VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

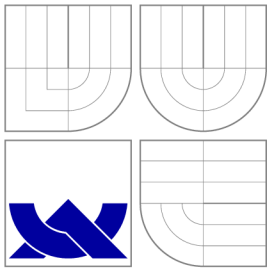
VISUAL DESIGN OF ONTOLOGIES FOR SEMANTIC WEB

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

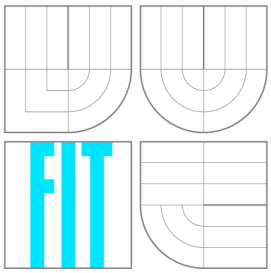
AUTOR PRÁCE
AUTHOR

JIŘÍ PROCHÁZKA

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

VIZUÁLNÍ NÁVRH ONTOLOGIÍ PRO SÉMANTICKÝ WEB

VISUAL DESIGN OF ONTOLOGIES FOR SEMANTIC WEB

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

JIŘÍ PROCHÁZKA

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. SVATOPLUK ŠPERKA

BRNO 2012

Abstrakt

Tato práce popisuje návrh a implementaci vizuálního editoru ontologií pro Sémanticý web, založený na RDF modelu, soustředící se na přehlednou kompaktní vizualizaci ontologií, jejich selektivní zobrazení z různých aspektů, a jejich tvorbu s rozšířitelností v nabídce ontologických jazyků.

Abstract

This thesis describes design and implementation of a visual ontology editor for the Semantic Web, based on the RDF model, focusing on compact ontology visualization, selective views of them from various aspects and their creation supporting extensible number of ontology languages.

Klíčová slova

Sémantický web, ontologie, vizualizace, RDF, editor

Keywords

Semantic web, ontology, visualization, RDF, editor

Citace

Jiří Procházka: Visual Design of Ontologies for Semantic Web, bakalářská práce, Brno, FIT VUT v Brně, 2012

Visual Design of Ontologies for Semantic Web

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Svatopluka Šperky. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Jiří Procházka
May 13, 2012

Poděkování

Tímto bych chtěl poděkovat mému vedoucímu práce panu Ing. Svatopluku Šperkovi za jeho cenné rady, trpělivost a dostupnost.

Contents

Chapter 1

Introduction

As the time and advances in computer science and engineering progressed, computers consecutively gained computational power and storage options expanded so far that it became possible to not to merely process and store data, but also metadata. Semantic Web aims to enable better integration and combination of data and metadata, in a way suitable for machine interpretation, as a part of World Wide Web (including private parts of it). Aside from standardization of common formats, a big part of that is definition and management of meaning of data — semantics. From Artificial Intelligence and its field of symbolic knowledge representation and conceptual modeling of Software Engineering stems definition of ontologies which serve as explicit conceptual knowledge models that make domain knowledge available to information systems. [6]

Ontologies find use in wide breadth of applications in areas like biology, chemistry, engineering or software management, health care, e–government and others. Their incorporation of taxonomical information makes ontologies suitable for graphical presentation, given their hierarchical nature, which isn't utilized by ontology authoring/editing tools much.

In this thesis I explore the current state of art of ontology editors with some visualization capabilities and I present my proof–of–concept design and implementation, which aims to surpass the existing alternatives, in terms of offering superior way of editing Semantic Web ontologies in visual graph–based environment, with extensibility in mind.

In chapter 2 more information about Semantic Web and introduction to its technologies, including ontologies and ontology languages, is provided. Chapter 3 explores ontology engineering, their visualization and evaluates existing visual editing tools. In chapter 4 the design of developed ontology editor is explained, together with its core features. The implementation is described in chapter 5. Chapter 6 evaluates the editor and contains suggestions for improvement and further development, and in conclusion in chapter 7 whole work is evaluated, including my personal experience with the project and its potential impact and future.

Chapter 2

Semantic Web and Ontologies

The Semantic Web is not a separate Web but an extension of the current one, in which information is given well–defined meaning, better enabling computers and people to work in cooperation. [3]

The web of today consists mostly of web pages and services meant for human consumption. If there is machine readable content, it is usually in form of documents and APIs with various syntax and semantics, often proprietary and with little or no effort for standardization. Such efforts more or less fall under the flag of Semantic Web. The World Wide Web Consortium (W3C) with its set of standards of Semantic Web like RDF, OWL etc. is leading the efforts. The more machine readable data is available on the web, the more tasks which today require user interaction can be automated, resulting in more intelligent applications and autonomous agents. The idea of knowledge bases for automated reasoning isn't new — building on volumes of Artificial Intelligence research, the idea of Semantic Web embraces properties of World Wide Web of working with dynamic growing set of incomplete, partially inconsistent data with varying availability. [3]

2.1 RDF

For Semantic Web to be successful on a scale similar to the WWW is to use language which is universal and agnostic enough to be generally useful for description of anything, which is a role of RDF (Resource Description Language) — RDF is used for representation of information about Web resources (metadata of a Web page/document) or by generalizing the concept of a „Web resource“, RDF can also be used to represent information about things that can be identified on the Web, even when they cannot be directly retrieved on the Web. [9]

One of most important aspects is using URIs (Uniform Resource Identifiers) [2] as identifiers of references of overwhelming majority of described things, so they are referenceable on the whole web, significantly simplifying their management.

Another important aspect is syntax agnosticism. Although first drafts were based on XML syntax, RDF isn't strictly said a language but a data model with various exchangeable serializations available, among them XML–based, JSON–based or others, such as Turtle, which is a popular, human friendly syntax. [9]

In RDF information is expressed using *triples*, consisting of *subject*, *predicate* and *object*, where predicate identifiers a relationship between things represented by subject and object, like a simple form of sentence in natural language (also it can be said that triples are

describing a resource with its properties and values, which sparks similarity with Entity–Attribute–Value model of information systems).
A set of triples is called an *RDF graph*, where the meaning of the graph is conjunction (logical AND) of its triples.
Subject and object are called *RDF nodes*, since very intuitive representation of the RDF model is a directed graph, explained in section 3.1.
Directed graph is a set of *nodes* connected by *edges* with a direction (set of ordered pairs of nodes).
Values of subject, predicate and object are *RDF terms*. *URI reference*, *blank node* or *RDF literal* are types which terms can take, however RDF places restrictions on various terms: predicate has to be a URI reference, subject a URI reference or blank node and object any of the three.
A URI reference or literal in subject or object position identify what that RDF node represents. A blank node is an RDF node that is not a URI reference or a literal and can be used as a reference, but without an intrinsic name (which limits the nodes reusability in other RDF graphs).
Literals are used to identify direct values such as numbers and dates by means of a lexical representation. Anything represented by a literal could also be represented by a URI, but it is often more convenient or intuitive to use literals.
Literals may be plain or typed — a plain literal is a string combined with an optional language tag. They may be used for plain text in a natural language and are self–denoting. A typed literal is a string combined with a datatype URI. It denotes the member of the identified datatype's value space obtained by applying the lexical–to–value mapping to the literal string. RDF doesn't have any set of datatypes of its own, but allows to use any by referring to them by URI and suggests to use some datatypes of XML Schema. [4]

2.2 Ontologies

An ontology is a specification of a conceptualization. [7] Ontologies exist for explicit formal representation of knowledge in information systems which in past have been implicit, known only to the designers and users of the information system — human agents. That didn't allow for data to be automatically reasoned about, using reasoning or rule engines implementing the ontology inference rules, which is one important feature of ontologies. Other feature is knowledge of classification schemes, like taxonomies. Classification schemes are hierarchical structures of classes, which are types or kinds of things, or different grouping of the classes. Taxonomies are classification schemes with more focus on nomenclature. Usually ontology languages provide the inference rules for their objects, so rarely ontology designer has to define them additionally. The term *ontology* has roots in philosophy, its branch of metaphysics, where it is concerned with the fundamental nature of existence, classification of things — their types. The philosophical meaning isn't so far disconnected from the meaning in computer science, however we limit ourselves only to particular domains for each ontology.

Further explained aspects of ontologies are formality, explicitness, consensus, conceptuality and domain specificity [6]:

- Formality — Ontology is expressed using ontology language, which ensures it is well–defined and machine–processable.

- Explicitness — Ontology features explicit knowledge to be known to the machines, which cannot infer implicit knowledge deemed by humans as common sense.
- Consensus — Development of ontology is accompanied by a process of reaching consensus among the target user group on the shared conceptualization which it represents.
- Conceptuality — Ontology specifies knowledge in conceptual way, such as the concepts intuitively make sense to humans.
- Domain Specificity — Ontology is limited to particular domain of interest, with desired detail.

2.2.1 Ontology Languages

Ontologies are defined in ontology languages, most common of them on Semantic Web are briefly described below.

RDFS

RDF Schema [8] describes itself as RDF's vocabulary description language. In context of RDF, vocabularies are understood as ontologies, albeit usually less strict, because RDFS doesn't provide vocabulary for describing existence or cardinality constrains of properties. It serves for building of hierarchies of classes and properties. It allows axiomatization only in form of domain and range restrictions besides subclassing and typing. In particular, RDFS does not exhibit the feature of expressing exclusion or negation of any form, which renders it as a semantically rather lightweight formalism. [6]

OWL

OWL Web Ontology Language [12] is a W3C endorsed ontology language, with strong ties to RDFS, with semantics based on Description Logic. Recently its revision called OWL 2 was released. OWL defines three variants — OWL Lite and OWL DL are tailored to easy implementation with basic functionality and strict Description Logic subset with eligible computational properties for reasoning engines, while the OWL Full language has some of the constrains relaxed, useful for database and knowledge representation systems also allowing unrestricted mixing with RDFS. Beside features of RDFS it provide means to construct classes by logical conjunction, disjunction, negation or use universal and existential restriction or other features like transitivity, functionality or inversion of properties.

