdam) saying the Capital City of Netherlands is Amsterdam. It should be also stressed out the subject of one statement can be the object of another. This is illustrated in Figure 2.1.

Code Listing 2.1: RDF example

```
<Description about="http://www.countries.org/countries#Netherlands">
  <hasCapital resource="http://www.cities.org/cities#Amsterdam">
    <areacode>020</areacode>
  </hasCapital>
</Description>
<Description about="http://www.cities.org/cities#Rotterdam">
  <trainConnection resource="http://www.cities.org/cities#Amsterdam" />
</Description>
```

¹<http://www.w3.org/RDF>

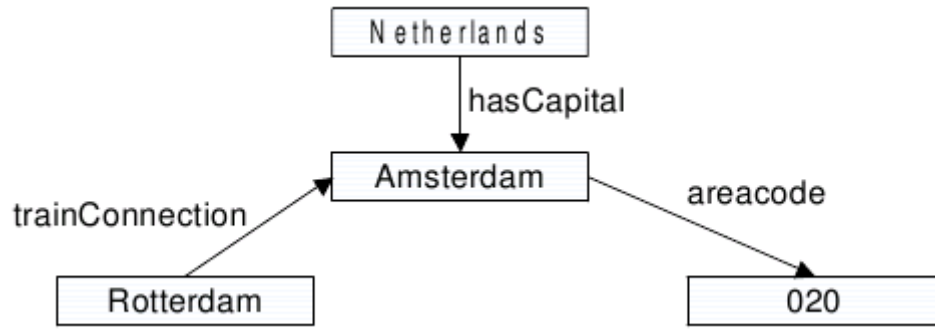


Figure 2.1: RDF example

At this point the RDF misses a way to define a vocabulary. It is not clear whether to use *country* or *nation* or if Netherlands is a country or something else. There is an RDF Schema established for such purposes. RDF Schema allows us to define vocabulary terms and the relations between these terms. It gives meaning to particular RDF predicates and resources. In more complicated words RDF Schema is an extensible knowledge representation language, providing basic elements for the description of ontologies, otherwise called RDF vocabularies, intended to structure RDF resources[1]. Main schema constructs are *classes*, *subclasses*, *property domains* and *ranges*. So the resulting simple RDF schema example could look as shown in Listing 2.2.

Code Listing 2.2: The schema file: <http://www.geography.org/schema.rdf>

```

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
  <rdfs:Class rdf:about="#Country">
    <rdfs:subClassOf rdfs:resource="#GeographicEntity" />
  </rdfs:Class>
  <rdf:Property rdf:about="#hasCapital">
    <rdfs:domain rdfs:resource="#Country" />
    <rdfs:range rdfs:resource="#Capital" />
  </rdf:Property>
</rdf:RDF>

```

Code Listing 2.3: RDF example

```

<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:geo="http://www.geography.org/schema.rdf#">
  <geo:Country rdf:about="#Netherlands">
    <geo:hasCapital rdfs:resource="#Amsterdam" />
  </geo:Country>
  <geo:Capital rdf:about="#Amsterdam" />
</rdf:RDF>

```

In the rest of this work and in our application we will always use RDF models since SWKM platform works with the very RDF models. But to be comprehensive also another language designed for ontology representation is going to be described in the next section, although we are not going to use it.

2.4 OWL Models

OWL abbreviates the Web Ontology Language, a family of knowledge representation languages for authoring ontologies, and is endorsed by the World Wide Web Consortium². According to [3] this family of languages is based on two semantics: OWL DL and OWL Lite semantics are based on Description Logics, which have attractive and well-understood computational properties, while OWL Full uses a novel semantic model intended to provide compatibility with RDF Schema. OWL is an extension to the aforementioned RDF, providing greater machine interpretability of Web content than that supported by XML, RDF and RDF Schema. This is enabled thanks to the fact that OWL has more facilities for expressing meaning and semantics.

For the better overall conception of previous models and how they fit into the world of Semantic Web, the Figure 2.2 is included.

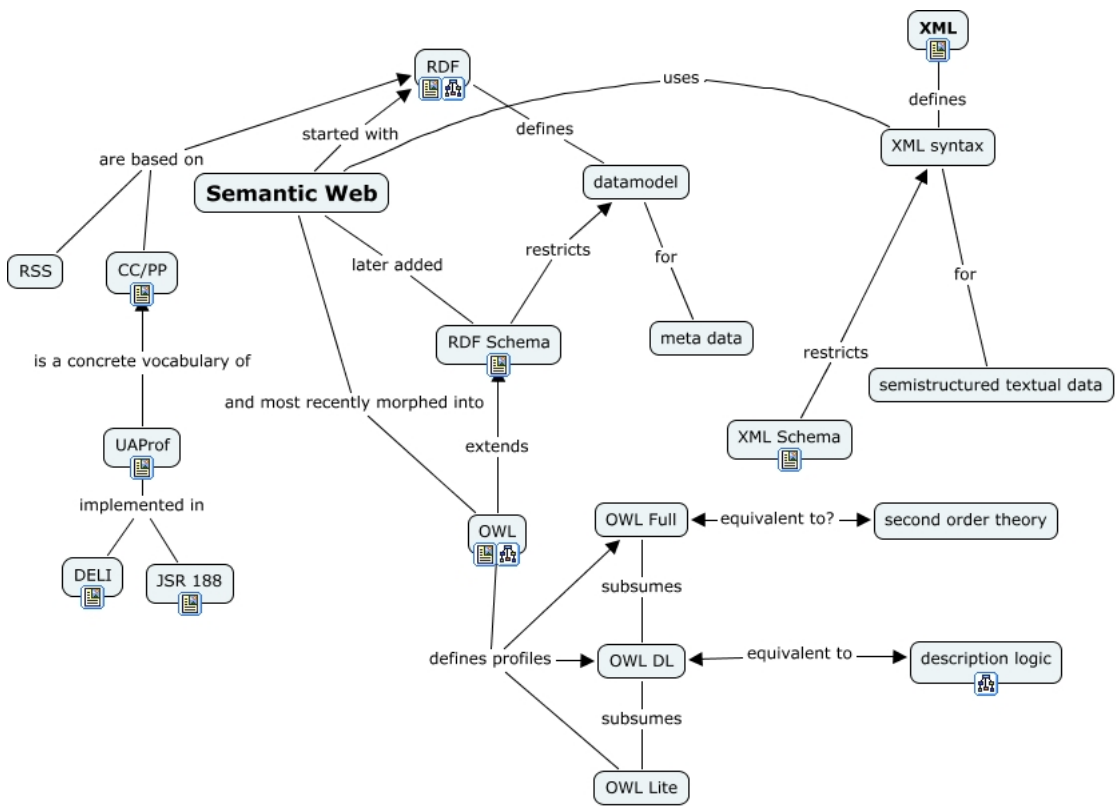


Figure 2.2: This figure demonstrates the structure and relationships between different ontology models mentioned and described in this chapter. You may observe that both models are based on XML (Extensible Markup Language) and especially the fact that the OWL language family extends the RDF schema.

<http://dsrg.mff.cuni.cz/gergic/versatile/cmaps/Semantic%20Web.html>

²<http://www.w3.org/TR/owl-features>

Chapter 3

SWKM

SWKM¹ stands for the Semantic Web Knowledge Middleware and it is a set of tools for parsing, storage, manipulation and querying of Semantic Web Knowledge bases represented by RDF models mentioned in Section 2.3. This platform is being developed by the Semantic Web research group at ICS-FORTH².

3.1 SWKM Platform

The SWKM platform consists of two main parts, the first one is called SWKM Model and the second one SWKM Services. The SWKM Services provides a set of services that aim at manipulating a remote RDF database via SOAP API calls. The remote database is represented by SWKM Middleware, which is a persistent storage mechanism for RDF models. Using these services we are able to import RDF Schemas or RDF files containing instances in the database, export schemas or instances, query upon the already stored material via the use of RQL queries or update stored instances via RUL queries. Here you can include *Import*, *Export*, *Query*, *Update* and *Delete* services. Other services offer functionalities to support evolving ontologies. Among these we can place *change*, *comparison*, *versioning* and *registry* service. The *comparison* service was originally intended to be used in our application. We have decided, nevertheless, to use a different approach that will be described later in Chapter 6, instead of computing sets of change operations (Deltas) and then re using these Deltas for visualization purposes. Although, this and the other services could be used as an extension altogether with the SWKM Middleware, more about that in Chapter 8.

This Semantic Web platform was mainly chosen due to the need of parsing RDF files. Parsing RDF files is a vital part for our application and that is where the SWKM comes into play. Unlike most other RDF parsers, this one provides both an Object-based Model and a Triple-based Model for main RDF manipulation. So far we use only the object oriented approach in our application, nevertheless. But again, the Triple-based method could be also exploited. For instance, when the user wants to manipulate the ontology, the easiest way how to achieve this, would be to provide the user methods for creating such triples and then let the SWKM to process them, thus applying the changes to the ontology. Also the *comparison* service produces a set of such triples to represent the difference between versions, so the Triples would afford a unified concept for general ontology manipulation and representation. Again, more about this is discussed in Chapter 8.

¹<http://athena.ics.forth.gr:9090/SWKM>

²<http://forth.gr>

3.2 SWKM in our Application

The SWKM model API is not used directly instead wrapper classes creating our own RDF API level are implemented. The main reason is to make the application independent upon the certain RDF API so at any time we can switch to another platform assuring RDF files parsing and other necessary features. Easier upgrade to newer versions can be taken as another benefit. During the development time this model was in its first version, later second version has been released, which could be also wrapped by our API, thus making the application work with the newer version without necessitating any further modifications.

Our RDF API provides several operations for all the main parts of every RDF model. Namely, for the model itself, for its namespaces, classes, instances, properties, property instances, literals and few others. The API does not provide all the functionality granted by the underlying SWKM Model, it ensures only the essential features for our application. The relation between the SWKM Model API and our application can be seen in Figure 3.1.

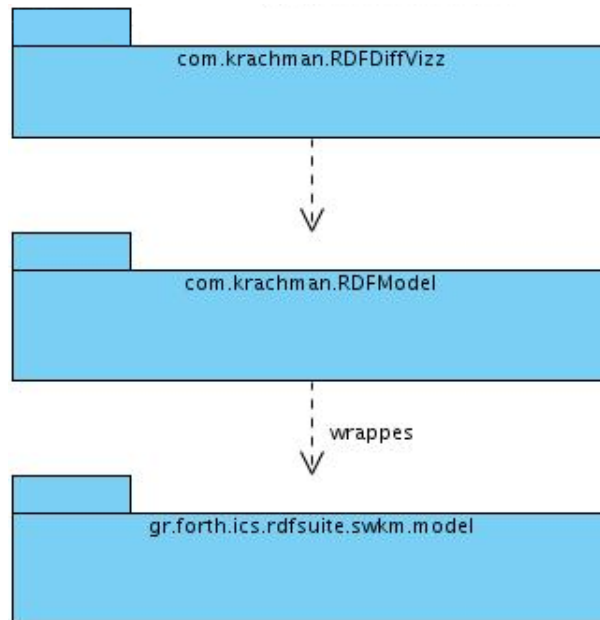


Figure 3.1: Package structure showing the placement of SWKM model API in our application. In the bottom layer there is the SWKM Model API. Objects from this package are wrapped by new RDF Model API which is used by the application.

Chapter 4

Delta Visualization Approaches

We could distinguish two primary approaches to visualize differences between ontologies: *text-based* views and *graph-based* views. This whole chapter deals with introduction, examination and comparison of both categories trying to collect the best qualities from each and propose their utilization in our application.

4.1 *Text-based* Views

This is the most common way to visualize the difference in software version control systems and *diff* applications. They use line-by-line comparison of two files or two versions and highlighting the differences. This way, for instance, OntoView[9] visualizes differences between two RDF files. In particular, their representation in RDF/XML is presented in parallel and aligned, and their differences are highlighted as we can see in Figure 4.1.

To evaluate the usefulness of such techniques we conducted one experiment. We decided to use a simple and small ontology, called *newspaper*, containing twenty-two classes and sixty-six instances. Using Protégé we created another (newer) version of the same ontology and stored it in a different file. We decided to try visualizing the difference using Vimdiff¹ tool to see whether this idea of line-by-line comparison of RDF files is sufficient for our needs. We exported two versions of one ontology into RDFS files and then compared these files using Vimdiff. Vimdiff displayed only the parts of the files that differ so the user is not disturbed with unnecessary information. However such visualization suffer from the following shortcoming: they do not offer good overviews of the differences. For instance, if someone wants to see the differences in a hierarchical structure they have to follow every single RDF entry and make the image of the ontology structure by themselves. A screenshot is shown in Figure 4.2. The same shortcoming exists even if we decide to represent the ontologies as sets of triples.

Subsequently, we tried Prompt (a plug-in of Protégé). One of the features of this plug-in is its ability to compare versions of the same ontology. Prompt plug-in offers two visualization methods. The first one is a simple table view where the first column lists all classes and instances of the first ontology while the second shows corresponding list of the second ontology. Selecting the corresponding row in the table reveals the specific details about any change. If there was a class *C* in the first version which has been deleted in the second version, then the particular row in the second column would be blank and vice versa. There are another four columns in the table that describe the change in more detail.

¹<http://www.vim.org/html/doc/diff.html>

</daml:Ontology>	</daml:Ontology>	
<pre><rdfs:Class rdf:ID="Animal"> <rdfs:label>Animal</rdfs:label> <rdfs:comment> This class of animals is illustrative of a number of ontological idioms. </rdfs:comment> </rdfs:Class></pre>	<pre><rdfs:Class rdf:ID="Animal"> <rdfs:label>Animal</rdfs:label> <rdfs:comment> The class of living things that have the capacity for spontaneous movement and rapid motor responses to stimulation, having cells without cellulose walls. </rdfs:comment> </rdfs:Class></pre>	identical
<pre><rdfs:Class rdf:ID="Male"> <rdfs:subClassOf rdf:resource="#Animal"/> </rdfs:Class></pre>	<pre><rdfs:Class rdf:ID="Male"> <rdfs:subClassOf rdf:resource="#Animal"/> <daml:disjointWith rdf:resource="#Female"/> </rdfs:Class></pre>	identical
<pre><rdfs:Class rdf:ID="Female"> <rdfs:subClassOf rdf:resource="#Animal"/> </rdfs:Class></pre>	<pre><rdfs:Class rdf:ID="Female"> <rdfs:subClassOf rdf:resource="#Animal"/> </rdfs:Class></pre>	
<pre><rdf:Property rdf:ID="hasParent"> <rdfs:domain rdf:resource="#Animal"/> <rdfs:range rdf:resource="#Animal"/> </rdf:Property></pre>	<pre><rdf:Property rdf:ID="hasParent"> <rdfs:domain rdf:resource="#Animal"/> <rdfs:range rdf:resource="#Animal"/> </rdf:Property></pre>	(characterization) (characterization) identical conceptual change
<pre><rdf:Property rdf:ID="hasFather"> <rdfs:subPropertyOf rdf:resource="#hasParent"/> <rdfs:range rdf:resource="#Male"/> </rdf:Property></pre>	<pre><rdf:Property rdf:ID="hasFather"> <rdfs:subPropertyOf rdf:resource="#hasParent"/> <rdfs:range rdf:resource="#Male"/> </rdf:Property></pre>	
<pre><rdfs:Class rdf:ID="Person"> <rdfs:subClassOf rdf:resource="#Animal"/></pre>	<pre><rdfs:Class rdf:ID="Person"> <rdfs:subClassOf rdf:resource="#Animal"/></pre>	

Figure 4.1: Ontoview comparison result.

These columns are designated as *renamed*, *operation*, *map level* and *rename explanation*. The first one can contain a Boolean value signifying whether the class or instance has been renamed or not. The second describes the change operation as *deletion*, *addition*, *splitting*, *merging* or *mapping*. This view, although useful, does not give us a good overview of what has really happened and has been changed. In this view you can not see the hierarchy of the ontology, only the list of classes and instances.