SKOS

Simple Knowledge Organization System [1] is an ontology for knowledge organization systems, such as thesauri, taxonomies and other classification schemes, which itself is an ontological data, thus SKOS, while lightweight, is practically an ontology language. It is used for labeling of concepts, grouping them, organizing in informal hierarchies or expressing their association.

Chapter 3

Ontology Engineering

Ontology editors and development environments exist to make ontology creation and management more effective. While lot of overall usability depends on subjective aspects of user interface, making it difficult to evaluate, I will describe several aspects, which positively influence quality of an ontology engineering tool:

- Speed of navigation in the ontology
- Accessibility of often used constructs of the ontology language
- Integration of reasoning engines, ontology matching tools

There are various ontology engineering methodologies, which ideally the editors should be compatible with, however they are out of scope of this thesis.

3.1 Graph Drawing

For in visual ontology editing environment graph drawing algorithms are employed. The theory will be briefly explained in this section. Graphs are mathematical concepts consisting of *nodes* and *edges* representing the relations between them. Graphs can be directed or undirected, depending if edges are directional. Another property of graph is whether it is cyclic or acyclic, which depends in presence of cycles in graph. Disconnected graph can be divided in multiple disjoint sets of nodes which have no path between any of their nodes. A graph is bipartite if its nodes can be divided in two disjoint sets such that there are no edges between nodes of each set. Planar graphs can be drawn in such way that no edges intersect. [5]

In our case of ontology visualization we will be working with directed graphs, acyclic if representing hierarchies only.

3.1.1 Layout Algorithms

Layout algorithms serve positioning graph elements, there exist many various algorithms, suitable for different purposes. They usually consist of node position and edge routing (bending edge curves), often to minimize intersecting. Following algorithms are useful for visualization of ontologies:

Force–directed layout

Analogies of physical systems are employed for force–directed layouting, such as polar and parallel magnetic fields, gravity, or springs as graph edges. The algorithm rearrange nodes for certain number of iterations, computing the physical system.

Hierarchical layout

Hierarchical layouts have nodes distributed in several layers. Algorithms based on famous Sugiyama [11] approach consist of several phases. First nodes are distributed in layers by ranking and edges making graphs cyclic are removed, then each layer is arranged and finally whole graph is positioned from the layers, including edge routing.

Orthogonal layout

Orthogonal layouts are different in obvious way of having all edges running either horizontally or vertically with 90° corners. These algorithms are often aimed on minimizing edge crossing, and some are based on minimal network flow algorithms, which create a directed graph with edges with capacity and flow, like for example water pipe systems, exploring various paths of the system in search of one with the most desirable properties.

3.2 Existing Visual Ontology Editors

I will describe several existing visual ontology editors and development environments, concentrating on their visualization and visual editing capabilities. For this purpose I define a simple ontology which I will show in various editors:
Man is a person. Woman is a person. Parent is a person who has at least one child who is a person. Father is a person and a parent. Mother is a person and a parent. Having a child is a relationship between a parent and a person, and also means having influenced it, which is a relationship between 2 people. One cannot have himself as a child nor his child can have him as a child.
Serialization of the ontology in Turtle syntax can be found in appendix C.

3.2.1 Protégé

Protégé [17] is a major ontology development environment with large community of users and developers. It is written in Java using Swing user interface framework, extensible through a plugin framework, featuring advanced features such as extensive reasoning support. For Semantic Web is relevant its version Protégé–OWL. While editing in Protégé is done in mostly in non–visual way as shown on figure 3.2, one of the provided plugins is OWLViz, which provides visualization. However this visualization only serves to view and navigate the OWL class hierarchy, no editing through the visualization is possible and it only serves as navigational and overview tool illustrated by figure 3.1. The tested version was 4.1.0.

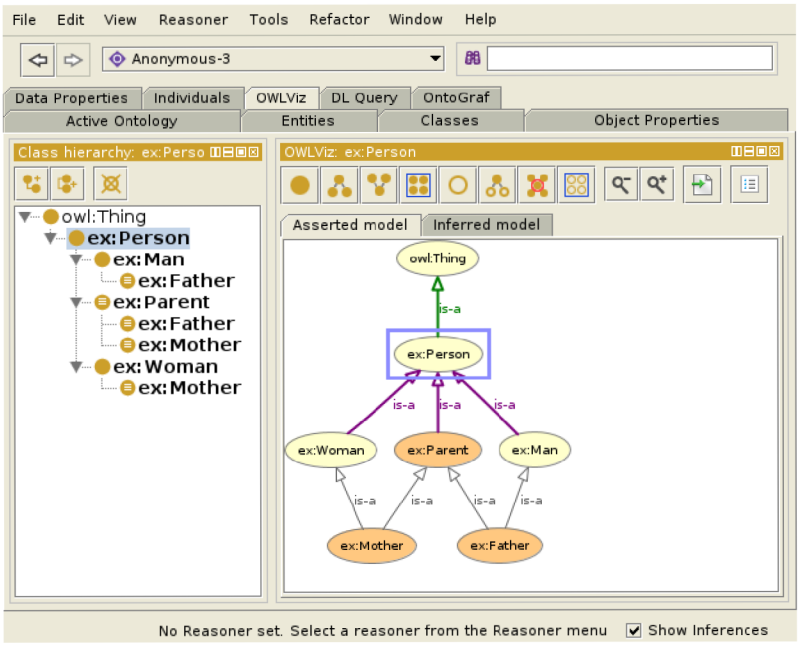


Figure 3.1: Example ontology visualization in Protégé

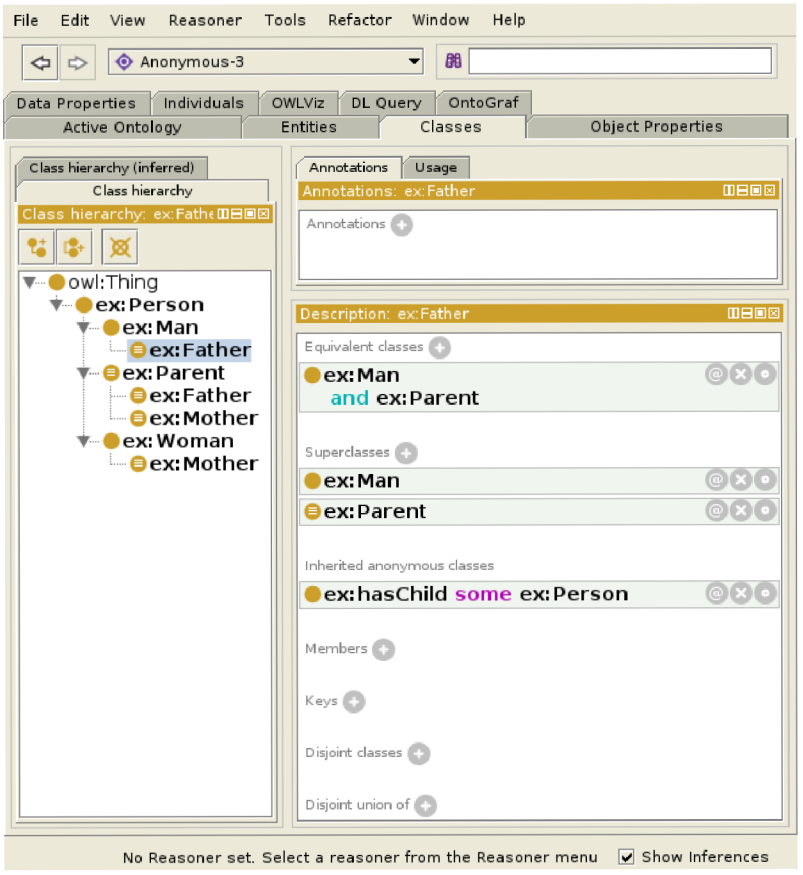


Figure 3.2: Editing of the example ontology in Protégé

3.2.2 CoGui

CoGui [14] is a graph–based visual tool for building Conceptual graph[1] [10] knowledge bases, written in Java. CoGui enables editing of RDFS/OWL ontologies, but it also focuses on other areas, like building of rules and queries. Its supported ontology expressivity is quite small, allowing only simple ontology creation, but it features editable both class hierarchy graph (figure 3.3) and property hierarchy graph (figure 3.4). It supports both hierarchical layout and force directed layout. Support of only basic ontology language features makes its visualization features appropriate, even for larger graphs, however it limits the tools utility. The tested version was 1.5b0.

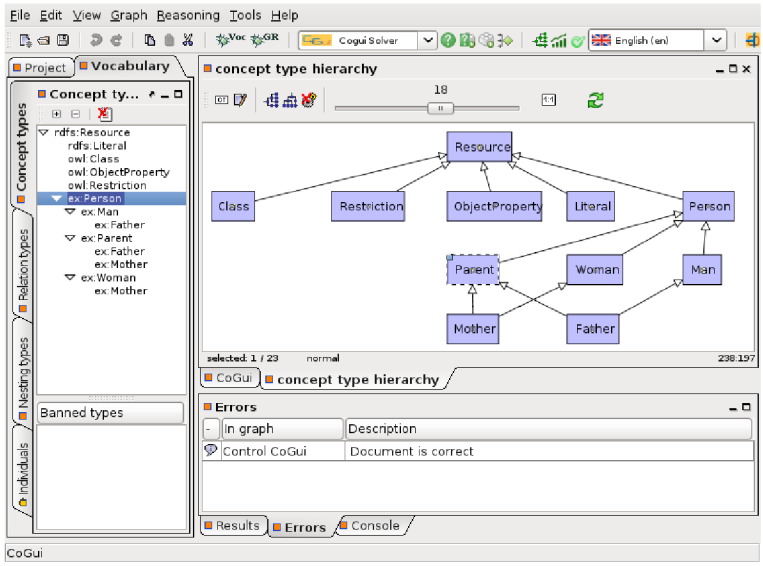


Figure 3.3: Example ontology class hierarchy in CoGui

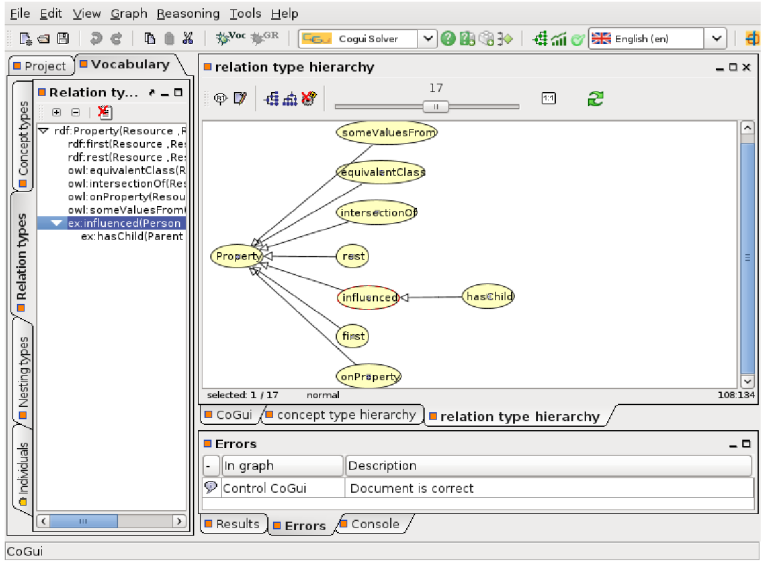


Figure 3.4: Example ontology property hierarchy in CoGui

[1] Conceptual graphs are a formalism allowing to graphically express meaning based on first–order logic.

3.2.3 COE

COE (Concept–map Ontology Environment) [15] is an ontology editor based on concept map software CmapTools. The tool is entirely visual, being a modification of a concept map editor. The ontology editing features are a set of conventions how to create the concept map, supplemented by autocomplete feature hinting user the possible constructs. Together with concept clustering of the ontology, as seen on figure 3.5, it enables quite user friendly ontology creation, although some problems and confusion may arise from the editor being primarily a concept map editor. COE supports OWL and automatic layouting of the graph using both hierarchical and force directed algorithms. The tested version was 5.0.03.

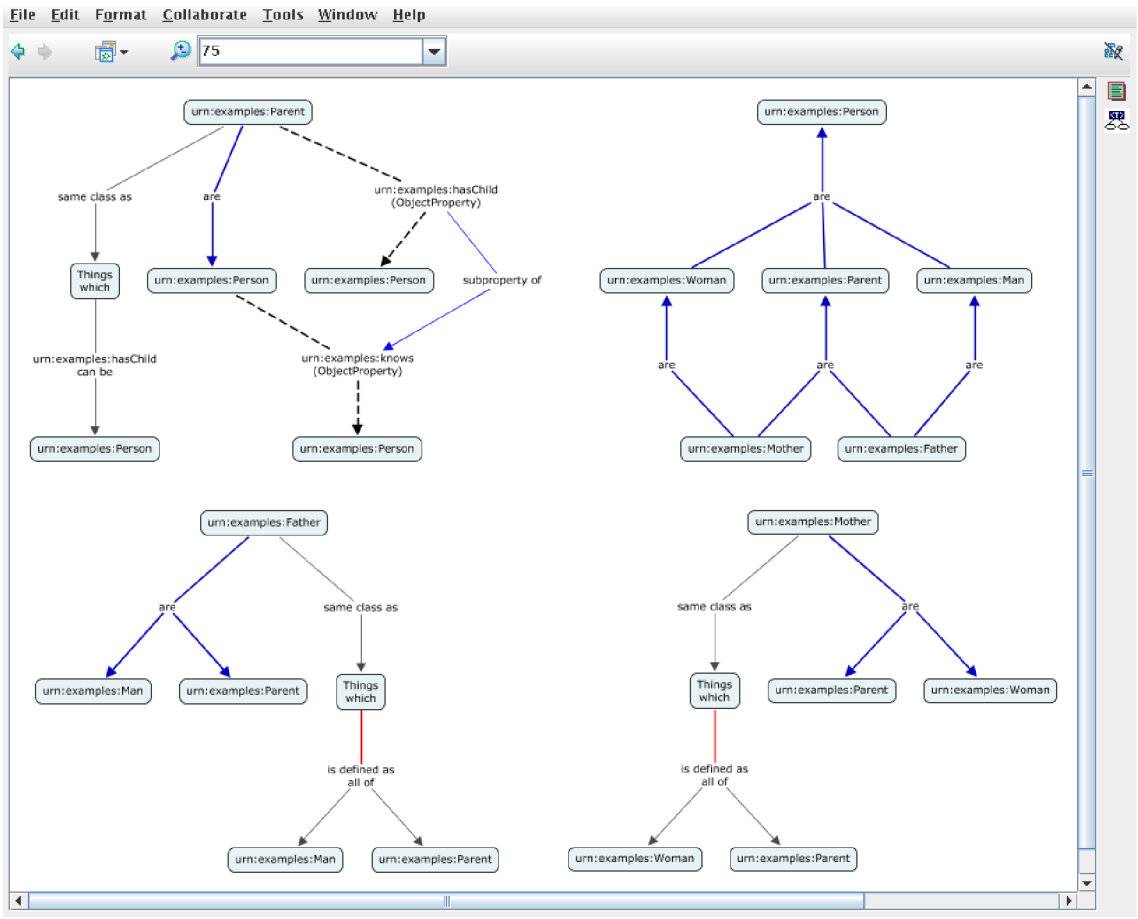


Figure 3.5: Example ontology in COE

3.2.4 Altova SemanticWorks®

Altova SemanticWorks®[13] is a commercial RDF and OWL editor. It supports interesting and unique visual editing features, as can be seen on figure 3.6. However its visualization is specific to each ontology class or property, allowing on editing of it and those directly related, switching resetting the view. This is very limiting, but coupled with its non–visual editing features it makes a viable ontology editor.

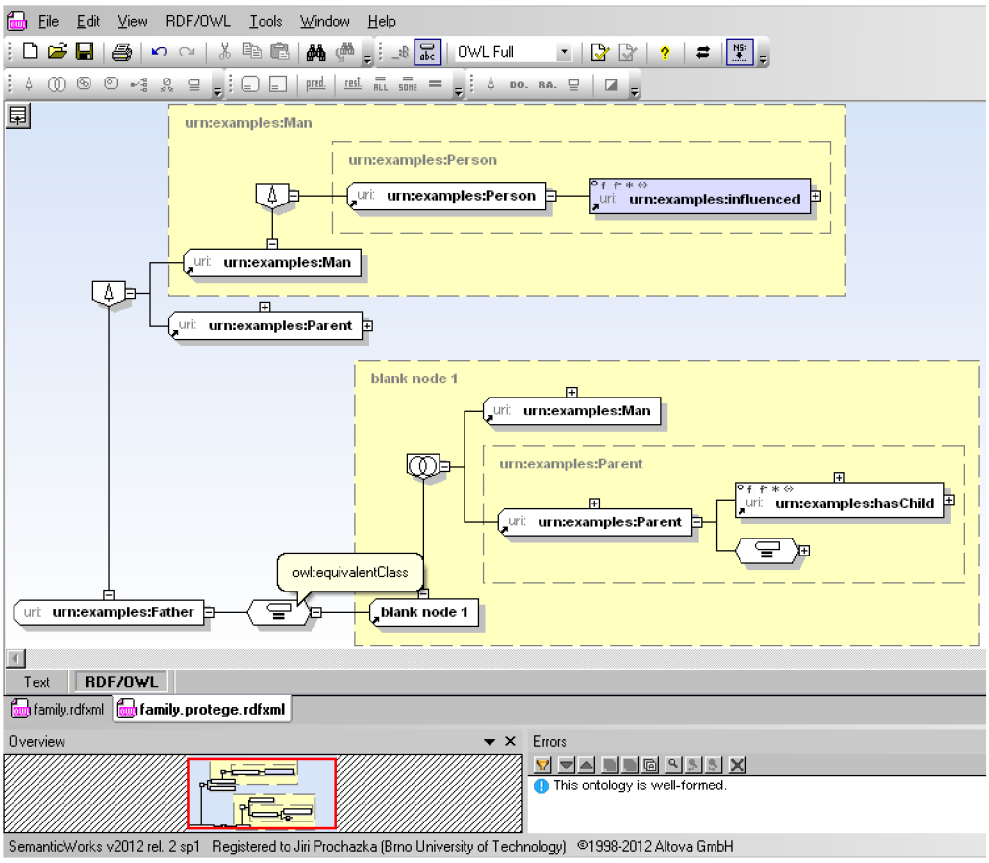


Figure 3.6: Example ontology in Altova SemanticWorks

3.2.5 TopBraid Composer™

TopBraid Composer™[20] is a commercial development environment for ontologies and other Semantic Web data, based on Java Eclipse, supports also advanced features like reasoning integration, data querying and inference rules. From visualization capabilities it offers two views: Diagram view, to be seen on figure 3.7, shows ontology in fashion similar to UML class diagrams enabling basic editing, while for advanced OWL features one has to use non–visual interface; Graph view, shown on figure 3.8, displays the ontology as raw RDF graph of triples, allowing even less editing. Both views initially show limited parts of the ontology (Graph view even just the one node) and user has option to expand the set of shown nodes. Unfortunately similar to SemanticWorks the views are not persistent, so when user switches between active classes/properties on the left tree list, the view resets. This clearly suggest the visualization features are designed to be for overview purpose and not main work. The tested version was Standard Edition 3.6.0.