Prompt plug-in offers also a *tree view*. According to [11] this interface was inspired by the interface that MS Word uses to present changes in documents: the text that was deleted is crossed out while the added text is underlined. In the layout there is an expandable horizontal tree component containing the first version of the ontology. All the changes are represented in this tree view. For example, if the second version contains class *C* which has been added and was not in the first version this class will be shown in the tree view and will be marked as *deleted*. There is also color coding for these two types of changes where different styles represent different types of changes. For example, newly added classes are underlined and in blue, deleted classes are crossed out and in red, while moved classes are grayed out in their old positions and appear in bold in their new positions. You can navigate through all the changes using *next* and *previous* change button and can apply other operation like *accept class change* etc. This is shown in Figure 4.4. Nevertheless, this view does not convey very good overview of changes either.

26	<rdfs:subClassOf rdf:resource="&rdof;Resource"/>	26	<rdfs:subClassOf rdf:resource="&rdof;Resource"/>
27	</rdfs:Class>	27	</rdfs:Class>
28	<rdfs:Class rdf:about="&rdof;Billing_Chart"	28	<rdfs:Class rdf:about="&rdof;Billing_Chart"
29	rdfs:label="Billing_Chart">	29	rdfs:label="Billing_Chart">
30	<rdfs:subClassOf rdf:resource="&rdof;Layout_info"/>	30	<rdfs:subClassOf rdf:resource="&rdof;Layout_info"/>
31	</rdfs:Class>	31	</rdfs:Class>
32	<rdfs:Class rdf:about="&rdof;Boss"	32	<rdfs:Class rdf:about="&rdof;Boss"
33	rdfs:label="Boss">	33	rdfs:label="Boss">
34	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>	34	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>
35	</rdfs:Class>	35	</rdfs:Class>
36	<rdfs:Class rdf:about="&rdof;Columnmist"	36	<rdfs:Class rdf:about="&rdof;Columnmist"
37	rdfs:label="Columnmist">	37	rdfs:label="Columnmist">
38	<rdfs:subClassOf rdf:resource="&rdof;Author"/>	38	<rdfs:subClassOf rdf:resource="&rdof;Author"/>
39	<rdfs:subClassOf rdf:resource="&rdof;Employee"/>	39	<rdfs:subClassOf rdf:resource="&rdof;Employee"/>
40	</rdfs:Class>	40	<rdfs:subClassOf rdf:resource="&rdof;External_employee"/>
41	</rdfs:Class>	41	</rdfs:Class>
42	<rdfs:Class rdf:about="&rdof;Content"	42	<rdfs:Class rdf:about="&rdof;Content"
43	rdfs:label="Content">	43	rdfs:label="Content">
44	<rdfs:comment>This is an abstract superclass of both advertisements and a	44	<rdfs:comment>This is an abstract superclass of both advertisements and a
45	<rdfs:subClassOf rdf:resource="&rdof;Resource"/>	45	<rdfs:subClassOf rdf:resource="&rdof;Resource"/>
46	</rdfs:Class>	46	</rdfs:Class>
47	<rdfs:Class rdf:about="&rdof;Content_Layout"	47	<rdfs:Class rdf:about="&rdof;Content_Layout"
48	rdfs:comment="Defines a rectangle of space on a page"	48	rdfs:comment="Defines a rectangle of space on a page"
49	rdfs:label="Content_Layout">	49	rdfs:label="Content_Layout">
50	<rdfs:subClassOf rdf:resource="&rdof;Layout_info"/>	50	<rdfs:subClassOf rdf:resource="&rdof;Layout_info"/>
51	</rdfs:Class>	51	</rdfs:Class>
52	<rdfs:Class rdf:about="&rdof;Director"	52	<rdfs:Class rdf:about="&rdof;Director"
53	rdfs:label="Director">	53	rdfs:label="Director">
54	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>	54	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>
55	</rdfs:Class>	55	</rdfs:Class>
56	<rdfs:Class rdf:about="&rdof;Director's assistant"	56	<rdfs:Class rdf:about="&rdof;Director's assistant"
57	rdfs:label="Director's assistant">	57	rdfs:label="Director's assistant">
58	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>	58	<rdfs:subClassOf rdf:resource="&rdof;Manager"/>
59	</rdfs:Class>	59	</rdfs:Class>
60	<rdfs:Class rdf:about="&rdof;Editor"	60	<rdfs:Class rdf:about="&rdof;Editor"
61	rdfs:comment="Editors are responsible for the content of sections."	61	rdfs:comment="Editors are responsible for the content of sections."
62	rdfs:label="Editor">	62	rdfs:label="Editor">

Figure 4.2: Comparison of two RDFs files using Vimdiff

f1	f2	renamed	operation	map level	rename explanation
	● Boss	No	Add		<null>
	● Director's assistant	No	Add		<null>
	● External employee	No	Add		<null>
	● Secretary	No	Add		<null>
	■ car	No	Add		<null>
● Director		No	Delete		<null>
● Columnist	● Columnist	No	Map	Directly-changed	frame name and type are the same
● Director Supervision Relation	● Director Supervision Relation	No	Map	Directly-changed	frame name and type are the same
● Reporter	● Reporter	No	Map	Directly-changed	frame name and type are the same
◆ Tim	◆ Tim	No	Map	Directly-changed	frame name and type are the same
◆ Anne BasketballHead	◆ Anne BasketballHead	No	Map	Isomorphic	frame name and type are the same
◆ directs	◆ directs	No	Map	Isomorphic	frame name and type are the same
◆ directs	◆ directs	No	Map	Isomorphic	frame name and type are the same
◆ directs	◆ directs	No	Map	Isomorphic	frame name and type are the same
◆ Joe Schmo	◆ Joe Schmo	No	Map	Isomorphic	frame name and type are the same
◆ Kelly	◆ Kelly	No	Map	Isomorphic	frame name and type are the same
◆ Kim	◆ Kim	No	Map	Isomorphic	frame name and type are the same
◆ Larry Tennis-nut	◆ Larry Tennis-nut	No	Map	Isomorphic	frame name and type are the same
◆ San Jose Mercury News	◆ San Jose Mercury News	No	Map	Isomorphic	frame name and type are the same
● Advertisement	● Advertisement	No	Map	Unchanged	frame name and type are the same
● Article	● Article	No	Map	Unchanged	frame name and type are the same
● Author	● Author	No	Map	Unchanged	frame name and type are the same

Figure 4.3: Comparison of two versions of the newspaper ontology using Protégé. The result is displayed using Table view.

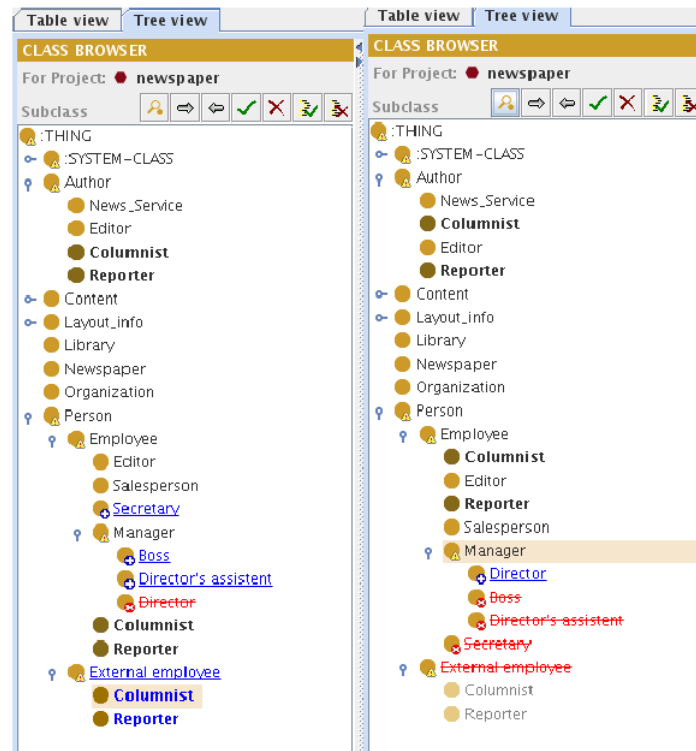


Figure 4.4: Comparison of two versions of the newspaper ontology using Protégé. On the left, the newer ontology was loaded and then compared with the older one. On the right, first the older one was loaded and then compared with the newer.

4.2 Graph-based Views

Why graphical representation? Do we actually need a tool capable of graphical visualization since we have got the aforementioned tools including the one new proposed (see Chapter 5) that are able to represent the difference using *text-based* views? Prompt and OntoView both lack visualizations that supports understanding how the ontologies have changed at an abstract level. But this is not the only reason.

The main reason for investigating *graph-based* views is that they could provide better overviews of the changes thus satisfy better the cognitive demands of users, especially if the involved ontologies are large (e.g. if contain hundreds of elements).

Text-based tools are not effective for large ontologies. For example, to answer a question such as “Where have the most changes occurred in the ontology?” is difficult for an ontology with 50,000 concepts using the default views in PromptDiff. So graphical tools could be good alternatives and in some specific cases even more effective.

In the rest of this Section we will introduce several graph visualization techniques and investigate their suitability for ontology visualization. We are going to group these graph drawing methods into two major groups reflecting characteristics of the presentation, interaction techniques and functionality supported.

Since most ontologies have a hierarchical backbone formed by *is-a* (`subClassOf`) relationships, visual layouts for trees are a natural way of representing them. There are several (pretty effective) techniques for displaying trees. The two primary techniques that are employed for drawing trees are *connection* and *containment*. The first method going to be explored: Node-link belongs to the first group.

4.2.1 Node-link

It is truly the most familiar method for drawing trees to the general populace. According to [8] this category of techniques represents ontologies (graphs, trees) as a set of interconnected nodes, presenting the taxonomy with a top-down or left to right layout. The user is generally allowed to expand and retract nodes and their subtrees, in order to adjust the detail of the information shown and avoid display clutter. One can find many tools using node-link technique to display ontologies. OntoViz is a Protégé visualization plug-in that uses the GraphViz² library to create a very simple 2D graph visualization method. The ontology is presented as a 2D graph with the capability for each class to present, apart from the name, its properties, and inheritance and role relations. IsaViz³ is a visual environment for browsing and authoring RDF ontologies represented as directed graphs. Other tools we just write out: OntoTrack, GoSurfer⁴, GOBar⁵, OntoSphere. Survey of all these tools can be found in [8].

Nevertheless, one of the primary problems with node-link diagrams is their poor utilization of screen space, leaving the root side of the tree completely empty, usually top or left of the screen, and overcrowding the opposite side. Although, there are some alternatives that can be used to overcome this problem. These alternatives tries to layout the nodes and links in a more space efficient manner. For instance, radial layouts or hyperbolic layouts (see Figure 4.5). Radial Tree according to [15] is a good technique for displaying the entire tree because it is able to summarize many different nodes using a limited amount of space

²<http://graphviz.org>

³<http://www.w3.org/2001/ii/IsaViz>

⁴<http://www.gosurfer.org>

⁵<http://Katahdin.csh.org:9331/GO>

and it does not naturally facilitate navigation. Compared to Treemap, the Radial Tree is better able to show the relationships among different nodes and it is also easier to tell the depth of a particular node.

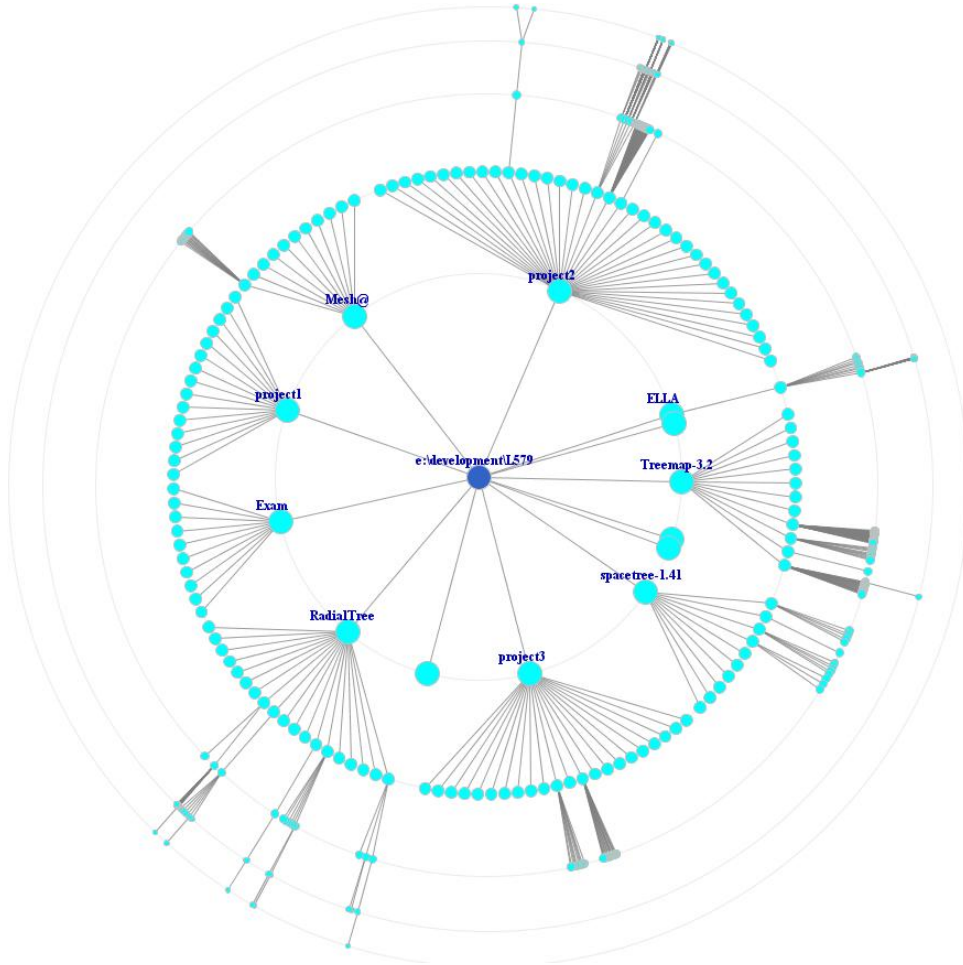


Figure 4.5: Radial tree drawing technique example. Starts with the root node in the center. Children of the root are placed in the inner-most ring. The angle occupied by a child is proportional to space required by that node. Space required is recursively defined to be equal to sum of space required by its children plus some border space.

But still node-link techniques does not satisfy our needs since the ontologies may comprise thousands of classes and instances. If properties are visualized then the layouts will suffer from a lot of crossings, as their semantic graphs tend to follow power-law distributions [17, 16].

4.2.2 Containment

Then we have the second category of containment or Space-filling algorithms. The idea of using containment to draw a tree is not as well known to most people as the node and link structure of the connection method. These containment techniques use the root node to occupy all or nearly all of available screen area. This area is then partitioned into

segments for each child continuing until only lead nodes remain. The primary advantage of these methods is that they use the available screen space much more effectively than previously mentioned connection methods such as node-link. This fact makes them practical for displaying large ontologies at once.

The typical representative is Treemap algorithm. Treemaps are introduced in [7], it is a visualization technique that depicts sub-ordination of tree nodes through recursive subdivision of rectangles that represent these nodes. The size of a rectangle is proportional to the size of the corresponding node, and users can choose among several partitioning algorithms. They are effective when the user cares mostly about leaf nodes and their properties but does not need to focus on the topology of the tree, or the topology is trivial, at most two or three levels. One Treemap example is shown in the Figure 4.6. Treemaps have since been widely adopted and applied to a variety of problems. For example, Treemaps have been applied in Wattenberg’s visualization of stock market data in the SmartMoney Map of the Market [21, 22], to present groups of related pictures in photo-browsing applications, to visualize Usenet activity and many others.

Another algorithm that has already been used directly for ontology visualization purposes is CropCircles [20]. It is relatively new technique for visualizing complex hierarchies by encircling with progressively larger circles on a plane. They are in effect a circular form of Treemap. Unlike Treemaps, CropCircles sacrifices the space-fillingness for better visual clarity, enhancing understanding of the topology. As the name predicates circles in this method represent nodes in a tree, every child circle is nested inside its parent circle. The smallest subtrees (the leaf nodes) have the minimum size of all circles. For every node, CropCircles sorts its children in descending order according to their subtree sizes, and then lays them out in that order. The sorting creates a sense of order in an otherwise unordered 2D space within a circle. The larger nodes occupy more space, showing more importance, and encourage users to explore them first.

Other algorithms such as Squarified Treemaps [5] an extension to the Treemap method or another modification of the Treemap algorithm found in [19], BeamTrees [18], TreeRings, StarTree and others can be found.