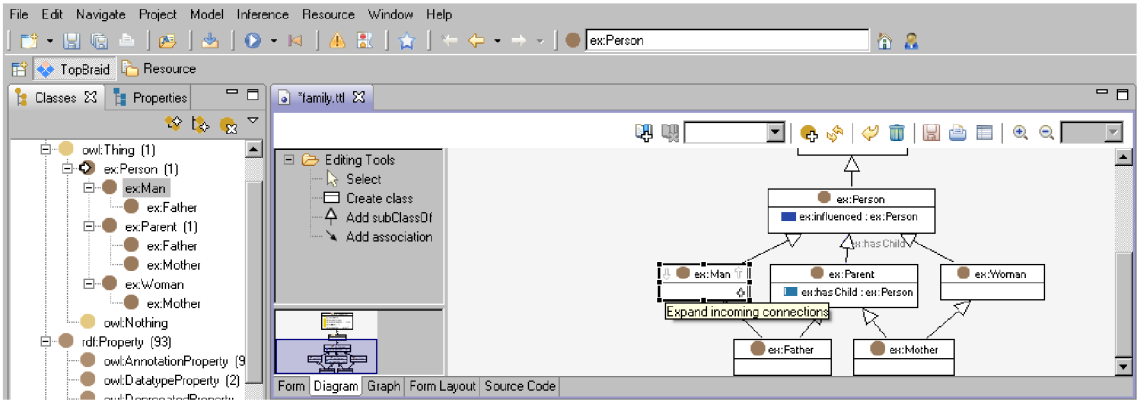


Figure 3.7: Diagram view of TopBraid Composer

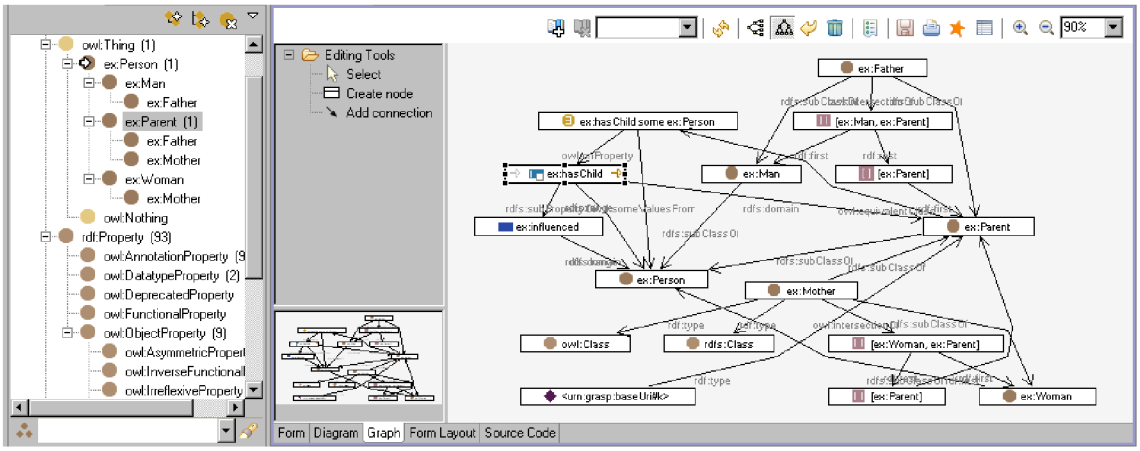


Figure 3.8: Graph view of TopBraid Composer

Chapter 4

Application Design

The design of the application is in part focused on addressing the shortcomings I observed in the existing ontology editors I explored:

- Limiting visualization capabilities only to overview and navigation functions
- Not allowing displaying of complete information of an ontology in visualization
- Resetting the visualization view when navigating the ontology

The other part of the design focus is guided by the beneficial user interface aspects of ontology engineering tools, as described in the beginning of the chapter 3, however since the purpose of the work is to build an innovative proof of concept ontology editor for Semantic Web, I decided to omit already well explored features, such as reasoning engine support. Instead I chose a design pursuing extensibility, and I developed concepts described in section below. The most important aspect of the application's extensibility is the choice of making the RDF model the data model of the editor. This means that by working in the editor, user is essentially editing a RDF graph. Aside from this being a logical choice, since the purpose of editor is authoring of ontologies for Semantic Web, it allows the editor to support dynamic number of ontology languages, which have a mapping to RDF, which all of those I described have. As a byproduct the application can be used not only as an ontology editor, but also as an RDF editor. How the example ontology looks as visualized RDF graph is depicted on figure 4.1.

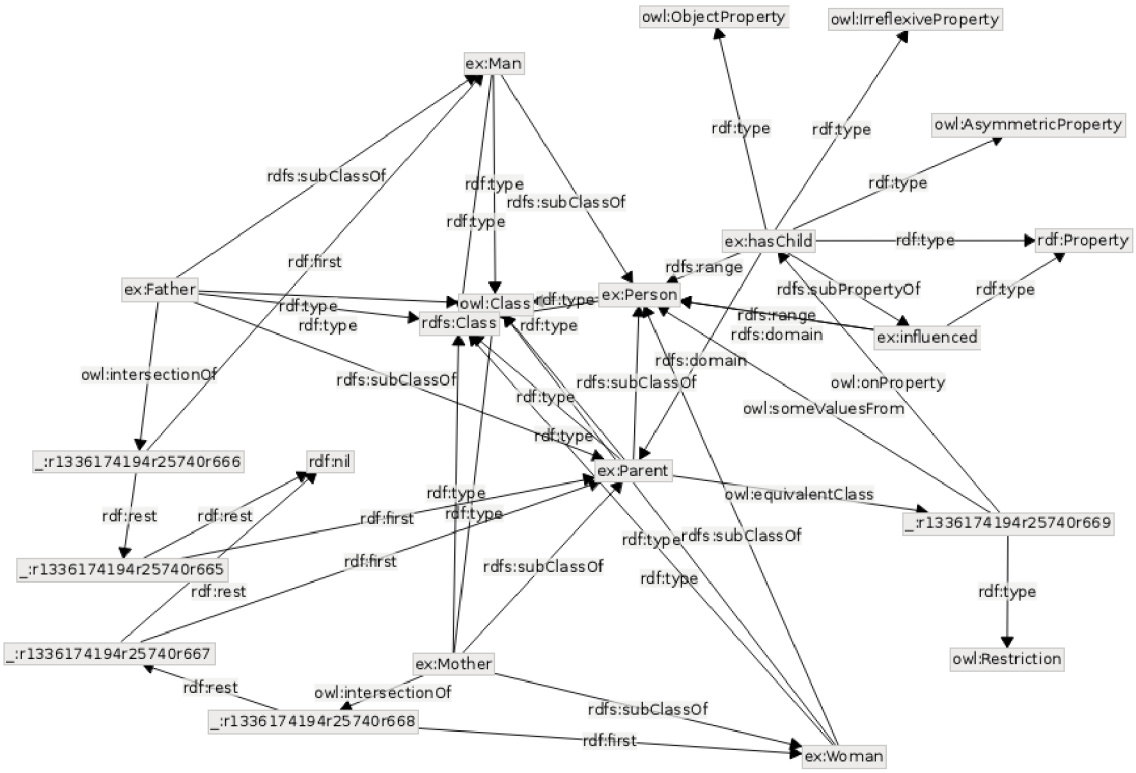


Figure 4.1: Example ontology as RDF graph

4.1 Node Aggregation

Because raw visualization of RDF graph is unsuitable to ontology editing, I developed two features which would simplify and clarify the visualization. One of them is node aggregation. Node aggregation is a feature consisting of aggregating certain relations of the RDF graph in the nodes of the graph of visualization. Example ontology, with node aggregating of RDF triples, where the object is of literal type, or the predicate is `rdf:type`, can be seen on figure 4.2. This allows to remove the aggregated relations from the graph, making it sparser. Resulting structure of node with aggregation is a tree, since the RDF nodes in object position can also have relations which are to be aggregated. Some loop detection is necessary to do, in order not to get into infinite loop, but about that in chapter 5. What relations are to be aggregated is a question regarding the second feature of graph simplification and is described in the next section.

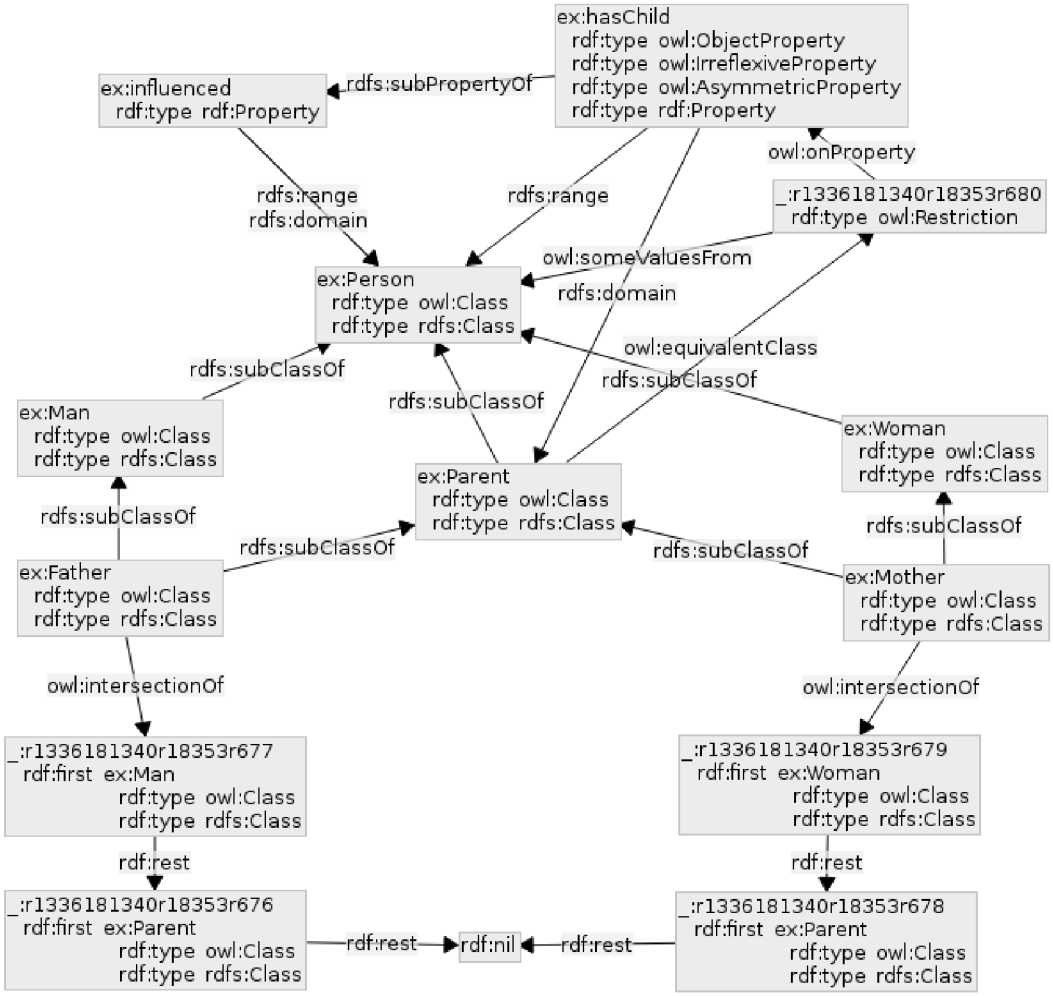


Figure 4.2: Example ontology with node aggregation of `rdf:type` relation

4.2 Lenses

Lenses are a feature enabling user to choose a particular specialized view of the graph, consisting of filtering of the graph and settings of node aggregation. The filtering consists of either blacklisting or whitelisting of relation types (defined by RDF predicate) which prunes the triples shown in the graph itself. Example of that can be seen in figure 4.3, which shows the usability of this feature, allowing user to easily switch between views of particular aspects of an ontology, such as class hierarchy in this case. From existing ontology editors I tried, similar effect was achieved by supporting just one filtered view of the ontology (like class hierarchy), which of course is not acceptable for fully visual editors.

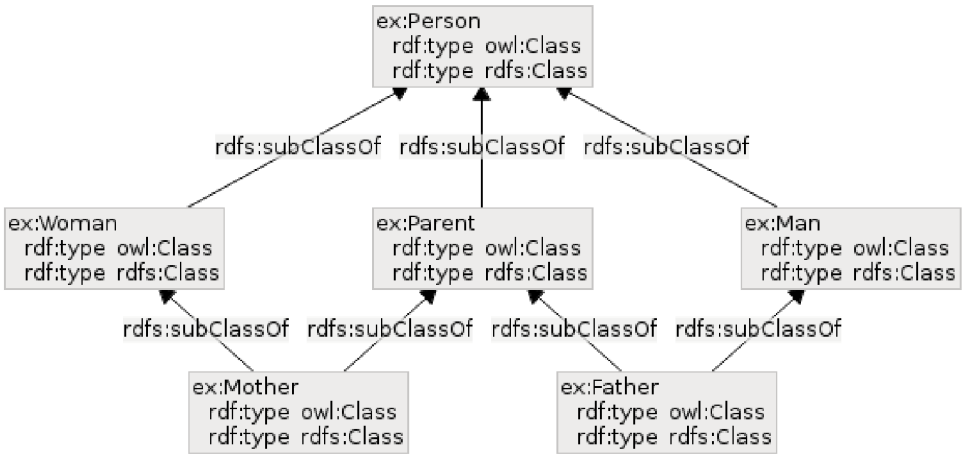


Figure 4.3: Class hierarchy lens view of the example ontology

4.3 Templates

Feature responsible for most of the editing functionality and accessibility of ontology language constructs are templates. Templates are RDF graph snippets of usually small size, which are inserted into the edited graph. Example showing a template for OWL universal quantification property restriction can be seen in figure 4.4. A node from the graph snippet can be marked to be replaced by currently selected node in the edited graph. In the example template that is the node `<urn:grasp:class>`. Additional information bundled with template description is specification of template contextuality — templates are often used for constructs which can be said that belong to a different construct, it is similar to member function ownership in object–oriented programming. The example construct, OWL property restriction, is tied to OWL class. This contextuality allows to arrange of the list of available templates into more useful data structure for presentation and filter out the templates which are not relevant to the selected node.

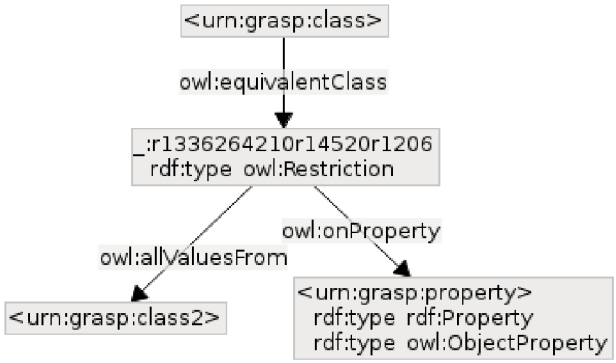


Figure 4.4: Example template for OWL universal quantification property restriction

Chapter 5

Implementation

5.1 Implementation Language and Used Libraries

For implementation I have chosen the programming language C++, for availability of wide range of libraries, performance reasons and its familiarity. There were other, more dynamic, contestant languages (mainly Python) in which could have made the process of implementation faster, however I decided that due to probably necessary implementation of missing features in libraries, it would probably not help from aspect of code base size, neither complexity of implementation and considerable performance would be lost, which for large RDF graphs, that the more extensive ontologies are, is desirable.

In following subsections I describe the libraries I chose to use. While deciding, aside the usability for the given purpose, I evaluated their cross–platform support and compatibility with open source code, which are both properties which I wanted the application to embrace.

5.1.1 Qt

Qt [18] is a mature and very extensive multiplatform C++ framework, focused on development of applications with rich graphical user interface (GUI). Large part of it is different implementation of portion of C++ Standard Template Library and with heavy use of macros it extends C++, requiring the code to be processed with its Meta Object Compiler tool. MOC enables important features of Qt to work and have simple syntax, like signal and slot framework which facilitates simple communication between Qt objects. Widgets, which are the user interface objects, like for example labels or buttons, can be organized in layouts (such as horizontal, vertical, grid etc.), which position and resize the widgets contained in them, to best use the available space. From now on, I will refer to these layouts as *widget layouts*, to avoid confusion with graph layout algorithms.

While normal GUI widgets Qt are based on raster graphics (pixel based), Qt features a Graphics View framework which is to be used for two–dimensional vector graphics, using floating–point numbers for coordinates, which is very convenient for graph visualization and obvious choice for an ontology editor. Graphics View framework enables to implement features such as zooming and export to Scalable Vector Graphics format (SVG) very easily. Qt provides a model/view framework, which is a simplified version of Model–View–Controller pattern, organizing code into models, for data management, and views, for rendering and user interaction, which is also used by the Graphics View framework. Graphics View framework reimplements its own versions of widgets, widget layouts, in classes `QGraphicsWidget`, `QGraphicsLayout` respectively, however the distinction between them and their regular

versions (`QWidget` and `QLayout`) are not interesting for the purpose of describing the application, so I will not disambiguate them and refer to them all as *widgets* and *widget layouts*.

GTK+ is an alternative cross-platform GUI library, written in C, however with bindings available for C++ and other languages. I chose Qt because of its mature Graphics View framework and based on my previous experiments with both Qt and GTK+, Qt suiting me more in many aspects, especially in being a C++ framework.

5.1.2 Redland RDF Libraries

Redland [19] is a collection of C libraries for working with RDF data. Redland comprises three libraries:

- Redland librdf — the core library, defining structures for basic RDF concepts, functions for their manipulation, and storage backend, for storing RDF graphs in memory or Oracle Berkeley DB, relational databases (MySQL, PostgreSQL, SQLite), triple-stores (OpenLink Virtoso), files or web resources.
- Raptor — implements parsing and serialization of various syntaxes of RDF.
- Rasqal — allows querying RDF data with SPARQL and RDQL query languages. It isn't used by the ontology editor implementation.

Redland is a C library but it follows conventions which make its design object oriented. In course of the implementation I made C++ wrappers for parts of the library, and implemented additional functions for URI printing to the Raptor library, which I submitted upstream to the project and will be available in the next release of the official distribution.