There are several surveys and comparison studies for such methods. For example [4] describes two experiments that compare four two-dimensional visualizations of hierarchies, in [10] you can find some user experiments with Tree Visualization Systems. Another great comparison is [8] which presents visualization techniques and categorizes their characteristics and features in relation with a set of requirements compiled for an ontology visualization tool.

According to [8] visualization of ontologies is not an easy task. An ontology is something more than a hierarchy of concepts. It is enriched with role relations among concepts and each concept has various attributes related to it. Furthermore, each concept most probably has instances attached to it, which could range from one or two to thousands. Therefore, it is not simple to create a visualization that will effectively display all this information and at the same time allow the user to easily perform various operations on the ontology.

In the field of ontology visualization, there are several works, mostly in 2D. Apart from the systems that propose visualizations especially tailored for ontologies, there are a number of other techniques used in other contexts such as graph or file system visualization, that could be adapted to display ontologies.

In [13] you can find comparison of PromptDiff and PromptViz plug-in. There is a user study and the results are quite interesting. This is basically comparison between a text-based view and a graphical representation. According to the results, both approaches



Figure 4.6: Comparison of two versions of the newspaper ontology using Protégé. In this figure the PromptViz plug-in is used to display the difference using Treemap. For example, the red rectangle in the bottom left corner is representing the class *Director* that has been deleted in the second version of the newspaper ontology. The highlighted rectangle represents the class *Columnist*, in the second version, this class is located in three positions in the class hierarchy, this is indicated by connections to other highlighted classes. Also this rectangle is marked by blue, due to the fact, it has been added as a subclass of the *External employee* class.

0.033449478447437286

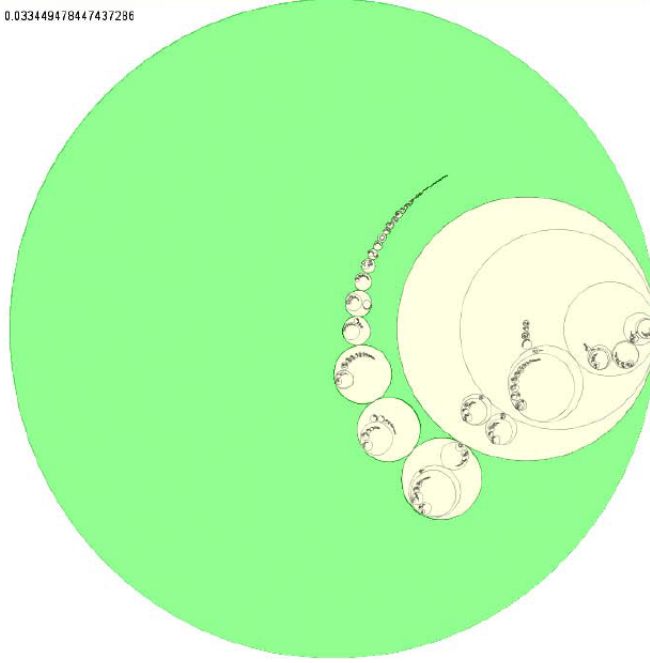


Figure 4.7: CropCircles tree visualization technique used for displaying ontologies.

have some advantages and disadvantages. For example navigation in a treemap was found difficult for some users or arcs between nodes in treemap were found confusing. But some questions concerning location or impact of changes can be answered more simply with Prompt-Viz than with basic PromptDiff. Anyway for more information look in [13].

4.2.3 3D

We also studied 3D graph drawing algorithms and their suitability for ontology difference visualization purposes. On the one hand, the extra dimension could give, literally, more "space", and this would ease the problem of displaying large structures such as ontologies and consequently their differences. It would offer another advantages against 2D graph drawing methods, for instance, users could navigate to find a view without occlusions. On the other hand displaying graphs in 3D can also introduce new problems. In spite of the fact implementing a tool capable of displaying ontology difference in three dimensions could be a very interesting task, the final outcome and its usefulness is obscured. We rejected this approach and stayed focused on more practical solutions.

4.3 Summary

For graphical representation of large ontologies one of the best methods is Treemaps. According to [10] Treemap turned out to arguably be the best visualization system overall in that study. It does not surpass the baseline system Windows Explorer though (in that study several graph visualization techniques were compared, in addition indented list method well know from the very Windows Explorer was also included in that experiment). Treemap users excelled both in the correctness of their answers and in their task solution times,

which came close to those of Windows Explorer. There were merely two questions that most subjects were unable to answer correctly. Treemap also received comparatively high user satisfaction ratings, which however fell far short of Windows Explorer. The last sentence is important, it was one of the main reason why we have decided to use the very indented lists in our application. Treemaps as a practise of ontology difference visualization has already been implemented in Prompt-Viz.

Implementing other graph visualization technique would not be useful, other techniques are even less suitable for our purposes than treemaps. Instead, we would like to use graph representation to demonstrate some local particular differences. More about this in Section 8.1 in Chapter 5.

Chapter 5

Our Approach

This is the fundamental chapter of this thesis and focuses on newly proposed ideas and approaches to ontology difference visualization issue. First sections deals with the new concept of *delta tags* and other metrics as an instrument for providing users additional information about differences and changes that has occurred between two ontology versions. The rest of the chapter discusses parallel aligned views with indented lists as an ontology difference visualization technique.

5.1 Brief Outline

We introduce a hybrid or multimodal approach. We decided to exploit and combine the best features from the previously mentioned solutions. Firstly, we keep the idea of two parallel views inspired by Vimdiff, OntoView and other tools. This is a well-proven way of representing differences between two files (possibly between two arbitrary resources), and a lot of experienced users are used to it. So the main window is separated into two vertical subwindows. Each subwindow depicts one version of the ontology. From the Protégé like method we adopt the idea of representing an ontology as a hierarchical tree. According to the survey in [10] this method showed very good overall performance, with regard to correctness, speed of task completion, and user satisfaction (while compared to other visualization systems for tree hierarchies). Users are able to navigate through this tree and expand and collapse single items. This method also offers the contextual presentation of changes (it is probably the best method for it). For example, if a class has been deleted, the user might want to know where in the class hierarchy has the class originally been located, whether it has had any subclasses or instances, what have its properties been like, and so on. In our parallel views all of this is matter of course.

One important question is what is going to be shown in this tree view. The simplest way is to show only classes. This is the example of Protégé PromptDiff plug-in, but this does not suffice our needs, since we would like to visualize the differences concerning properties, instances and property instances, too.

To extend the tree view for showing properties is a straightforward task assuming that the number of properties for a class is not very high. After expanding a particular class its list of all properties is shown altogether with their differences. Representing class instances in this tree view is more difficult due to the fact the number of instances can be very high. In the worst case thousands of instances for a single class. It would be difficult for the user to navigate through such high amount of instances so another way should be adopted.

Color coding like the one used in Protégé is used to help the user better and faster understand the difference.

The last goal that should be accomplished concerns complex changes. Consider this case, in the second version of the ontology one arbitrary class was moved somewhere else in the class hierarchy. Let us say, the moved class was wrapped by another class making the new wrapping class super class of the old one. In this particular case, we would like to present this change rather as a move than a sequence of an *add* and *delete* operations. Such scenarios and their solutions are discussed in Section 5.6.

5.2 Delta Tags: Motivation

As we have already outlined in Chapter 1 we would like to provide users with additional information to help them better understand the difference between two versions.

To motivate this intention suppose one ontology with a class called *person* having a subclass named *employee* and some other subclasses such as *student*, *director*, *author* for example. The subclass *employee* has many other subclasses such as *editor*, *columnist*, *reporter*, *salesperson* and others as shown in the left part of Figure 5.1. Now consider a newer version of the same ontology like the one shown in the right part of Figure 5.1. Notice that all subclasses of the class *employee* have been changed. Some of the existing have been deleted, some of them have been just changed (properties added or deleted) and many new subclasses have been added. In other words, the neighborhood of the class *employee* differs a lot in these two versions.

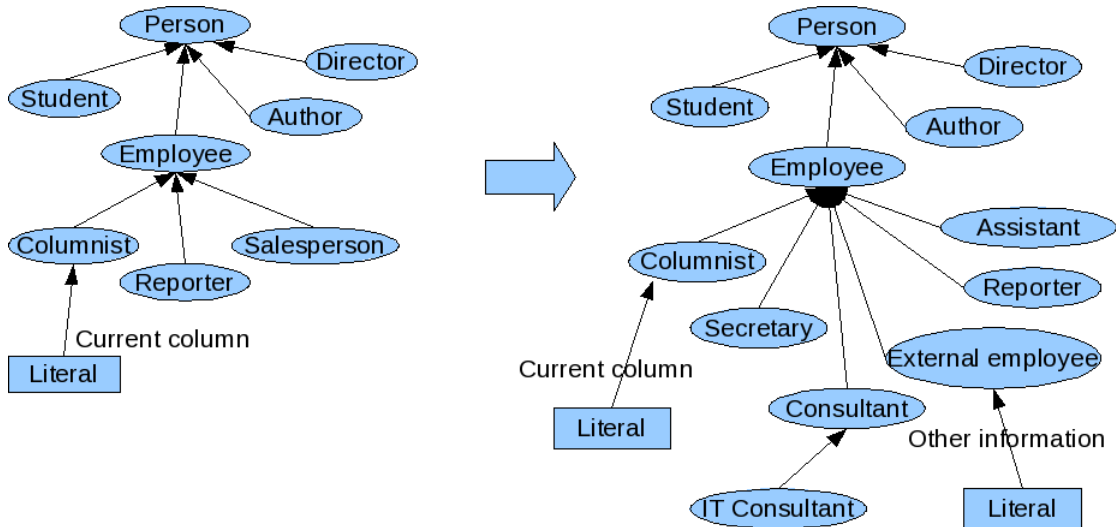


Figure 5.1: Class *Person* with changes. The left part of this Figure represents first version of an ontology containing this class and the right side its newer version with obvious changes.

When users want to compare these versions a new window with two parallel aligned views will appear. The tree views containing ontologies are collapsed (by default) so only the top-level superclasses are visible. In particular, class *person*, *employee*, *author* and other super classes are displayed. Class *employee* and its icon variegated with an exclamation mark indicates one or more subclasses has been changed. There might be some other changed classes in the top hierarchy marked with exclamation mark, too. After first look

the user has no overall notion of changes between the two versions. The only thing he/she is able to see and deduce any conclusion of is the top-level class hierarchy, any further deduction would necessitate exploring subclasses. Users are not able to answer for example following question “How much of the ontology has changed?” Answering such questions would require expanding and scanning all nodes in the tree views. This might be annoying task.

To tackle this issue we propose *delta tags*.

5.3 Delta Tags

We identify and provide metrics for several aspects of the compared ontologies’ elements. For each element, we provide both *absolute* and *similarity* metrics. The former corresponds to the number of change operations that the result of a comparison function (delta) would contain and involve the element at hand. The latter quantifies the similarity in a [0,1] scale in order to allow users to estimate how much the element differs. Questions like how much or how many percent of the class, property or instance have changed should be then much easier to answer.

Table 5.1 describes the applicability of metrics. Specifically, the left column lists seven aspects (of metrics). The cells of the table indicate to what kind of resources each aspect applies (e.g. the aspect *instances* is not applicable to Class Instances and Property Instances).

Aspect	Class	Class Instance	Property	Property Instance
<i>sup</i>	•		•	
<i>sub</i>	•		•	
<i>propsFrom</i>	•	•		
<i>propsTo</i>	•	•		
<i>props</i>	•	•		
<i>instances</i>	•		•	
<i>classes</i>		•		•

Table 5.1: Applicability of Metrics

For each metric we want to provide an absolute version of the metric where we count the cardinality of the symmetric difference, and a similarity metric based on the Jaccard similarity coefficient. For instance, suppose that e is an element of K and e' the corresponding element in K' . If the compared elements are class instances or property instances then the *classes* aspect is applicable according to Table 5.1. The absolute and the similarity metric for this aspect are defined as follows:

$$\Delta_{classes}(e) = |classes(e) - classes(e')| + |classes(e') - classes(e)| \quad (5.1)$$

$$Sim_{classes}(e) = \frac{|classes(e) \cap classes(e')|}{|classes(e) \cup classes(e')|} \quad (5.2)$$

Now we will describe computing single delta tags in more detail, primarily the *propsFrom*, *propsTo* and *props*, whose meaning deserves better explanation. Obviously the *propsFrom* metric applicable to classes and class instances designates similarity of a class or an instance according to its properties (or slots). For classes it is being computed the same way as the *classes* metric, so we will skip it and take a look at computing this metric for class instances, which is not that straightforward.

This time we cannot take only instance's properties and create the union and intersection and count the Jaccard similarity coefficient. In comparison with classes there are two major differences that need to be considered. First, instance's properties have their values (they usually reference another instance or the value is literal as string or number) that need to be taken into account. Second, we must implicate in our computation cardinality of each property, which is either *single* or *multiply*. To sum up, the resulting similarity in term of properties will be computed as following:

$$common(e) = props(e) \cap props(e') \quad (5.3)$$

$$Sim_{props}(e) = \frac{|common(e)|}{|props(e) \cup props(e')|} \quad (5.4)$$

$$Sim_{propsInsts}(e) = \frac{|values(common(e)) \cap values(common(e'))|}{|values(common(e)) \cup values(common(e'))|} \quad (5.5)$$

$$Sim_{propsFrom}(e) = Sim_{props}(e) * Sim_{propsInsts}(e) \quad (5.6)$$

where

- $props(e)$ returns set of all instance's properties (only properties, not their values).
- $common(e)$ returns set of all properties the two instances have in common.
- $Sim_{props}(e)$ computes the similarity between the two instances only according to their properties (property values are not included).
- $values(x)$ returns set of all property instances, where x is set of properties to take into account.
- $Sim_{propsInsts}(e)$ computes the similarity between the two instances according to their properties' instances, but only properties and their instances both class instances have in common are included.

Another delta tag to be clarified is the *propsTo* similarity metric. According to table 5.1 this metric is applicable to classes and class instances. The quintessence may not be obvious from its denomination. In the final application *propsTo* metric is also referenced as *range* similarity, which may be more self explanatory. First, we are going to explain it for the case of classes, but the essence is the same for instances, too. By *propsTo* we define similarity between two classes according to properties, where this particular classes occur as field values (range). Each property in an ontology has its range, this range can be a class, this way the property's value is a reference to an instance of such class, or literal, where the values can be numbers, strings etc. In the first version class *Employee* can occur as range in four properties (like *responsible_for*, *superior* and two more). But in the second version, the property *responsible_for* has been deleted or its range has changed to class *Author* for example. In this example the final *propsTo* similarity of the class *Employee* would be 0.75. The terminal computation is the same as for *classes* so we will not place it here.

Note that the above aspect/metrics are defined over the direct neighborhood of the elements. Well, except for the *sup*s and *sub*s metrics. The ontology comparator used in the final application can compute these *sub*s and *sup*s aspects for classes and properties taking into account either only direct super/sub classes/properties or all the super/sub classes/properties. Then in the application there can be a possibility to choose between these two options or to use both of them. It is also helpful for users to get an idea of

the changes in the more expanded region. This is especially useful in our way of ontology difference visualization as we look at the ontology from the top hierarchy level. We should provide the user hint at possible changes deeper in the hierarchy, so the changes can not be overlooked. More about this in the next section.

At the end of this section we will supply few screenshots demonstrating utilization of these metrics in the final application. In the figures you might notice different denomination of metrics, the reason is to make them more self explanatory for the users. So for instance, the *propsTo* metric is renamed to *Ranges*.