5.1.3 OGDF

Open Graph Drawing Framework [16] is a C++ library of graph layout algorithms and other graph algorithms. Originally I wanted to use Graphviz for graph layout algorithms, however OGDF offers nice C++ interface, where no serialization to strings and parsing of them is required, like for Graphviz C library, which, aside from being easier to implement, would slow down already very computationally intensive part of the application. From the library the layout algorithms used are hierarchic Sugiyama algorithm and force–directed Fast Multipole Multilevel Method (FMMM) layout algorithm.

5.2 User Interface and Functionality

The user interface, as can be seen in figure 5.1, is divided in several parts: the menu, the graph view, layout window and the prefix window. The layout and prefix windows are detachable or can be moved to any side of the graph view which is the main widget of the application.

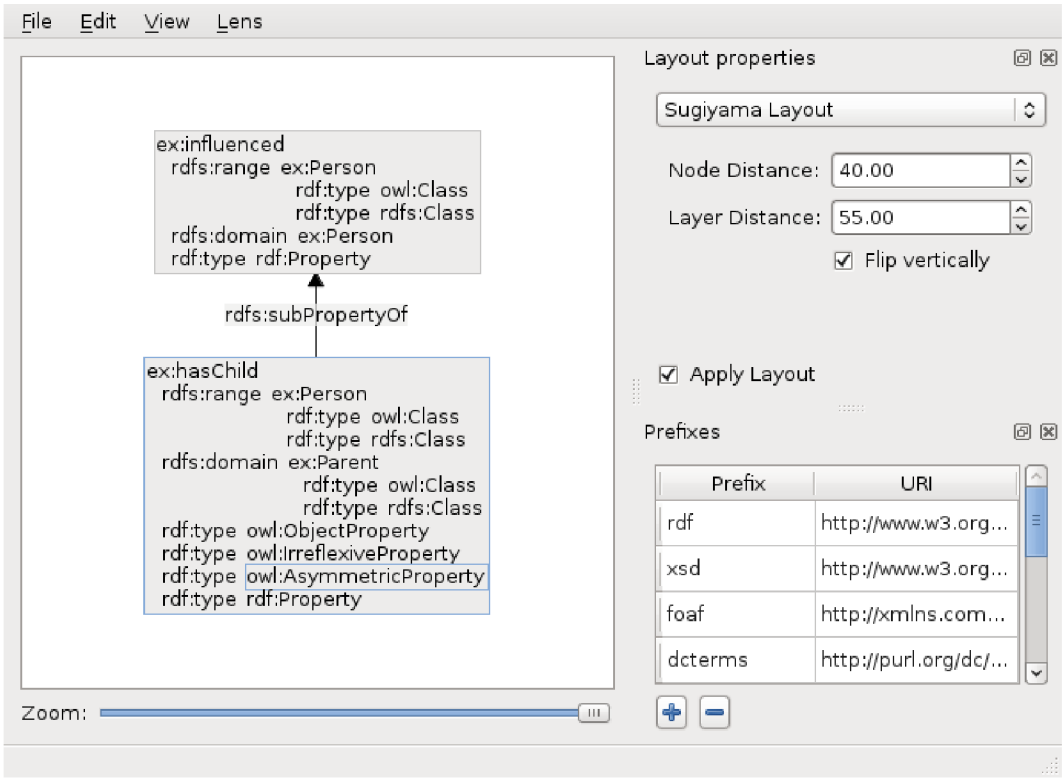


Figure 5.1: User interface of the application

Prefix window is for editing URI prefixes, which allow shortened form of URI to be viewed, replacing beginning of an URI with corresponding the prefix (if any) followed by a colon and the rest of the URI. Prefixes are well known in concept, sometimes also called as namespaces, or the prefixed URIs as QNames or CURIEs, and are used in various RDF syntaxes, such as Turtle. The implementation used is from Raptor library, and the functions for the URI prefixing outside of convoluted serializer code are what I had to additionally implement.

Layout window allows user to apply one of the graph layout algorithms to the graph view. The layout selection is independent of lenses, because lenses are for filtering of the graph view and the positioning is left to a user, which is arguably for the best, since user might position the nodes himself (by dragging them), and then switch freely between the lens views. For Sugiyama layout the configuration consists of the distance between the nodes on each layer, the distance between the layers and an option to flip vertically the graph positions, which is useful if user wants to view hierarchies using properties such as `rdfs:subClassOf` or `rdfs:subPropertyOf` in natural way — most abstract concepts being on top. For FMMM layout it consists just of the unit edge length which by increasing can be used to avoid node overlapping of large graphs.

Both prefix and layout windows can be re–opened if closed, from the view menu, which apart of that contains a checkbox for toggling visibility of nodes which have no edge towards or from them, because they were filtered out. Having them visible is useful when editing the ontology, but not for general overview, thus it isn't enabled on any of the screenshots. The lens menu contains list of available lenses to switch to and an option to reload the list, since lenses are specified as RDF data, which can be edited by the editor itself, allowing user to define a new lens when necessary.

The application reads and saves ontologies (or any other RDF graphs) in Turtle RDF serialization. By nature of Turtle serialization this also includes prefix information. Additionally the application saves information about node positioning and active lens in a way which is ignored by standard Turtle parsers, as explained in chapter 5. This ensures compatibility of the used syntax in both ways — the application reads any well–formed Turtle files and standard Turtle parsers read ontologies saved by the editor, omitting the information which are relevant only to the editor.

When the application is launched, the default file is loaded. This is an empty RDF graph with some useful prefixes predefined, however one can very easily change the default RDF graph, because it is just a file like any other edited ontology.

Most editing is done through the context menu, which is most often invoked by right–clicking. The editor supports some primitive RDF graph operations: add/remove relation and remove node, however most effective is to use templates which as well can be inserted to the graph from the context menu. When user doesn't have selected any node or an edge, the editor offers all templates, regardless of their contextual information. The implemented contextuality of templates are associated RDFS/OWL classes. So contextual menu when having selected a node will show only templates which have associated class of which is the selected node. For example, as shown on figure 5.2, `ex:Mother` is a `rdfs:Class` and `owl:Class`, so only templates for those 2 classes are shown.

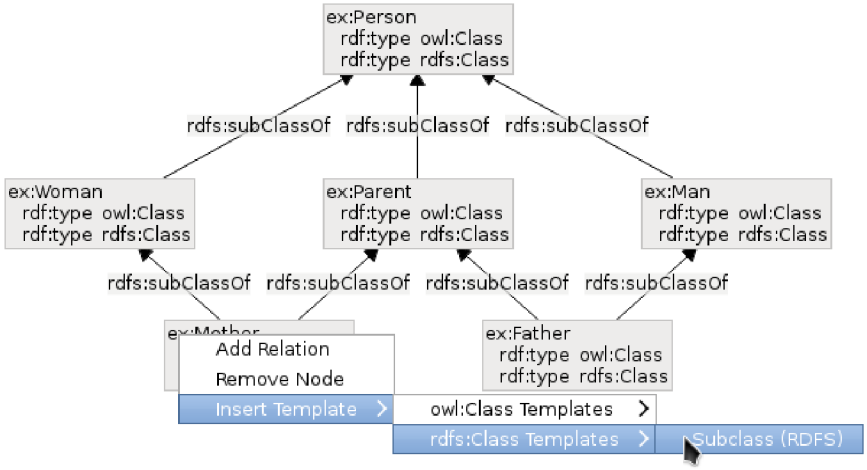


Figure 5.2: Contextual templates for a class

The other part of editing is invoked by double–clicking on a node or an edge and consists of editing the nodes as RDF nodes and edges as RDF predicates — one can „rename“ them, however it is imprecise to call it that. It is changing of what the RDF node or predicate is, whether URI reference, blank node or a literal, within the limits of the RDF model of course. Templates are inserted with temporary placeholders which are to be edited this way,

to build an ontology. Because all editing is happening on the RDF graph, the graph view is redrawn and possibly new relations are displayed, get aggregated in nodes or disappear by being filtered out. Utility of this approach is that in large graphs user doesn't have to know where the node he wants to refer to is, he just edits the placeholder value to correspond to the referred node. It is arguable how good this approach is compared to more traditional approach like connecting nodes by dragging, however I decided to concentrate on innovative features, so if the editor development would move past proof–of–concept, it would be best to support both ways.

Last notable feature is search. Search is the one option of the edit menu, offering easy forward and backward search of either nodes, or edges, scrolling the found element into the view.

5.3 Code Overview

The application is using Berkeley DB backend of Redland for storing its data, saved in files `maindb-*` also including a persistent incrementing counter which is used for generating unique identifiers. In `main.cpp` the RDF libraries and `MainWindow` are initialized, and `QApplication` is started.

`MainWindow` is a `QMainWindow` subclass, implementing the main window UI, loading of templates and lenses. Qt offers option to build GUIs using its tool Qt Designer, which is suitable for less dynamic GUI widgets, which was used for `MainWindow` and also for the RDF node edit dialog and the search dialog.

`GraphView` is a `QGraphicsView` subclass which manages the opened `Graphs`, zoom, search functions and UI for opening graphs.

`Graph` is possibly the most important class of the editor, subclass of `QGraphicsScene`, it is a class representing both the model of the `GraphView`, but also the RDF graph (which is in Redland vocabulary called *context*, but I will use the standard name). It handles refreshing of the graph when the RDF graph was changed by method `contextChanged()` and the mechanics of opening and saving the RDF graph. `Graph` contains dictionary container for positions of nodes of graph, key being hash of their corresponding RDF node. These hashes of RDF nodes are used as identifiers in the edited ontology files, because serialization and parsing of RDF nodes themselves would be too complicated, especially in case of RDF nodes of literal type, possibly with various whitespace characters, requiring lot of escaping. Hashing of blank nodes is special case, since Redland assigns random label to blank nodes, which would make their hash value different every time. So every `Graph` has a hash value translation dictionary which assigns value of the number of blank nodes which already are in the `Graph` to every new blank node, incrementing the count. This provides blank nodes with usable hash based identifiers, if the parser will parse the blank nodes in same order, which isn't guaranteed, but is often the case.

The `rdf` namespace contains all C++ Redland wrapper code and other functions used for manipulation of the RDF graph.

`GraphNode` and `GraphEdge` are classes representing nodes and edges on the `Graph`, both are subclasses of `QGraphicsWidget`, and implement mainly drawing functions and manage pointers to each other (`GraphNodes` sets of in/out edges, `GraphEdges` from/to nodes), while processing move events and forwarding double–click and context menu events to subclasses of `GraphicsLabel` — `GraphicsNodeLabel` and `GraphicsPropertyLabel` — which they own and which contain the RDF node and triple respectively, which they represent, and they manage opening of the edit dialog and context menu. `GraphicsLabel` is just a relatively

simple label widget (subclass of `QGraphicsWidget`), which the Graphics View framework didn't provide.

Node aggregation is composed of two classes `GraphAggregNode` and `GraphAggregProperty`, which are subclasses of `QGraphicsLinearLayout`, acting as horizontal and vertical layouts respectively, as drafted in figure 5.3. Aggregation is created by a call to recursive method `GraphNode::genAggregLevel(GraphicsNodeLabel *subjNode, QGraphicsLinearLayout *aggregProps)`, with nodes label and vertical layout for aggregation as arguments. `GraphNode` has a property `aggregStatements_`, which is a set of RDF triples, in which are collected those which already were aggregated, so the code doesn't enter infinite loop, or generate redundant structures.

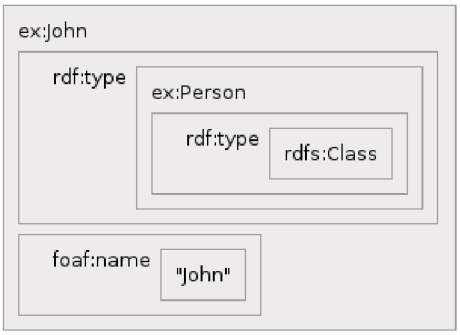


Figure 5.3: Draft of layout composition in node aggregation

The rest of the code is quite straightforward, has no significant points of interest and requires no further explanation.

5.4 Lens and Template Definitions

Lenses and Templates are described in RDF files `lens.ttl` and `templates.ttl`. I will briefly describe how they are defined, full specification of Lens and Template ontologies is in appendix.

Example lens definition can be seen in figure 5.4. The lens `lens:PropertyHierarchy` is in same namespace (using same prefix) as lens ontology, but that is fine, unless it conflicts with existing terms. Its type `lens:Lens` states, that it actually is a lens, other types `lens:WhitelistProperties` and `lens:NotAggregateLiterals` state, that triples with properties specified by relation of `lens:property` will be displayed in the graph and that triples with RDF literal nodes in object position will not be aggregated into the node of RDF node in subject position of the triple. The only other option is to use `lens:BlacklistProperties` instead of whitelisting it shows all triples by default, unless they are blacklisted or aggregated into a node, which is done by defining the property of triples to aggregate, by relation with predicate `lens:aggregatedProperty`.

Figure 5.4: Example of lens definition

As shown on example template definition in figure 5.5, templates are of RDF type `t:Template` and their name is specified by `t:name` property. As the information in `templates.ttl` is about the templates, not the content of templates, we need to point to the content with a `t:path` property, specifying path relative to the editor executable. Property `t:variable` is used for marking a RDF node which is to be replaced with selected RDF node, if any is selected. Lastly, using property `t:class` is stated what RDFS/OWL class the template belongs to contextually.

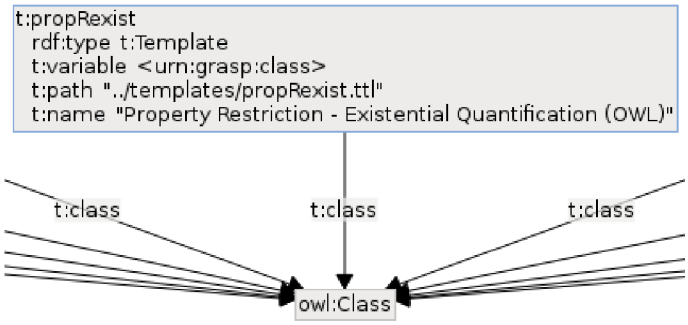


Figure 5.5: Example of template definition (cropped graph)

Chapter 6

Evaluation and Future Work

The implemented ontology editor is definitely viable ontology authoring tool, and the extensibility it offers allows it to extend its support of ontology languages easily, however it is noticeable that it is mainly an experimental proof–of–concept application, missing lot of features which are required for comfortable work. Indicative of the viability of the editor is that the lens and template ontologies (appendix D and E) were created using it. Anyway, to achieve state of full–featured ontology editor with friendly user interface, implementation of following features seems necessary:

- Drag and drop relation creating between nodes (with some modifier key pressed).
- Copy and paste, including whole nodes with all aggregated triples.
- Selection of multiple nodes and edges.
- Undo/redo.
- At least import and export to other RDF syntaxes, and ontology language syntaxes.
- Reasoning engine integration. Filtering out certain actions, like insertion of particular template, which would lead to inconsistent ontology.
- Extend template contextuality to arbitrary triple patterns. Extending template definition to allow deleting whole inserted templates.
- Choice of storage backends.
- Editing of RDF graphs of triplestore using SPARQL and web resources.

There are some more advanced features which deserve a mention, which would greatly increase the applications value and even further the state of art of visual ontology editing. One of them is advanced graph layout algorithms. Ontology editors would certainly benefit from dynamic graph layout algorithms, which are designed to change as little as possible when small part of the graph is modified. Other graph layout algorithm innovation could be combining properties of multiple algorithms, such as layout class and property hierarchies using hierarchic layout, then position the rest of the nodes around them using for example force–directed layout algorithm. I have searched for a graph layout library allowing this, however I was unsuccessful and design and implementation of such algorithms is itself a large enough topic for a thesis itself.

Another area to be explored is visual editing of multiple RDF graphs in one graph view. It is difficult to outline any obvious ways to manage that, but working with multiple RDF graphs and provenance is attracting lot of interest in Semantic Web community, and as well for ontology editing it is relevant — metadata as the node positions and the active lens could be saved as RDF data in separate RDF graph where metadata for the ontology RDF graph would reside, similar to how it is already now with the templates of the implemented ontology editor, however spreading all this related data among so many files is somewhat inconvenient.

Perhaps even developing support of semi–automated ontology mapping would be a worthy endeavor, which could help user to better define his ontology terms, by navigating existing ontologies and being offered to map his terms, even if the ontologies would have served only as informative examples.

The project was developed on installation of Arch Linux distribution of Linux operating system, however the code and all the libraries are cross–platform, so compilation on other platforms should be possible, however it was not tested.

Chapter 7

Conclusion

The purpose of this project — to create an innovative proof–of–concept visual ontology editor for Semantic Web — has been accomplished. While the editor is lacking in user interface comfort, it implements new features which address shortcomings of the existing ontology editors with visualization capabilities, and presents new alternative ways of ontology authoring, well tied to the technologies of Semantic Web. Editor is currently suitable for editing and navigation of RDFS and OWL ontologies, however by the virtue of its extensibility in this aspect it can be with ease made to support other ontology languages which are mapped to RDF. By choosing RDF as base model of the editor, I have traded quality of support of one ontology language I could have chosen, for the extensibility and flexibility, because implementation of some advanced features proved to be very time consuming, which could have been different if most of the flexibility was sacrificed.

The process of development of the application consisted of designing a relatively large application in C++, implementing complex GUI elements using Qt toolkit and features for creation of ontologies in Semantic Web ontology languages. I worked together with Dave Beckett, the creator of Redland RDF Libraries, on minor extension of his Raptor library and my patches were accepted into the main source code tree.

The results of the project are beneficial to the other visual ontology editing tools, which could incorporate ideas and features I described and explored, regardless if the project itself is developed further. All the core ideas — node aggregation, templates and lenses — are for consideration.