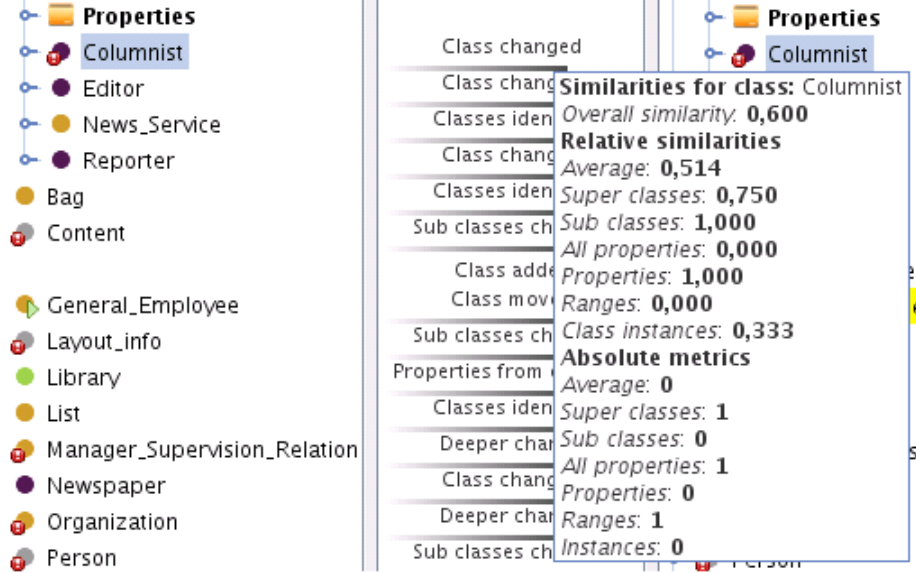


Figure 5.2: In this figure you can observe the similarity metrics computed for the class *Columnist* from the *newspaper* ontology. You can see the *sup*s similarity value 0.750, in the first version this class had three super classes but in the second version it has four. Also in the first version there were two instances and one of them has been deleted and a new one added, that is why the instances similarity is 0.333. You can notice two new metrics, namely *overall* and *average*, they will be explained later.

5.4 Global Neighborhood Metrics

Consider that $a_1 \in U_1$, $a_2 \in U_2$ and $m(a_1) = a_2$, a_1 denotes element from first ontology version denoted U_1 , accordingly a_2 denotes the opposite element from second version of the same ontology U_2 . We can then define:

$$Sim_x(a_1, a_2) = \frac{|x(a_1) \cap x(a_2)|}{|x(a_1) \cup x(a_2)|} \quad (5.7)$$

where $x \in X = \{Classes, Instances, SubClasses, SuperClasses, propsFrom, propsTo\}$. If one of a_1 or a_2 is null then we assume that $Sim_x(a_1, a_2) = 0$ for every x . We can also define the average similarity as:

$$Sim(a_1, a_2) = \frac{1}{|X|} \sum_{x \in X} Sim_x(a_1, a_2) \quad (5.8)$$

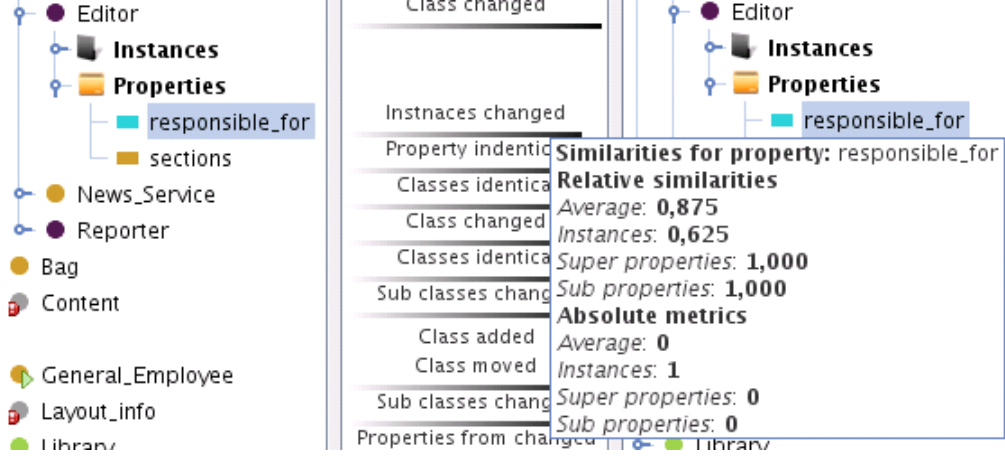


Figure 5.3: In this figure you can see the similarity metrics computed for property called *responsible_for* of the class *Editor* again from the *newspaper* ontology. There you can observe the *instances* similarity and its value 0.625, which indicates instances referenced by this property in different instances have changed quite lot.

or employ a weighed formula.

However the above metric takes into account only the direct neighborhood of a pair of matched nodes. For example consider a class c such that $\Delta_{instances}(c) = 0$ and $Sim_{instances}(c) = 1$. Assume that c has 100 instances in both KBs. However the properties of all these instances could be the same (in both KBs) or completely different (these are the two extremes). Consequently there is a need for metrics that distinguish such cases. We suggest two concepts how to achieve this. The first one is the one actually used in our application, it is a simpler and more straightforward way, satisfactory for our needs, though. The second one was proposed by my supervisor in Greece Yannis Tzitzikas and represents more sophisticated approach, unfortunately there was not enough time to implement and evaluate it. This could be matter of further research. This technique is described in Subsection 5.4.2.

5.4.1 Overall Class Similarity

To define a metric that takes into account the entire graphs of the two KBs, we define a graph Γ by taking the union of the two graphs. Specifically, if $a \in U_1$ and $map(a) \neq null$ then Γ contains one node for both elements (e.g. for both a and $m(a)$). In addition Γ contains nodes for all objects of U_1 and U_2 that match to null.

For every node (class, instance and property) in Γ similarity metrics according to table 5.1 are computed. And also the average similarity is computed as:

$$Sim(a_1, a_2) = \frac{1}{|X|} \sum_{x \in X} Sim_x(a_1, a_2) \quad (5.9)$$

where

- $x \in X = \{sups, subs, propsFrom, propsTo, props, instances\}$ for classes
- $x \in X = \{propsFrom, propsTo, props, classes\}$ for class instances

- $x \in X = \{sups, subs, instances\}$ for properties

For the case of class instances, properties and property instances the aforementioned average similarity is computed plus the particular similarity metrics only.

The new *overall* concept is introduced for the case of classes. Let us outline several cases where the new *overall* metric would be useful. Our *text-based* point of view of the difference is based on classes. Classes are the first thing user can see. No matter where any change between versions occurred he/she should be immediately able to see whether there has been a change somewhere possibly very deep in the class hierarchy. This is the most important reason why to employ this new metric. The user must not miss any change, so every class has to reflect changes of its every direct or indirect sub class. Other good reason has been already mentioned and regards changes in class instances. The *inst* metric from table 5.1 applicable to classes reflects only instances themselves. But what if the instance's properties have changed. The same consideration applies to class properties and their instances or values. Now it is time to explain how the *overall* similarity metric is being computed:

$$Sim_{overall}(c) = \tag{5.10}$$

$$q_1 \times sim_{average}(c) + \tag{5.11}$$

$$q_2 \times \frac{1}{|sub(c)|} \times \sum_{b \in sub(c)} sim_{overall}(b) + \tag{5.12}$$

$$q_3 \times \frac{1}{|propsFrom(c)|} \times \sum_{p \in propsFrom(c)} sim_{average}(p) + \tag{5.13}$$

$$q_4 \times \frac{1}{|insts(c)|} \times \sum_{i \in insts(c)} sim_{average}(i) \tag{5.14}$$

The *overall* similarity consists of four parts. In the first step the *average* class similarity is multiplied by coefficient q_1 . So the *overall* metric reflects all the other class metrics from table 5.1. In the second step we take all direct (it is important that we take only direct sub classes, not all sub classes) sub classes, their *overall* similarity and count new average sub classes similarity, then again multiply it by coefficient q_2 . So it is a recursive function, in the algorithm implementation we start from the leaf nodes (classes) and then propagate the computed values up to higher levels of the class hierarchy.

In the next step we will compute the average similarity of all the properties and instances. In the end our new metric reflects all the necessary information we need. The coefficients used are $q_1 = 0.7$, $q_2 = 0.1$, $q_3 = 0.1$ and $q_4 = 0.1$. For better understanding take a look at the Figure 5.4.

In the figure there is a little demonstration prepared. For the illustration purposes the chosen class is the *newspaper* ontology top class *Person*. The light blue area designated by number one represents parts of the model which similarity computation is covered by computing $sim_{average}(Person)$ using equation (5.11). There are included all the *Person's* sub classes (*Employee*, *Editor*) and properties (*name*, *other information* and *phone_number*). That is all, as you can see this class has no super class, no direct instances and no property has this class as range (*propsTo*).

The upper right orange area marked by number four covers the (5.13) part of the (5.10) equation. The tiny green rectangles stand for property instances.

The big middle grey area identified by number two and confined within blue curved line represents the *overall* similarity for class *Editor*. This value is eventually propagated up to the *Person* class through the class *Employee* using (5.12).

Here we can illustrate what part of the ontology is covered with *overall* metric. As usually the *Editor's* super classes and properties are covered thanks to (5.11), the interesting part comes with the instance *Ms Gardiner*. The instance itself is covered also in the $sim_{average}(Editor)$ but what about its properties and their values? Their similarities are computed using (5.14). So in the final *overall* similarity value *Ms Gardiner* belonging to property *name*, the salary 45 000\$ and also section *Lifestyle* will be included. That does not constitute any new information, it is all part of delta tags from table 5.1 appertaining to instances. At this point the algorithm stops, it does not continue deeper in the hierarchy.

What will be missing are the grey classes and property instances. *Editor's overall* similarity will not be influenced by the fact that class instance *Lifestyle* has different name or some new super classes. But this does not matter at all, since the class instance *Lifestyle* will surely appear somewhere else in the class hierarchy and all its similarities will be computed and presented to users so they will not miss any changes.

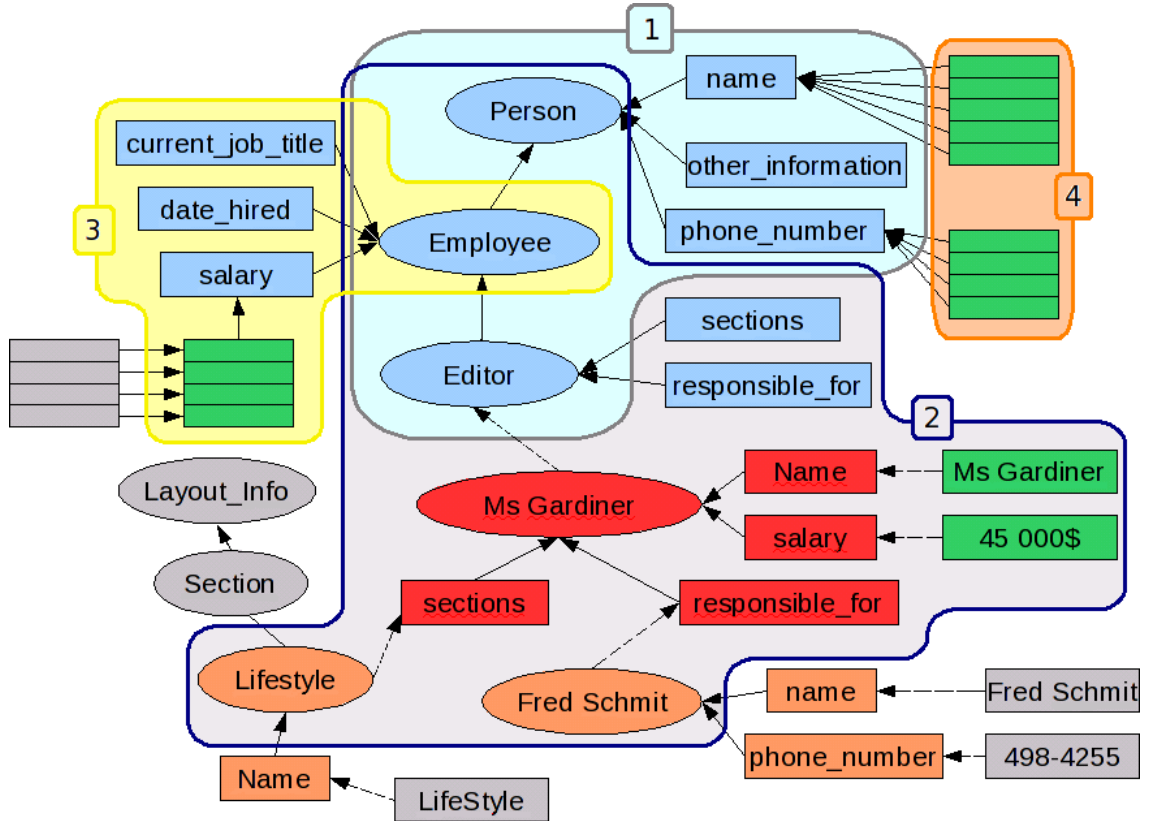


Figure 5.4: Computing the *overall* similarity metric illustration. Various colored areas mirror various ontology parts whose similarity computation is covered with different similarity metrics.

5.4.2 Network Similarity

The idea is that we consider that similarity "flows" in the graph. The similarity attenuates while it propagates and is split (according to the encountered edges). Specifically, we can define the *network similarity* of a node a from Γ (the union of the two graphs) by the following equation:

$$NSim(a) = q_1 Sim(a) + q_2 \sum_{b \in in(a)} \frac{NSim(b)}{|out(b)|}$$

where $q_1 + q_2 = 1$. Note that the constants q_i determine the relative importance. (e.g. if $q_1 = 1$ then $NSim(a) = Sim(a, map(a))$). The $in()$ and $out()$ are defined according the kind of propagation that we desire, e.g. according to the direction we want "similarity to flow". For example, one might want to consider the following rule:

if a *subClassOf* b then $b \in in(a)$ and $a \in out(b)$.

According to this rule the similarity flows towards subclasses (which is quite reasonable). One can easily justify the following default rule: $n_1 \rightarrow n_2$ if one of the following hold:

- n_1 is a direct superclass of n_2 (as n_2 inherits properties from n_1)
- n_1 is direct class of n_2 (as n_2 is member of that class).

Of course a more refined formula with different weights (according to the kind of edges traversed) can be defined. In general, the way $in()$ and $out()$ are defined can be specified also by the user at run time. In all cases, a node n takes high score if it has high Sim value and the elements pointing to n are also similar (and so on). A node n takes low score if it has low Sim value and the elements pointing to n are not similar (and so on). Note that the score of an unmatched nodes is based only on the similarity of the nodes that point to it.

The above scoring can be exploited in several different ways:

- **Tagging:** The NSim value of an element can be considered as a global network-based delta-tag that can be displayed at the GUI as a measure of how similar/dissimilar this element is in the two versions.
- **Ranking:** We could rank the elements according to their NSim value. The most similar element will have the higher values, while the less similar will have the lower values.

The NSim values can be computed by an iterative algorithm, similar to those employed for computing link-analysis metrics, which practically converges if the number of iteration equals the logarithm of the number of edges.

5.5 Parallel Tree Views and Delta Tags

In this section we will finally introduce and discuss details of our parallel tree views approach. As already mentioned in this chapter's introduction, the main idea was inspired by other tools developed for the purposes of displaying differences between diverse resources.

Namely, tools such as Vimdiff, KDiff3¹ and many others. The ontology itself is then textually represented using indented lists.

As you have surely noticed from figures throughout this thesis (or see figure 6.1 with the whole application) the main view is divided into three vertical columns. The first column contains simple tree view filled with resources from the first ontology, then in the middle information panel is presented containing brief guidelines giving the users first sight notion of the particular change between two resources. And then the last column represents the second ontology version the same as the first one does.

All the three panels are synchronized, it means whenever user scrolls one the two remaining are scrolled automatically to the same terminate position, thus enabling the user to easily see the propriate context from the opposite ontology version. Also when expanding or collapsing nodes from the tree views the appropriate node from the other version is expanded or collapsed, too. Again, this facilitates users better orientation in the ontology difference context.

5.5.1 Directed Acyclic Graph to Tree

The first problem needed to be resolved was how to represent ontologies with aforementioned indented lists since these lists are able to display only tree structures. Ontologies create *Directed Acyclic Graphs* (DAG) structure, therefore we need to somehow transform such graph to tree structure indented list is powerful to display. In practise, the only solution coming in useful is to portray classes with more than one super class in more places in the tree, thus reorganizing the directed acyclic graph structure into tree structure. As a result, class with three direct super classes will appear three times in the tree. This is apparently redundant information, although it does not matter as far as we provide the user enough hints and instruments how to handle and be oriented in such situations.