Bibliography

[1] Bechhofer, S.; Miles, A.: SKOS Simple Knowledge Organization System Reference. [online], 2009-08-18 [cit. 2012-05-08].
URL http://www.w3.org/TR/2009/REC-skos-reference-20090818/

[2] Berners-Lee, T.; Fielding, R.; Masinter, L.: Uniform Resource Identifier (URI): Generic Syntax. RFC 3986 (Standard), Jan. 2005 [cit. 2012-05-08], [online].
URL http://www.ietf.org/rfc/rfc3986.txt

[3] Berners-Lee, T.; Hendler, J.; Lassila, O.: The Semantic Web. *Scientific American*, 2001 [cit. 2012-05-08], [online].
URL http://www.scientificamerican.com/article.cfm?id=the-semantic-web

[4] Carroll, J. J.; Klyne, G.: Resource Description Framework (RDF): Concepts and Abstract Syntax. [online], 2004-02-10 [cit. 2012-05-08].
URL http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[5] Diehl, S.: *Visualization Basics*. Springer Berlin Heidelberg, 2007, ISBN 978-3-540-46505-8, 15-33 pp.
URL http://dx.doi.org/10.1007/978-3-540-46505-8_2

[6] Grimm, S.; Abecker, A.; Völker, J.; et al.: *Ontologies and the Semantic Web*. Springer Berlin Heidelberg, 2011, ISBN 978-3-540-92913-0, 507-579 pp.
URL http://dx.doi.org/10.1007/978-3-540-92913-0_13

[7] Gruber, T. R.: A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, vol. 5, no. 2, 1993 [cit. 2012-05-08]: pp. 199–220, [online].
URL http://tomgruber.org/writing/ontolingua-kaj-1993.htm

[8] Guha, R. V.; Brickley, D.: RDF Vocabulary Description Language 1.0: RDF Schema. [online], 2004-02-10 [cit. 2012-05-08].
URL http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[9] Miller, E.; Manola, F.: RDF Primer. [online], 2004-02-10 [cit. 2012-05-08].
URL http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[10] Sowa, J. F.: Conceptual graphs for a data base interface. *IBM J. Res. Dev.*, vol. 20, no. 4, Jul. 1976: pp. 336–357, ISSN 0018-8646, doi:10.1147/rd.204.0336.
URL http://dx.doi.org/10.1147/rd.204.0336

[11] Sugiyama, K.; Tagawa, S.; Toda, M.: Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions On Systems Man And Cybernetics*, vol. 11, no. 2, 1981: pp. 109–125.

[12] OWL 2 Web Ontology Language Document Overview. [online], 2009-10-29 [cit. 2012-05-08].
URL http://www.w3.org/TR/owl2-overview/

[13] Altova SemanticWorks®. [online], [cit. 2012-05-08].
URL http://www.altova.com/semanticworks.html

[14] CoGui. [online], [cit. 2012-05-08].
URL http://www2.lirmm.fr/cogui/

[15] Concept-map Ontology Environment. [online], [cit. 2012-05-08].
URL http://www.ihmc.us/groups/coe/

[16] Open Graph Drawing Framework. [online], [cit. 2012-05-08].
URL http://www.ogdf.net/doku.php

[17] Protégé Ontology Editor. [online], [cit. 2012-05-08].
URL http://protege.stanford.edu/

[18] Qt Framework. [online], [cit. 2012-05-08].
URL http://qt.nokia.com/

[19] Redland RDF Libraries. [online], [cit. 2012-05-08].
URL http://librdf.org/

[20] TopBraid Composer™. [online], [cit. 2012-05-08].
URL http://www.topquadrant.com/products/TB_Composer.html

Appendix A

CD Contents

- Source code with makefile: `/application/`
- Build instructions and other information: `/README.txt`
- License information: `/LICENSE.txt`
- Doxygen generated documentation: `/docs/`
- Bachelor's thesis in PDF format: `/projekt.pdf`
- Source code of the thesis: `/tex/`
- Example ontology in Turtle RDF syntax: `/family.ttl`
- Lens ontology in Turtle RDF syntax: `/lensOntology.ttl`
- Template ontology in Turtle RDF syntax: `/templateOntology.ttl`

Appendix B

Manual

Application is run with no special arguments, everything is controlled through the GUI as described in the thesis. Build instructions, including required dependencies, are specified in the `README.txt` file.

As can be seen on figure B.1, the GUI is very simple. Its elements are described in the thesis.

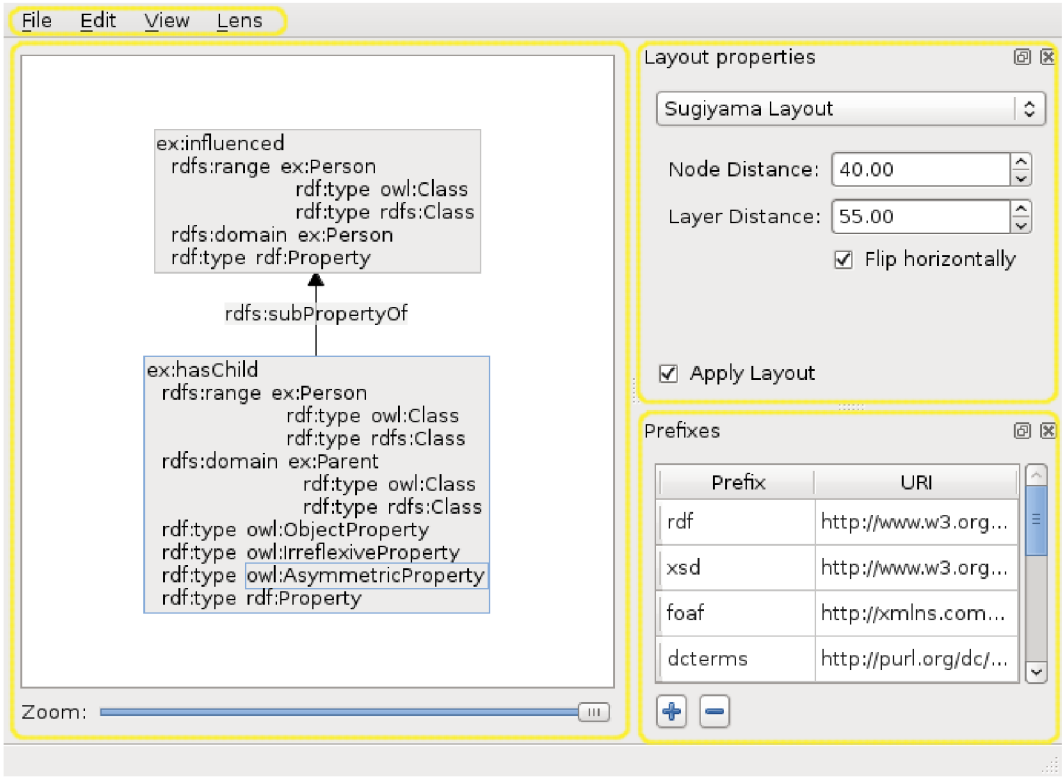


Figure B.1: User interface of the application

Appendix C

Example Ontology

Example ontology in Turtle RDF syntax:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix ex: <urn:examples:> .

ex:Father
    a rdfs:Class, owl:Class ;
    rdfs:subClassOf ex:Man, ex:Parent ;
    owl:intersectionOf (ex:Man ex:Parent ) .

ex:Man
    a rdfs:Class, owl:Class ;
    rdfs:subClassOf ex:Person .

ex:Mother
    a rdfs:Class, owl:Class ;
    rdfs:subClassOf ex:Parent, ex:Woman ;
    owl:intersectionOf (ex:Woman ex:Parent ) .

ex:Parent
    a rdfs:Class, owl:Class ;
    rdfs:subClassOf ex:Person ;
    owl:equivalentClass [
        a owl:Restriction ;
        owl:onProperty ex:hasChild ;
        owl:someValuesFrom ex:Person ] .

ex:Person
    a rdfs:Class, owl:Class .

ex:Woman
    a rdfs:Class, owl:Class ;
    rdfs:subClassOf ex:Person .

ex:hasChild
    a rdf:Property, owl:AsymmetricProperty, owl:IrreflexiveProperty,
        owl:ObjectProperty ;
```

```
    rdfs:domain ex:Parent ;
    rdfs:range ex:Person ;
    rdfs:subPropertyOf ex:influenced .

ex:influenced
    a rdf:Property ;
    rdfs:domain ex:Person ;
    rdfs:range ex:Person .
```

Appendix D

Lens Ontology

Lens ontology in Turtle RDF syntax:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix lens: <http://mud.cz/sw/ed#lens/> .

lens:Lens
    a rdfs:Class, owl:Class ;
    rdfs:comment "Class of lens." .

lens:BlacklistProperties
    a rdfs:Class, owl:Class ;
    rdfs:comment "Class of lens, which don't show relations with
specified properties in the graph view." ;
    rdfs:subClassOf lens:Lens .

lens:WhitelistProperties
    a rdfs:Class, owl:Class ;
    rdfs:comment "Class of lens, which show only relations with
specified properties in the graph view." ;
    rdfs:subClassOf lens:Lens .

[]
    a owl:AllDisjointClasses ;
    owl:members (lens:BlacklistProperties lens:WhitelistProperties ) .

lens:property
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying properties for
whitelisting/blacklisting." ;
    rdfs:domain lens:Lens ;
    rdfs:range rdf:Property .

lens:NotAggregateLiterals
    a rdfs:Class, owl:Class ;
    rdfs:comment "Class of lens, which don't aggregate relations with
literal objects in graph nodes as is usual." ;
    rdfs:subClassOf lens:Lens .
```

```
lens:aggregatedProperty
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying properties for aggregating in
graph nodes." ;
    rdfs:domain lens:Lens ;
    rdfs:range rdf:Property .
```

Appendix E

Template Ontology

Template ontology in Turtle RDF syntax:

```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix t: <http://mud.cz/sw/ed#templates/> .

t:Template
    a rdfs:Class, owl:Class ;
    rdfs:comment "Class of templates." .

t:class
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying a class to which the template
belongs to, as a means of classifying it in the menu for template insertion." ;
    rdfs:domain t:Template ;
    rdfs:range rdfs:Class .

t:name
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying template name" ;
    rdfs:domain t:Template .

t:path
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying path from the editor executable
to the template RDF file." ;
    rdfs:domain t:Template .

t:variable
    a rdf:Property, owl:ObjectProperty ;
    rdfs:comment "Property for specifying an RDF node which is to be
replaced by selected node on template insertion." ;
    rdfs:domain t:Template .
```