How such conditions are solved? For every class the user is provided additional information disposable in class tool tip windows. One of this information is information about all class' super classes and direct super classes. If a class has more than one direct super class, than it has to appear at least twice in the class hierarchy. Then we grant the users possibility to visit all the class' occurrences in the tree. There is a search tool, where when you input the class' name you will be able to navigate through all the occurrences. In the Figure 5.5 you can see an example.

5.5.2 Displaying Delta Tags

The next task to be sorted out concerned proposing as transparent as possible way of informing the user of all the precomputed similarities (so called delta tags). Many possibilities could be conceived, across all feasible menus, table views up to popup windows.

At last we decided to squeeze in to our parallel view another information strip(panel). This panel is located in the middle of the views displaying ontology tree structures. At first sight the panel shows brief but as apposite as possible description of potential change between two particular resources (classes, instances, properties or property instances). The terminate message shown in this panel is chosen on the basis of all disposable similarity information. This descriptive guide line starts from the most accurate in the best case, and ends with the most general in the worst case. Let us clarify this. When the class change concerns only one observed aspect from table 5.1 then the message describes this

¹<http://kdiff3.sourceforge.net/>

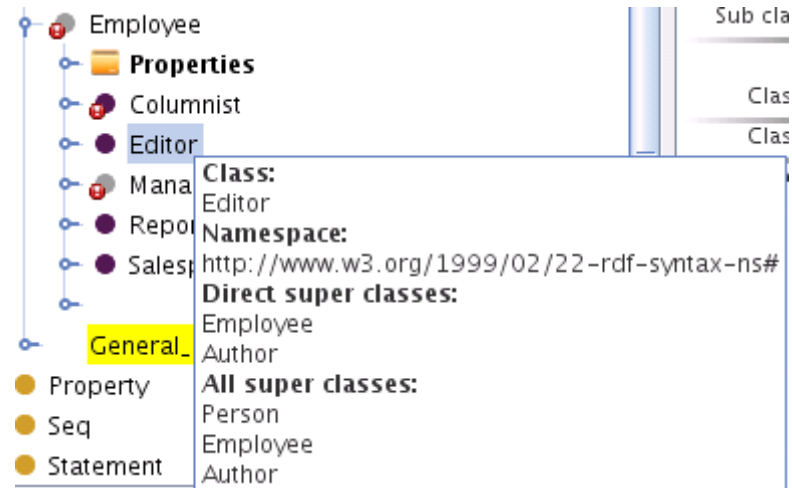


Figure 5.5: Example of a class with more than one direct super class. In this case it is the class *Editor* and the corresponding tooltip informing the user about all its super classes.

aspect. For example, when the class' sub classes have been changed, for instance, one new sub class has been added and nothing else has changed, then the message will say "sub classes changed", thus informing the user what kind of changes to look for. In the worst case, more aspects have changed and the essence of this change cannot be expressed by one simple guide line, then the message will say "class changed".

All the other important similarity metrics are hidden in a tooltip window, that will appear when the user positions the mouse cursor over this panel. Such case has already been shown in Figure 5.2 and 5.3. The middle information panel contains one last feature we would like to mention here. As you may have already noticed from the figures under the most of previously described guide lines a small progress bar is placed. This progress bar reflects the overall similarity metric introduced and described in Section 5.4.1. It is supposed to give the user idea of the overall class change without exploring all the similarity metrics and all sub classes.

5.6 Complex Changes

The simplest ontology change operations on the lowest level are *additions* and *deletions*. When a new resource has been added into second version blank space is left in the appropriate opposite place in the view representing the first version. The same situation arrives when a resource has been deleted from the first version. Our tool is capable of identifying and presenting such changes to the user. Although, tracking only such primitive changes would be inefficient.

Pretend the following example, in the first ontology there is a class hierarchy which has about ten top level classes, classes with no other super classes. Then in the second version the ontology itself has not been changed at all, but new super class wrapping all the previous top level classes has been added. What would happen in our parallel aligned tree view, if we did not identify such complex change operation and provide the user only list of deletions and additions. The result would be misleading, after opening the window with comparison results we would see something similar to what is shown in the Figure

5.6, the user would see the newly added class aligned with the top classes from the first version. All the former top classes would be guided as deleted (or deleted on that level) and after expanding the new class *Newspaper_Ontology* the same classes would appear under it guided as added. Such behaviour would be unarguably ineligible.

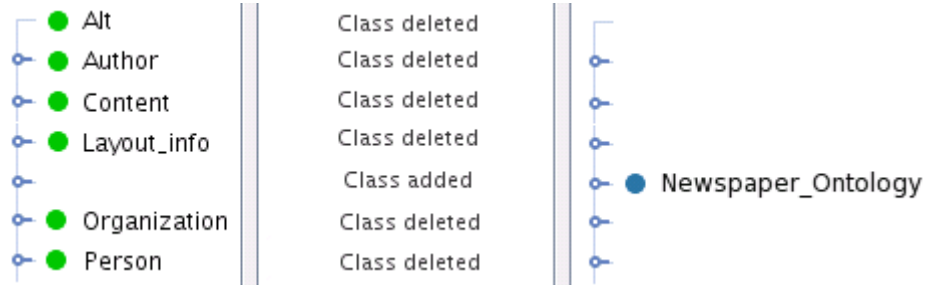


Figure 5.6: Example of difference visualization without identifying moved classes. This is one of the extremes that could happen, the whole class hierarchy from the first version has been wrapped by a new class.

All the previous indicates the need of identifying *moved* classes not as deleted from one position and added to another but as a single complex atomic operation. This is nothing new and all the other ontology difference visualization systems offer such feature. Novel is the concept of displaying such moved classes in our parallel view. The elementary approach would be to show blank space opposite the moved class in the other parallel view. But this way user would loose the possibility to easily compare such classes since for all the moved class' sub classes blank space in the opposite view would appear, too. A straightforward solution to this problem presents itself. We can temporarily show the moved class aligned in the opposite view marking it distinctively so the user would immediately recognize the class is not really located in that place in the ontology, but serves only as a hint to provide all the comparison features available for other changed classes. The final application offers the user possibility to switch on/off showing such moved classes in the tree views. In the Figure 5.7 you can observe utilization of this feature.

In addition, resources can be renamed in newer ontology version. Identifying renamed resources is not an easy task. Every RDF resource is identified by a URI in the corresponding RDF file. When a resource is renamed, its URI is changed and there is no exact way how we could distinguish such renamed resources as equals. Although, our ontology comparator and resulting similarity metrics could be further exploited in order to provide users list of resources that could represent the renamed one. Such resources would be chosen based on their similarity metrics and then it would be up to the user to choose definitely the one renamed resource.

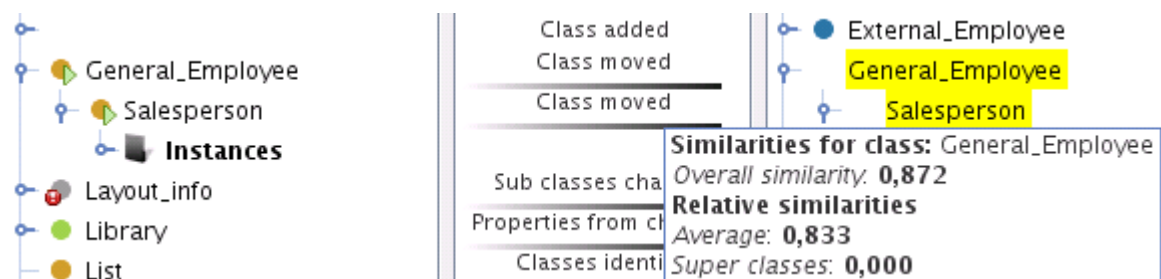


Figure 5.7: Example of displaying aligned temporary moved classes. As you can see, the class *General_Employee* has been moved to different position in the class hierarchy in the second version. But to provide the user all the useful similarity metrics, we display this class aligned with the original class at the same line. Notice the similarity metric *Super classes* and its value 0.000, indicating these two classes have no super classes in common, which is obvious. In the first version the class was a top level class with no super classes, on the other hand in the second version the class is located somewhere deeper in the class hierarchy, so it has to have at least one super class, thus the similarity according to super classes equals zero.

Chapter 6

Implementation

This chapter covers implementation details of our work. The main application was developed using *Java* programming language. This was an obvious decision since the SWKM platform (introduced in Chapter 3) and other related software like StarLion¹ or Protégé are also implemented using the same programming language. Different question was whether to start developing the application from the beginning as a brand new piece of software or to use existing software solution and integrate it to it. First possibility that came to our minds was to involve our application into StarLion. StarLion is a visualization tool for RDF schemas. It can load ontologies expressed in .rdfs files or stored at the FORTH-ICS Semantic Web Knowledge Middleware. This tool uses force-directed layout algorithms for displaying ontologies. The expected convenience gained from such approach was the re usage of some existing code, mainly for RDF files loading, RDF model wrapping the original SWKM Model and perhaps few other classes for working with the model classes.

Another possibility was to integrate it into Protégé² platform. Protégé is an extensible ontology and knowledge base editor developed at Stanford University, provides plug-and-play environment that makes it a flexible base for rapid prototyping and application development. You can find many available plug-ins for Protégé aimed for diverse ontology related problems. Starting with simple modeling up to ontology visualization. Among visualization plug-ins you can find TGViz³ or interesting project Jambalaya⁴, this plug-in uses the SHriMP⁵ (Simple Hierarchical Multi-Perspective) 2D visualization technique which is based on a nested graph view and the concept of nested interchangeable views.

At last, we decided to develop our tool from the beginning, only with partial use of some existing code from StarLion. The code used was the RDFModel java package created to wrap the original SWKM Model. Although, this model was highly modified for our needs, many new classes had to be added (to supply work with property instances, literals etc.). Next section gives a brief introduction into GUI design, followed by section describing application's architecture and then another section is devoted to our ontology comparator and its features.

¹<http://www.ics.forth.gr/~tzitzik/starlion>

²<http://protege.stanford.edu>

³<http://users.ecs.soton.ac.uk/ha/TGVizTab/TGVizTab.htm>

⁴<http://www.thechiselgroup.org/jambalaya>

⁵<http://www.thechiselgroup.com/shrimp>

6.1 GUI Design

This section describes graphical user interface design and implementation details. The original design was based on several requirements imposed on this kind of application. The resulting application will not be able just to load two ontologies and display their comparison. Design takes into account possible further extensions concerning more sophisticated support for ontology manipulation (more information in Chapter 8). To give better support to work with more ontologies and use more advanced features, our design was enriched for the projects concept. Every ontology that is about to be opened has to be embodied in a project. You may create as many projects as you want. Under each project you may open as many ontologies as you want. This way user can easily work with many ontologies, explore them and compare them. So the main application window is divided into two parts. On the left, you can find the projects manager, in other words tree view with opened projects and loaded ontologies. From this view you can always choose ontologies to open and explore in a simple ontology viewer. You can also at any time choose two ontologies to compare.

The second part of the main application window creates panel capable of displaying internal frames (internal windows). To each opened project exactly one internal frame is associated to. Every such internal frame contains a *tabbed pane* where every opened ontology or comparison result is being displayed as a new tab. This architecture enables the users to easily work with many opened projects and many opened ontologies. They can switch among the internal frames, minimize, maximize or close them. Design of the main comparison parallel view is shown and described in Chapter 5. In the Figure 6.1 you may observe the final application with two projects created and with five ontologies loaded, altogether. Also corresponding project frames are visible.

6.2 Architecture

The final application consists of three main more or less independent components. This fact is then reflected in the resulting package structure, but this will be shown and explained later. The first component implemented was new RDF model API, as already outlined in Chapter 3, the new model API is intended to wrap the underlying SWKM model API in order to not to use that SWKM API directly thus make the application as independent as possible on it. This goal was achieved by creating two Java packages, the first package `RDFModel` contains interfaces for classes defined in the second package `RDFModel.impl`. The second package comprises all the necessary classes to support work with RDF models, including reading and parsing RDF files, subsequent filling the memory model with data received from those files, thus representing the loaded ontology. The rest of our application uses only interfaces from `RDFModel` package.

The second component implemented is the ontology comparator, we also use *Local Comparator* denomination for it. Also in this case two Java packages are used in implementation, where the first one represents interface for the second one (`comparator.impl`). The comparator has its own package again in order to make the application independent on it. But this is not the only reason, as described in Section 5.4.2 another technique for computing the similarity metrics could be employed. Then there would be nothing easier than make another package implementing the comparator's interface and the rest of our application would be able to work with it with no need to make any further changes. The last thing to be indicated, the comparator has modular structure, it contains several classes for computing all the similarity metrics, separately for classes, properties, instances and property

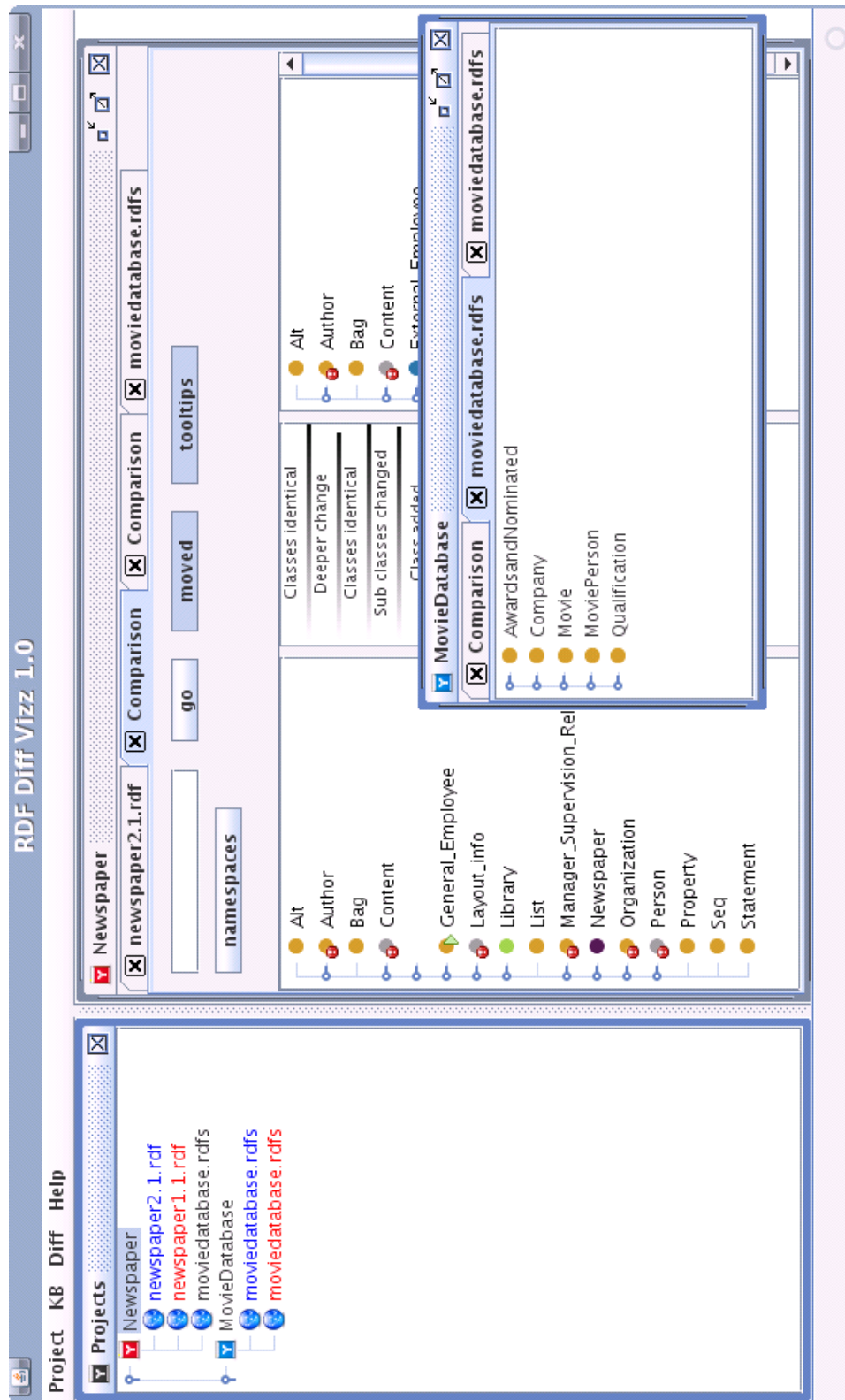


Figure 6.1: The final application is presented in this figure. On the left side you may notice the project manager with two opened projects (the red icon indicates currently active project), on the right side there are projects's frames with different tabs. In the *Newspaper* frame, the comparison view is currently opened, in the *MovieDatabase* frame the ontology viewer is opened.

instances. The main algorithm is again independent upon the implementation and different similarity computers could be provided.

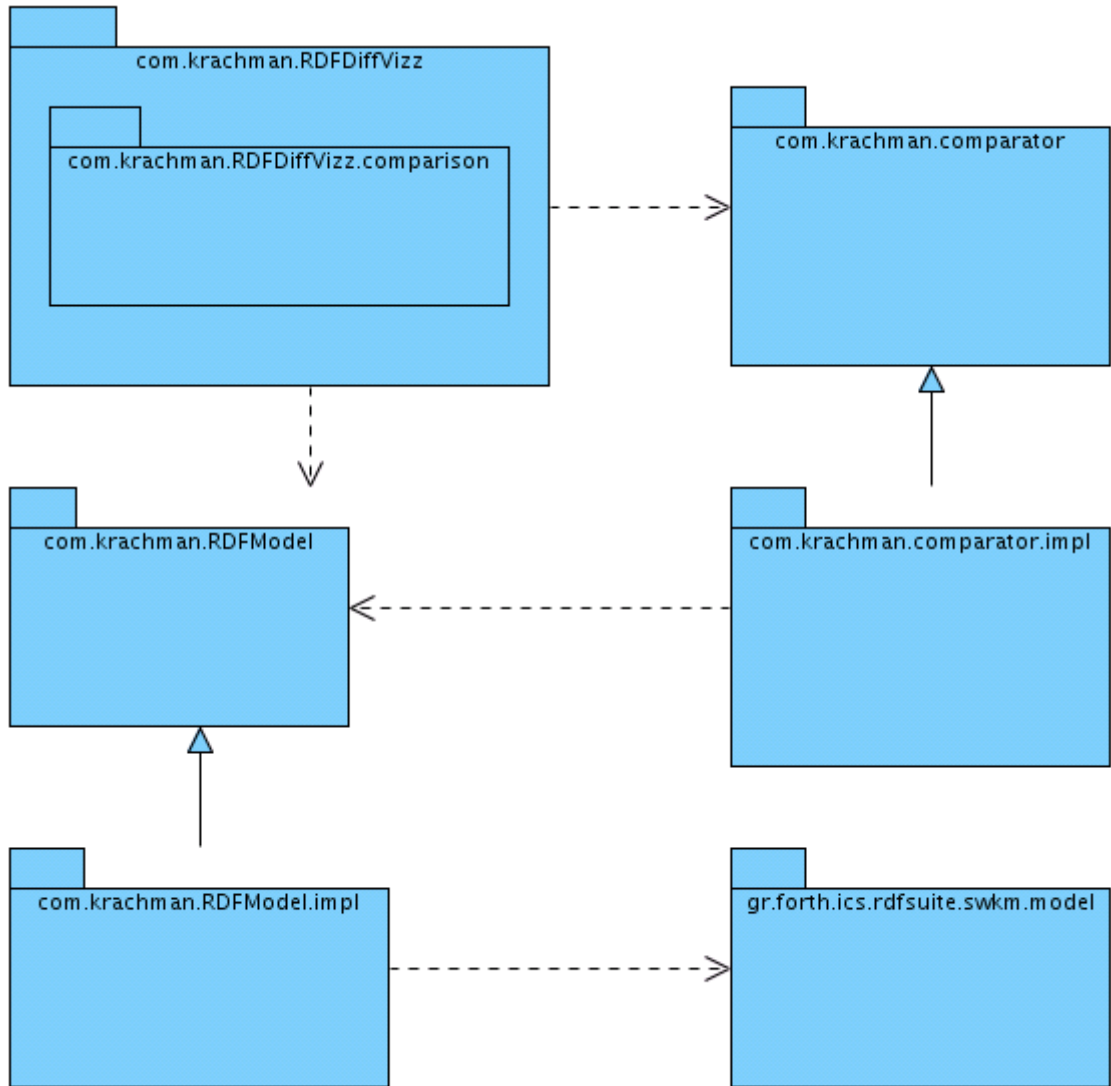


Figure 6.2: Package structure of the application

6.3 Local Comparator

This part describes how the comparator (similarities computer) works. The main algorithm uses BFS (Breadth First Search) to traverse the class hierarchy. In the beginning the algorithm was designed to compute the similarity metrics (delta tags) only for classes that are currently loaded into memory, e.g. classes visible in the parallel views (viz 6.4) in order to make the computation as fast as possible. Every Knowledge base may comprise hundreds of classes and even more instances, thus the previous thought was based on well-founded need. Nevertheless, this idea had to be rejected, due to the fact of introducing *overall* similarity metric (described in 5.4.1). To be able to compute the *overall* similarity all

classes have to be visited and their similarity computed. How this is achieved is described using the following pseudo code:

```

queue.insertSuperClasses ();           // start with all top level classes
leaves = new List ();                 // list of leaves (classes with no sub
                                     // classes , bottom of the hierarchy)
while (!queue.isEmpty ()) {           // while queue is not empty
    clazz = queue.poll ();             // get class
    if (notVisitedYet(clazz)) {        // visited for the first time
        oppositeClass = getOpposite(clazz); // get the same class from
                                     // the second model
        if (oppositeClass != null) {   // if in the second model
            compareClasses(clazz , oppositeClass); // compare classes
            compareInstances(clazz , oppositeClass); // compare instances
        } else {                       // the class is only in the first model
            // do nothing, we do not need to store
            // information about deleted or added classes ,
            // they will be recognized later , when no
            // similarity information will be available for
            // such classes
        }
        if (clazz.hasNoSubClasses ()) { // if the class is a leaf
            leaves.add(clazz );         // add to list of leaves
        }
        queue.add(clazz.getSubClasses ()); // enqueue sub classes
    }
}
compareProperties ();                 // properties are compared apart
// at the end propagate the computed similarities to higher
// levels in order to compute the overall similarity
for (clazz in leaves) {
    startBackPropagation(clazz );
}

```

6.4 Lazy loading

Dimension of a particular ontology may differ a lot. There are small ontologies comprising tens of classes, but you can also find knowledge-bases with thousands of classes and even more instances. Loading such huge amount of data would be inefficient. Our tool uses so called *Lazy loading*, it means the class and its content is being loaded into memory only when it is really needed. Well, this is not literally precise, the application loads two levels at once, the actual level being displayed is loaded plus one direct under it.

Chapter 7

Experimental Evaluation

We conducted a preliminary user study to test both the functionality and usability of our approach. Before our study, our hypothesis was that our solution could be more intuitive for the users but still provide enough or even more functionality to help the users to orient in the ontology difference and easily answer related questions. We also expected, that the view method based on indented lists would prove good overall performance, with regard to correctness, speed of task completion and mainly user satisfaction as it has already proven for this technique in other comparison studies [10]. In addition, the effectiveness of various visualization techniques for answering these questions was also of interest.

The following sections describe in detail the participants, the procedure of the study, the tasks and the results of the study.

7.1 Subjects

The user study was performed by the author and involved seven students. Three of them were graduate computer science students at this University. The rest four non computer science students from other universities. None of the students had any substantial experience using tree visualization tools. Everyone was familiar enough with indented lists as they appear in many applications for everyday usage, nevertheless none of them ever heard of Treemaps or used them before this study. Prior to main evaluation each of them was briefly introduced to concepts of ontologies and ontology difference, also the essence of Treemaps, their utilization and advantages were explained to the participants. The tool's order of appearance in the consequential experiment was chosen at random so each tool was used as first, second and third respectively at least once.

7.2 Evaluation tests

For the purposes of evaluation we provided the participants two versions of one ontology and ask them to examine the changes based on a set of provided rules.

Prior to testing particular tool every subject was given a short verbal explanation how to use the tool and simple guidelines how to solve the tasks. In the case of our tool brief tutorial was provided directly in the application and the subjects were allowed to use it any time they need. For the own experiment the author's laptop with all three tools installed was used. The author was also present for the whole experiment to be able to observe the subjects while they were performing given tasks and to help them with any possible

questions regarding the usage or concept of ontologies. We wanted to see how users would use our tool, what they start working with and how they exploit all the supplied features. However, we did not interfere in order to keep the results neutral.

The ontology used in our evaluation is the *newspaper* ontology. It is the same ontology mentioned many times throughout this paper. For example, in Section 4.1 it was used in our conducted *vimdiff* experiment, where in Figure 4.2 you can observe the results. In Figure 4.3, 4.4 and 4.6 you can examine it again this time processed by various Protégé plug-ins. Although, for this final experiment both versions of this ontology were slightly more modified to comprehend as much various differences between the two versions as possible.

Ultimately, the first version contains thirty-five classes, fifty properties and sixty-six instances. On the other hand the second version being compared comprises forty classes, fifty-six properties and seventy-one instances. So more or less one can say this represents middle size ontologies and since also class instances are present this one ontology should be enough for the whole evaluation.

7.3 Compared tools

The goal of this user study is to compare our new approach and on its bases build tool with existing solutions for ontology difference visualization purposes. We decided to compare our tool to Protégé plug-in PromptDiff[11] and to its visualization extension PromptViz[13] as these are also intended to visualize and represent changes between versions of ontologies. The first plug-in as well as our tool uses textual representation, while the second one is a representative of *graph* based views.

Another reason why the fore mentioned Protégé plug-ins were chosen for the evaluation study is that they represent two different approaches that have already been compared in another user study in [13] and it would be interesting to compare them to our tool. After all, there are not many other tools capable of displaying ontology difference that could be used in our evaluation. Convenient fact is that these three tools (including ours) represent three thoroughly distinct solutions to ontology difference visualization problem.

Protégé version used in this experiment was Protégé 3.1. This is an older version than the currently available 4.0 beta, but it was also the latest version with working Prompt plug-ins we found.

7.4 Tasks and questions employed

We prepared several task and questions for the participants. These tasks and questions were often inspired by other surveys (like in [10], [13]) already conducted and thematically related to ours.

These questions and related tasks were provided to the subjects printed on several questionnaires. The questions were also read to them verbally and explained in more detail. The subjects were required to fill all their answers in the questionnaires. Subjects were instructed to abort unsolved tasks after five minutes and to note down this fact.

Altogether, there were three questionnaires prepared for the subjects to fill in. The first questionnaire (pre-study questionnaire, see Appendix A) contained questions regarding the subjects' experience with Treemaps or ontologies, also the question whether they are computer science students or not was asked. The second questionnaire (see Appendix C) included the main tasks related to ontology difference visualization. The final set of tasks

to work out comprised ten questions. These tasks were designed to cover several aspects we would like to examine and find out how our tool stands in solving such aspects compared to other applications. The aforementioned aspects could be divided into two following groups.

- Couple of questions were focused on the main features implemented to support work with ontologies, such as providing enough information about the topology, all the necessary information for each resource (class, instance, property or property instance), ability to search for resources and others. All the compared tools provide such features, but each one uses different approach, so we wanted to verify that our tool handles the general work with ontologies as well as or even better than the other tools. For the previous reasons solving few tasks requires seek a resource in the hierarchy, or locate all the resource's occurrences. Elaborating on other tasks necessitates finding diverse information about resources. Examples of such questions come next:
 - Find the name of the parent class of the class Columnist. If there is more than one parent class write down all of them. First, write the parent class/classes in the first version, then in the second version.
 - Write down the original class path for the class Small One. Start writing from the top super class down to the direct super class. Do the same in the first and second version.
- Secondly, the general ontology difference visualization and presentation capabilities of our approach had to be attested and compared. That is why almost every question concerns the difference between versions. The tasks are designed to cover up as most different change operations as possible. In the questionnaire you can find questions on class changes, property changes, instance changes, moved classes and others. Again, preview of such questions comes next:
 - How many new employees (instances of the class Employee) have been added and how many have been deleted?
 - Find class that has changed the most. Including changes between class properties, instances and class itself. By "changed the most" we do not mean class that contains the most changes (change operations) but class that is the least similar to its equivalent in the other version.

In the end of tasks completion, further inquiry into the subjects' opinion of the tool was gathered through a post-study questionnaire (see Appendix B). This part was aimed at receiving additional information about the subjects' satisfaction with using our tool and also at obtaining feedback and ideas of possible improvements from our subjects. All the conclusions gained upon this questionnaire are summarized in the few following sections.

7.5 Results and discussion

Our evaluation experiment was not ideal, the number of subjects should have been higher to get more meaningful estimates whether the tool really is better and it would have been more convenient if the subjects would have been experienced with working with ontologies, for instance, Protégé users or other professionals who work and design knowledge bases. Nevertheless, this intention could not have been accomplished. Though, we think the outcomes from this experiment are useful and helped us to review our ideas and suggestions.

Next part discusses results gained from the post study questionnaire. The rating concerned four aspects. These aspects are identical to those used in a similar user study conducted in [13].

- Overall ease of use
- Ease of learning
- Intuitive to use
- Informative (The visualization and controls were useful)

The subjects were instructed to rate each aspect with the grade from one to five, where one is the worst and five the best grade. This grade could also be comprehended as points, then in the final each application could gained the maximum of thirty-five points for each measurable aspect because seven subjects were involved in the experiment. In the Figure 7.1 you may observe the results represented with bars chart. There are three clusters, each supplies particular tool compared. The first one is our application (RDFDiffVizz), then PromptDiff and PromptViz plug-in. Each color mirrors one subject and grades (points) gained from this subject.

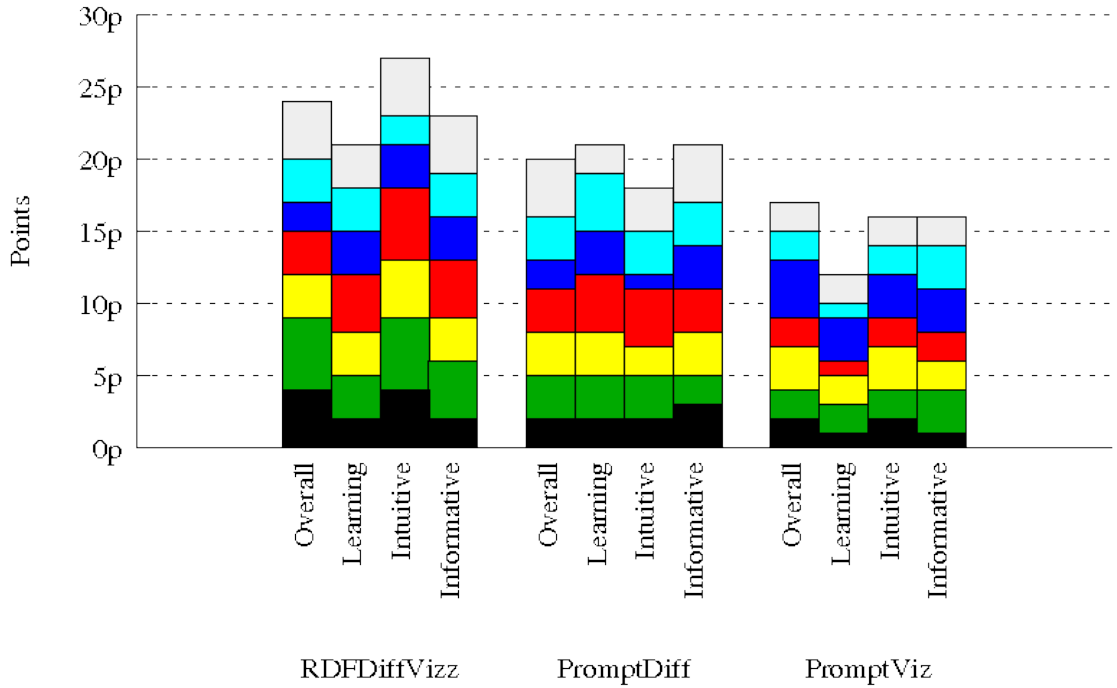


Figure 7.1: In this figure there is a chart representing final rating gained from the post study questionnaire. There are three clusters, each supplies particular tool compared. The first one is our application (RDFDiffVizz), then PromptDiff and PromptViz plug-in. Each color mirrors one subject and points gained from this subject.

Piece of knowledge gained from the user study are summarized in the following sections.

7.5.1 User satisfaction

As you can reproach from the graph in Figure 7.1 our application competed well with the other solutions. The overall aspect obtained twenty-four points which exceeded our expectations. Let us focus on every single evaluated aspect.

The learning aspect. Purpose of this aspect was to compare the tools according to learning time and difficultness. The resulting score was equal to score of PromptDiff, this was not surprising, since both tools offer similar functionality. Both exceeded PromptViz in this category which was also expected, even in the original study comparing PromptViz and PromptDiff the users gave dissatisfied ratings to PromptViz for the ease of learning and intuitiveness. This proves, that Treemaps and other similar techniques are not user friendly to most users and approaches based on well known indented lists acquire much better satisfaction.

The next aspect, intuitive to use was the most tracked one, since our application was designed to be as intuitive as possible, especially the two parallel aligned views were expected to help users better navigate and understand changes between ontology versions. All of this was acknowledged by subjects, they found these parallel views much transparent than the one view combining both versions used in PromptDiff, especially for questions regarding class paths or the class structure the PromptDiff view was found confusing. The last category, informative concluded similarly for all compared tools, our tool slightly won. It provides more useful information than the other tools.

7.5.2 Task completion

Ten task were designed to examine features of all three applications. All the task can be found in Appendix C. The first question was difficult for subjects to answer using our application and PromptDiff. This is obvious, determining how much of the ontology has changed (in percent) would require expanding all nodes and examining every class, property or instance. This could be accomplished for small ontologies, but not for large ones. Even for our test ontology this was time consuming task. PromptViz offers additional information that helps to answer the aforementioned question. In every rectangle representing a class, number of change operations (additions, deletions, movements...) is shown using a bar. In our application the overall similarity metric for top level classes could be used to distinguish how much of the ontology has changed without the need of expanding every node, but this ontology similarity could not be expressed in percent. This led to proposing an extension in the form of another parallel column showing number of change operations, for more detail see Section 8.2.

Answering the second question was difficult in PromptViz, there was no possibility to look up all the occurrences of the class *Columnist*, using the other tools subjects answered this question correctly.

The third question was difficult to answer in all applications, in PromptDiff this is almost impossible, it would require exploring every class, every property, every instance or property instance. In PromptViz, thanks to aforementioned bars with number of change operations, it could be easier, although nobody answered this question correctly. Our tool did not perform well in this task, either. Nevertheless, two subjects were able to find the class we were looking for. The overall similarity metric was suitable to complete this task, but it required expanding all class nodes and then comparing the similarity. Solution to this, could be some sort of table view to all computed similarity metrics, where the users

would be able to look for particular similarities, order classes according to their similarities etc.

The fifth question concerning type of an instance was answered correctly only in our tool, because there is special information about all types for each instance. Questions concerning instances were ignored when working with PromptViz, since this tool does not support displaying instances or properties and their changes. Answering other questions resulted in similar problems and pointed out to no new findings, so they will not be described here. The last think to mention in this chapter is properties comparison support offered in our approach. In other tools it was impossible to compare property *name*, to be more specific, it is possible to compare properties in PromptDiff, but no subject solved this task correctly.

7.5.3 Feedback

This part discusses proposals gained from subjects and observations made while watching them work with all three applications. Some subjects found buttons for expanding and collapsing all nodes (at least class nodes) missing. This was due to the fact, some tasks required expanding and examining all classes. In real situation, such task would probably not be employed and what is more in the case of large ontologies covering thousands of classes, this would be infeasible (expanding all class nodes). One subject suggested expanding and collapsing class nodes only for each hierarchy level. For instance, in the beginning only top level classes are visible, this function would then expand only this classes, so the first sub level would become visible. Few subjects also suggested, that tooltips showing detail information about resources could be implemented using separate windows, so one could open this additional information of several resources and then compare them.

Another important feature was already mentioned in the previous section. It regards information about number of change operations for each resource. This number would be sum of all change operations that concern this resource and all change operations of each sub resource (sub classes, for example). Such additional information is available in PromptViz and was found very useful.

7.5.4 Results conclusion

This evaluation study proved our original hypothesis, that our solution could be more intuitive for users but still providing them enough functionality to help orient in the ontology difference. Also we wanted to test the overall functionality and usability of our approach. Finally, we can state, that the main hypothesis was proven. Subjects found our tool the most intuitive and simplest to use and learn how to use. The study tasks were not designed to utilize our similarity metrics, since the other compared tools do not offer anything similar and it would be then improper to compare them. Although, in few questions users exploited these metrics. Similarity metrics were useful for orientation in changes, they can tell whether to explore sub classes or super classes etc.

Chapter 8

Further research

8.1 *text-based* views with local *graph-based* views

Another extension or approach we suggested but there was not enough time to implement it so it remained a matter of further research is to provide 2D layouts. We could exploit the layout services provided by StarLion¹[17]. As already mentioned in Chapter 6 StarLion is a visualization tool for RDF schemas. It offers configurable force-directed layout algorithms (appropriate for semantic networks).

Motivation The main idea was motivated by the fact our solution offers *text-based* point of view only. Why it was chosen and why it is probably the most convenient approach is discussed in Chapters 4, 5. On the other hand *graph-based* methods have some advantages that we would like to put to use. The principle consists in the possibility to visualize the ontology difference at every class hierarchy level. So this *graph-based* view would not be aimed at the overall difference visualization, for such purposes one could use tools like PromptViz [13]. Instead, it would be beneficial to enable the user to visualize only some local parts of the ontology. Below we suggest the following approaches:

- **Three-colored Merge approach** The two KBs are merged into a single model which is visualized using the visualization features provided by StarLion. Specifically, the visualization requires at least three colors: one for the common elements (e.g. those in $K \cap K'$), one for those elements that belong to $K - K'$ and another for those that belong to $K' - K$.

The user can see that merged KB entirely, or he/she can navigate using star-graph view browsing.

- **Triple-frame Star-Graph View Browsing** A variation of the above approach is the provision of star-graph view approach where the visualization frame is divided into three vertical areas. The central depicts the focal node e_f plus all directly connected nodes that belong to both KBs. Those elements of the neighborhood of e_f that belong only to K are shown in the leftmost frame, while those that belong only to K' in the rightmost. An example where the focus is the concept *Manager* and the radius of the star-graph view equals 2, is shown in Figure 8.1.

¹www.ics.forth.gr/~tzitzik/starlion

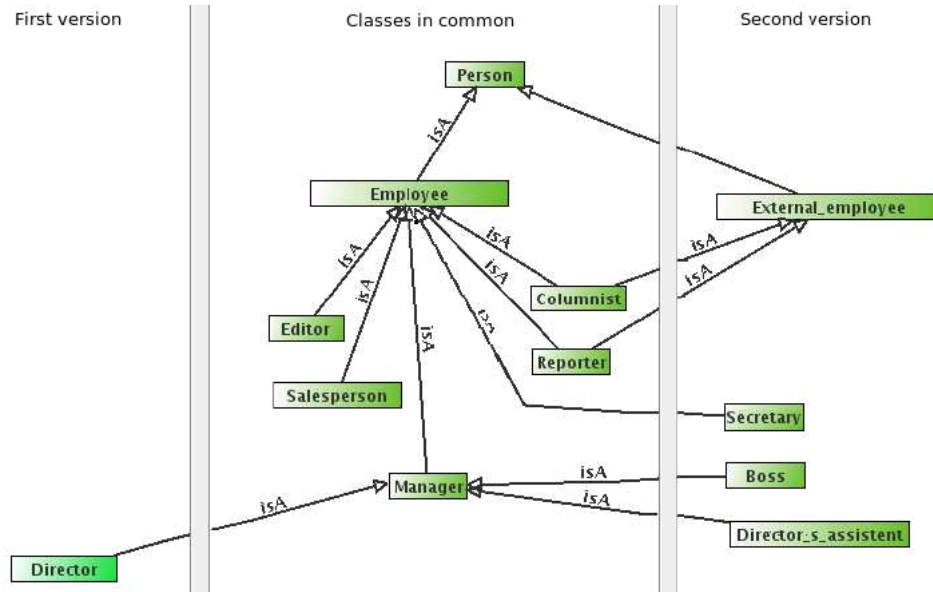


Figure 8.1: Local *graph-based* visualization. This is visualization focused on the concept *Manager* from the *newspaper* ontology.

8.2 Navigation among changes

Changes may be hidden deeply in the class hierarchy, the user should be provided some sort of navigation feature, that would help him/her to track the changes easily. Such tool would automatically navigate user to the next change, even when it is located in a "remote" part of the ontology. There could be two buttons placed in the main comparison control panel, the first one designated *next* change and the second one *previous* change.

8.3 SWKM Middleware

This whole application could be further extended towards a tool supporting work with already mentioned SWKM platform. This platform ensures parsing, storage, manipulation and querying of Semantic Web Knowledge bases represented by RDF models. Two main features of SWKM could be exploited and included to our application:

- The first extension that comes to our minds would be support for loading and storing knowledge bases from databases through SWKM. Using our application you are able to load KBs only from locally stored files, empowering the ability to load RDF models from databases would extent the area of ontologies that could be compared directly using our tool.
- Not only the storage mechanism would be beneficial to cover up in possible extensions. Another vital feature of SWKM platform is the *Diff* service capable of computing the difference between two ontology versions. This service produces set of Triples in the form of subject-predicate-object. These Triples may be used to transform one ontology version to another newer version. In our application this could be exploited as following.

Pretend such user case scenario. The user has two versions of one KB. First he/she opens our application, loads the necessary models and then compares them using our comparator. The result is subsequently displayed in parallel aligned views. Then he/she decides to compute the difference between these version using aforementioned SWKM Diff service. He/she would be then able either to explore the resulting set of Triples transforming one version to another, or keep working with our comparison view. This view, specifically, the middle information panel would be enhanced with new additional information as shown in Figure 8.2. Under each number denominating the actual number of change operations (deltas) list of all the corresponding Triples would be hidden. The user would be then able to explore this list and modify it, thus adapting the difference between versions. After every change he/she would do with these Triples, subsequent alternation would take place immediately to help to decide whether to keep it or return to previous state. After all necessary modifications are done and the result is satisfactory, then confirmation of foregoing operations comes to turn, transforming the first version to the second once and for all. The result could be then saved into database again through SWKM platform.

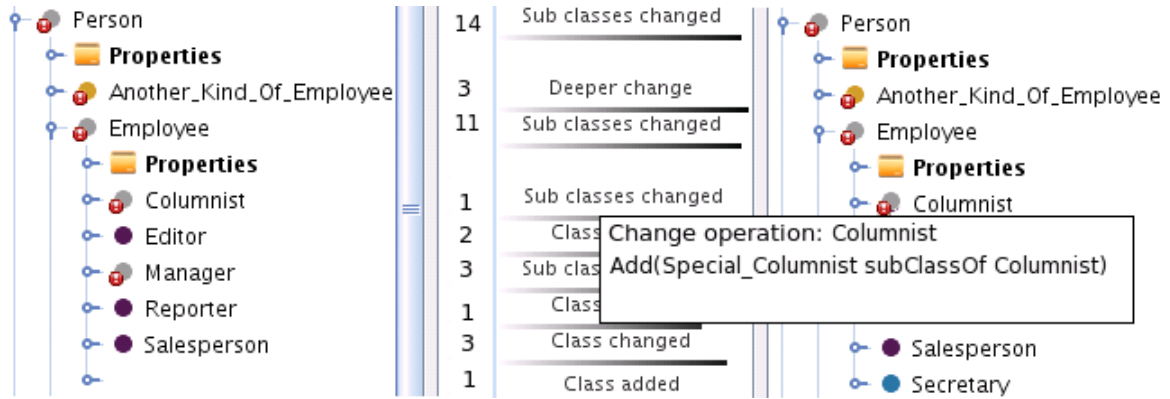


Figure 8.2: This figure shows possible exploitation of the SWKM Diff service in our application. Notice the middle information column and digits aligned on its left side. These digits represent actual number of change operations assigned to particular nodes (classes). The class *Columnist* has one new sub class added, this is mirrored with the tooltip window showing this specific operation.

8.4 Manipulating ontologies using Triples

As already suggested in Chapter 3 RDF API provided by Semantic Web Knowledge Middleware supports aside from Object-oriented Model also Triple-based Model for main RDF manipulation. These Triples were mentioned in the previous Section 8.3 when SWKM Diff service result set was discussed. Such Triples would offer easy way for ontology manipulation. All we need is to provide the user methods for creating such Triples and then let the SWKM to process them, thus applying the changes to the ontology. The question for further research is whether users would understand and find working with these Triples easy and useful.

8.5 Instances

One of the problems whose resolution remained for further research is the way how to visualize higher number of class instances in our parallel tree views. By "higher" number we mean thousands of instances under one single class. How is this issue solved in other tools? In PromptDiff Tree view displaying the difference between ontologies, instances are listed and alphabetically sorted in a simple list. Next to this list there is a table with all the instance's properties and their values (also differences among property values between the two versions are displayed in this table). In the case of an ontology containing thousands of instances, this list would contain thousands of instances and it would become bothersome for the users to navigate through it despite the fact, there is a simple search tool. In our application the situation is the same. We also list all the class instances alphabetically sorted one after another and in the case of a huge ontology with many instances the resulting usability would be similar. PromptViz plug-in does not support displaying class instances, their properties and the differences among them at all.

One solution to such problem that came to our minds is following. Class instances could be grouped into several groups according to their labels. For example into twenty-six groups where each group would embody instances whose label starts with that letter. Such grouping would fail when all the instances' labels started with the same letter, so more sophisticated algorithm could be employed. How this whole idea could look like is shown in the Figure 8.3. The same solution could be also applied to classes and class properties.

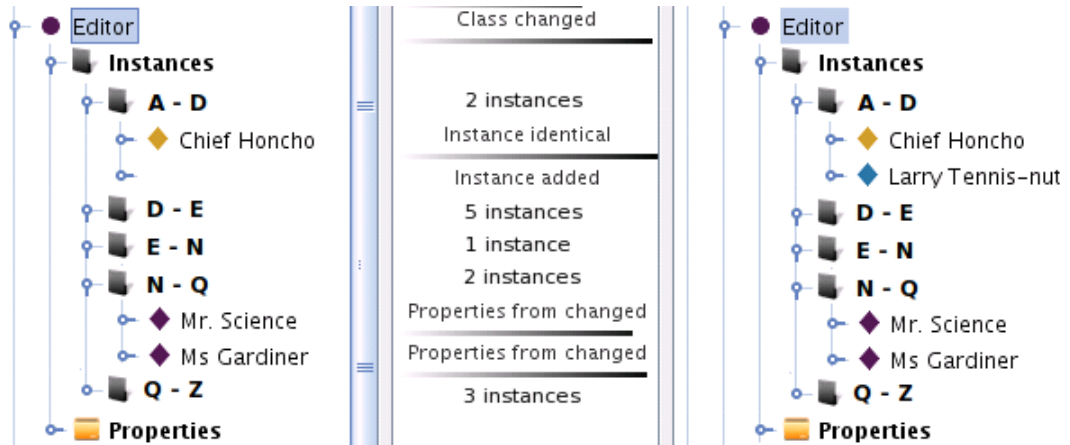


Figure 8.3: This figure demonstrates how the problem of displaying huge number of instances could be solved.

8.6 Fish-eye view

The last instigation for further research mentioned in this paper is the idea of employing so called *fish-eye* view in order to help users better orient and navigate in large ontologies still using our parallel tree views. This method belongs to *focus+context* techniques group. This practise is usually used combined with diverse graph drawing algorithms, usually with node-link (4.2.1) algorithm. Focus+context techniques has already been implemented in several

applications coping with ontology visualization. For instance, StarTree², TGVizTab³ or 3D Hyperbolic Browser, tool that can handle more than 20,000 nodes and is very effective for a representation of a large graph on a small space, as it uses distortion to provide focus and context.

A fish-eye camera lens is a very wide angle lens that magnifies nearby objects while shrinking distant objects [14]. It is a valuable tool for seeing both "local detail" and "global context" simultaneously. This fish-eye technique could be used in our application as shown in the Figure 8.4 where you can see the nodes (in our case it would be classes, properties or instances) that are in focus are displayed with normal font size while nodes further from the focus are displayed using small font size. Usually the further the node is situated the smaller its size is.

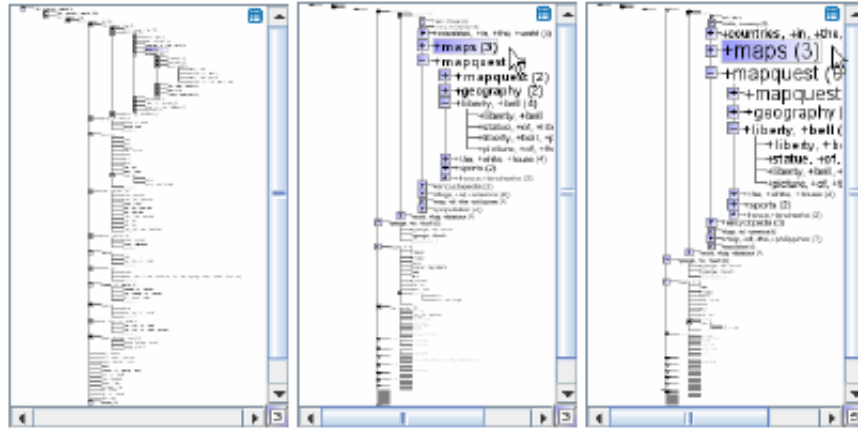


Figure 8.4: Example of fish-eye view technique applied to ordinary tree view (for instance, JTree component as used in our application).

²<http://www.inxight.com>

³<http://users.ecs.soton.ac.uk/ha/TGVizTab/TGVizTab.htm>

Chapter 9

Conclusion

In this thesis we analyzed the need for visualizing the difference between two versions of ontologies, or knowledge bases in general. We compared several existing approaches and tools not necessarily intended for ontology difference visualization purposes, and distributed them into two main categories: *text-based* and the *graph-based*. Each category was then put through inspection in order to pick its advantages and disadvantages and based on the results we attempted to combine the benefits and propose new approach. Specifically, we proposed delta-tagged parallel aligned tree views where textual representation of the ontologies is employed for viewing and exploring the comparison results inspired by other *diff* tools generally used for comparing text files. The aforementioned delta tags are then further explored and discussed. As we are aware of, our tool is the first attempt that uses parallel tree views with similarity metrics to represent changes between two ontologies.

In the end we conducted simple evaluation study to demonstrate that our new application acknowledged the original hypothesis and was in fact more intuitive and effective than other compared tools. The results gained from subjects involved in this experiment were mostly affirmative and eventually proved our suppositions right. This study also resulted in proposing other new features to help the users better answer some ontology difference related questions.

Not all the ideas we came up with could have been implemented within this thesis, hence one whole chapter dedicated to such ideas and proposals whose exploitation and implementation remained as a matter of further research can be found at the end of this thesis.

Bibliography

- [1] Rdf schema. [online]. [cit. 2009-04-26].
http://en.wikipedia.org/wiki/RDF_schema.
- [2] Resource description framework. [online]. [cit. 2009-04-26].
http://en.wikipedia.org/wiki/Resource_Description_Framework.
- [3] The web ontology language. [online]. [cit. 2009-04-26].
http://en.wikipedia.org/wiki/Web_Ontology_Language.
- [4] T. Barlow and P. Neville. A comparison of 2-d visualizations of hierarchies. In *INFOVIS '01: Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, page 131, Washington, DC, USA, 2001. IEEE Computer Society.
- [5] M. Bruls, K. Huizing, and J. van Wijk. Squarified treemaps. In *In Proceedings of the Joint Eurographics and IEEE TCVG Symposium on Visualization*, pages 33–42. Press, 1999.
- [6] T. R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43, 1995.
- [7] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization '91*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [8] A. Katifori, C. Halatsis, G. Lepouras, C. Vassilakis, and E. Giannopoulou. Ontology visualization methods—a survey. *ACM Comput. Surv.*, 39(4):10, 2007.
- [9] M. Klein and D. Fensel. Ontoview: Web-based ontology versioning. [online]. [cit. 2008-11-11]. <http://www.cs.vu.nl/~mcaklein/papers/ontoview.pdf>.
- [10] A. Kobsa. User experiments with tree visualization systems. In *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization*, pages 9–16, Washington, DC, USA, 2004. IEEE Computer Society.
- [11] N. F. Noy, S. Kunnatur, M. Klein, and M. A. Musen. Tracking changes during ontology evolution. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *Third International Semantic Web Conference*, pages 259+, Hiroshima, Japan, November 2004. Springer Berlin.
- [12] N. F. Noy and M. A. Musen. Prompt: Algorithm and tool for automated ontology merging and alignment. In *AAAI/IAAI*, pages 450–455, 2000.

- [13] D. Perrin. Prompt-viz: Ontology version comparison visualizations with treemaps. Master's thesis, University of Victoria, 2004. [online]. [cit.2008-11-11].
http://www.cs.uvic.ca/~chisel/thesis/David_Perrin_Thesis.pdf.
- [14] Manojit Sarkar and Marc H. Brown. Graphical fisheye views. Technical report, Providence, RI, USA, 1993.
- [15] S. T. Teoh. A study on multiple views for tree visualization. San Jose State University, CA, USA. Department of Computer Science.
- [16] Y. Theoharis, Y. Tzitzikas, D. Kotzinos, and V. Christophides. On graph features of semantic web schemas. *IEEE Trans. on Knowl. and Data Eng.*, 20(5):692–702, 2008.
- [17] Y. Tzitzikas, D. Kotzinos, and Y. Theoharis. On Ranking RDF Schema Elements (and its Application in Visualization). *Journal of Universal Computer Science*, 13(12):1854–1880, 2007.
- [18] F. van Ham and J. J. van Wijk. Beamtrees: Compact visualization of large hierarchies. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 93, Washington, DC, USA, 2002. IEEE Computer Society.
- [19] F. Vernier and L. Nigay. Modifiable treemaps containing variable-shaped units. CLIPS-IMAG Laboratory – University of Grenoble, 2000.
- [20] T. D. Wang and B. Parsia. Cropcircles: Topology sensitive visualization of owl class hierarchies. In *International Semantic Web Conference*, pages 695–708, 2006.
- [21] M. Wattenberg. Map of the market. 1998. [online]. [cit. 2009-01-05].
<http://www.smartmoney.com/marketmap/>.
- [22] Martin Wattenberg. Visualizing the stock market. In *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*, pages 188–189, New York, NY, USA, 1999. ACM.
- [23] D. Zeginis, Y. Tzitzikas, and V. Christophides. “On the Foundations of Computing Deltas Between RDF Models”. In *Proceedings of the 6th International Semantic Web Conference (ISWC-07)*, Busan, S. Korea, November 2007.

Appendix A

Pre Study Questions

A.1 Few questions before the experiment

- Are you a computer science student? Yes ☐ No ☐
- Are you familiar with Treemaps? Yes ☐ No ☐
- Do you know what *ontology* is? Yes ☐ No ☐
- Do you have any previous experiences with similar *diff* tools? Yes ☐ No ☐

Appendix B

Post Study Questions

B.1 Questions to answer after the main study

One is the worst grade, five the best.

	1	2	3	4	5
Overall ease of use	☐	☐	☐	☐	☐
Ease of learning	☐	☐	☐	☐	☐
Intuitive to use	☐	☐	☐	☐	☐
Informative (The visualizations and controls were useful)	☐	☐	☐	☐	☐

Appendix C

User Study Questionnaire

C.1 First for each question check one of the following possibilities

- Selected tool: RDFDiffVizz ☐ PromptDiff ☐ PromptViz ☐

C.2 Study questions

Please use the selected tool to answer the following questions to the best of your ability. If you are unable to answer a question continue with the next question.

- How much of the ontology has changed between the two versions? We are looking for the percentage of classes, properties and instances that have been modified in some way (added, deleted, moved or just changed). The percentage number does not mean to be accurate, just try to make as best estimation as possible.
- Find the name of the parent class of the class *Columnist*. If there is more then one parent class write down all of them. First, write the parent class/classes in the first version, then in the second version.
- Find class that has changed the most. Including changes between class properties, instances and class itself. By "changed the most" we do not mean class that contains the most changes (change operations) but class that is the least similar to its equivalent in the other version.
- How many new sub classes has the class *Person*?
- How many classes is the *Columnist* instance *Kelly* type of. The "type of" is meant as how many classes does the *Kelly* instance belong to?

- How was the property *name* changed?
- How many classes were moved to different position in the hierarchy?
- Write down the original class path for the class *Small_One*. Start writing from the top super class down to the direct super class. Do the same in the first and second version.
- How many new employees (instances of the class *Employee* have been added and how many have been deleted?
- For how many employees is the Editor Ms Gardiner responsible for in the first and the second version? "responsible for" is a property of class *Editor* and its value is type of class *Person*.

Appendix D

User manual

D.1 User manual

This manual describes all the main features implemented in RDFDiffVizz application.

- Prepared Example Project
- Create New Project
- Load an Ontology
- Display Ontology Model
- Before Comparing two Ontologies
- Display Ontology Difference
 - Comparison Tab
 - Comparison Control Panel
 - * Searching
 - Parallel Aligned Views
 - * Color Coding
 - * Similarity Metrics
 - * Overall Similarity Progress Bar

D.2 Prepared Example Project

There is a simple project prepared with two ontologies. Choose Diff → Load Example from the main menu, new project *Example project* will be created with two ontologies loaded *newspaper1.1.rdf* and *newspaper2.1.rdf*. Plus new window with the comparison results will appear presently.

D.3 Create new project

From the main menu choose Project → New project. New window will appear, in this window you must specify name of the project. Then you can immediately load ontologies (knowledge bases) you intend to compare. You can load here as many ontologies as you want. If you decide not to load any ontologies in this new project window, you can do so any time later in the application. You can create as many projects as you want, at every time, only one project is active and is marked with red icon, other non active projects are marked with blue icons. Some actions, like loading ontologies, always refer to currently active project. You can set project as active in the project popup menu.

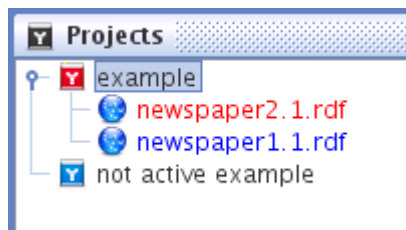


Figure D.1: In this Figure one active and one inactive project example is demonstrated. The red icon designates the currently active project. Under the active project there are two loaded ontologies, both set to be compared.

D.4 Load an ontology

You can load ontologies into application two ways. First, when creating new project. Second, from the KB menu pressing Load KB. Ontologies can be loaded only from local files. You can use the Local file tab or Sample file, where several sample ontologies are prepared. After selecting the ontology file new window will appear where you can choose namespaces to be included and loaded with the particular ontology. Later you can change the namespaces at any time, in the project manager after clicking right mouse button on any loaded ontology, select Load namespaces and the same window with all possible namespaces will appear again.

D.5 Display ontology model

When you have loaded an ontology, you can display it in a new frame and then walk all its classes, properties and instances. To do so, click right mouse button on the loaded ontology (in project manager window) you want to see and from the popup menu choose display model, new frame will appear with the desired ontology. In that window you can expand and collapse class nodes, if a class has any properties node Properties will be placed before other sub classes, if the class has any instances, node Instances node will be placed before any sub classes. When you place cursor above each node, simple tooltip window will appear showing some useful information about resources.

D.6 Before comparing two ontologies

Before the main comparison, you must choose two ontologies from the current project that will be compared. To achieve this, right click one ontology and from the popup menu chose Set as first to compare or Set as second to compare. First ontology to compare is marked with blue color, second ontology is marked with red color. You can select the same ontology as first and second to compare, this way it will be marked with yellow color.

D.7 Display Ontology Difference

After you have selected two ontologies to compare you can display the result by selecting Diff → open diff from the main menu. New tab will appear in the particular project frame. This tab is designated as Comparison.

D.7.1 Comparison Tab

The comparison tab is divided into two panels, in the top, there is a control panel and then the main parallel aligned views.

D.7.2 Comparison Control Panel

In the control panel you can find one input box and four buttons.

Searching

The input box is a search tool. You can place the cursor inside it by pressing Ctrl-F at any time. After you enter name of the resource you want to find, press Enter or Go button. Searching is case sensitive and you do not have to specify namespace of the resource you want to find. Searching will be then performed in all loaded namespaces, otherwise only in the one specified namespace. One resource can be located more than once in the hierarchy, when F3 pressed search tool will find another occurrence if so. You can search for classes and properties just using their names, in the case of instances it is a little bit complicated. You can not search for an instance using its label (the label displayed in tree views), you must use its ID. The instance's ID can be obtained in the tooltip window that will appear above each instance. (When searching for an instance you need not specify its namespace, just the ID, for example newspaper_00020).

Other Buttons

The moved button toggles displaying moved classes in the opposite view as temporary classes (marked with yellow rectangles). Tooltips button toggles showing tooltip windows in tree views and namespaces button toggles showing namespaces in those tooltip windows.

D.7.3 Parallel Aligned Views

There are three parallel views, in the first tree view, the first ontology version is displayed, the middle is the information panel with additional information about changes. And the third one is the tree view displaying second ontology version.

Color coding

Color coding for classes is explained in the corresponding Figure. When the icon contains small exclamation mark it means some deeper change occurred. In other words it prompts the user to explore deeper levels (sub classes, instances...). Color coding for properties and instances is similar, green color indicates deleted properties/instances and blue color added properties/instances.

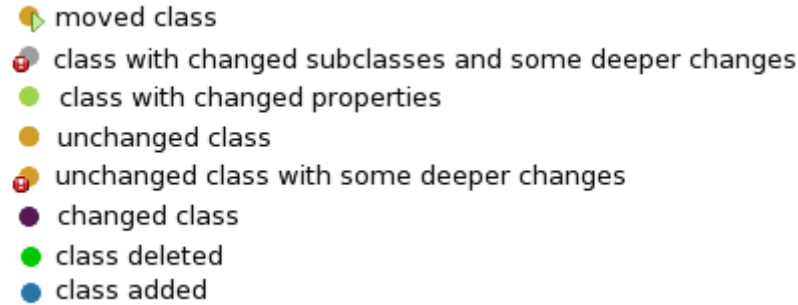


Figure D.2:

D.7.4 Similarity Metrics

In the middle column you can find similarity metrics computed for each resource (except for newly added or deleted resources). More information about each metric and how it is computed can be found in documentation.

Class Similarity Metrics

For classes there are seven main metrics plus the overall similarity. These metrics are:

- Average: This is the arithmetic mean of all six following metrics.
- Super classes: Similarity according to class' super classes (all super classes are counted, not only direct super classes)
- Sub classes: Similarity according to sub classes (again, all sub classes are counted, not only direct sub classes)
- All properties: Combines the Properties and Ranges metrics.
- Properties: Similarity according to class' properties (not inherited properties from super classes)
- Ranges: Similarity according to properties where this class is as range.
- Instances: Similarity according to class' instances.

Then there are absolute values of these metrics, plus the overall metric explained in documentation.

Property Similarity Metrics

- Average: This is the arithmetic mean of all three following metrics.
- Instances: Similarity according to all property instances. If the range of this property is another class, then compares all the instances of that class. If the range is Literal, for instance String, then compares all the String values of this property in all instances.
- Super properties: Similarity according to super properties.
- Sub properties: Similarity according to sub properties.

Instance similarity metrics

- Average: This is the arithmetic mean of all three following metrics.
- Classes: Similarity between two instances according to classes they belong to (in other words, according to their types).
- Properties: Similarity according to instance' properties and their values, detail explanation of this metric can be found in documentation.
- Occurrences: Similarity according to instance occurrences. Takes all the properties (from the ontology) with this instance as a value.

D.7.5 Overall similarity progress bar

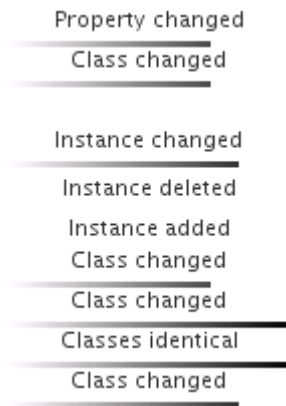


Figure D.3: This progress bar reflects the overall similarity metrics in the case of classes, in the case of properties and instances it reflects the average similarity metrics.