A Synchronized Graphical and Source Code Editor for RDF Vocabularies

Alexandra Similea

Matriculation number: 2776909

November 9, 2016

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Sören Auer

Niklas Petersen

Institut für Informatik III

Rheinischen Friedrich-Wilhelms-Universität Bonn

Contents

Chapter 1

Introduction

1.1 Background

1.2 Motivation

1.3 Contributions

1.4 Thesis Structure

Chapter 2

Related Work

In this chapter, we will focus on previous work that is related to some extent to the problem stated in Introduction. We will first present a theoretical approach for creating a synchronized graphical and code editor. Next, we will introduce a list of existing tools for editing RDF vocabularies (visually, textually or both). Finally, we will show why visualizing semantic data can raise several problems and we will explain the solutions that were found.

2.1 Theoretical Approach

In the paper "Robust Real-Time Synchronization between Textual and Graphical Editors" [1], Oskar van Rest et al. present an approach which is meant to ease the work with languages that feature both textual and graphical syntax. This work also suggests ways to overcome common synchronization problems such as error recovery and layout preservation.

The main idea of this approach is having an underlying model as a common factor for the two views. Another concept involved in the synchronization process is the abstract syntax tree (AST), which can be regarded as a tree representation of the syntactic structure of the code written in the textual view. Abstract syntax trees are commonly used by compilers during semantic analysis, in order to verify that the elements of the programming language are correctly used.

Figure 2.1 presents a broad overview of this approach. For the textual to graphical view synchronization, the code is parsed into an abstract syntax tree, which is then turned into a model. The resulting model is merged with the one belonging to the graphical view and the this view is updated accordingly. The inverse synchronization process starts with pushing the graphical changes into the model, which is next transformed into an abstract

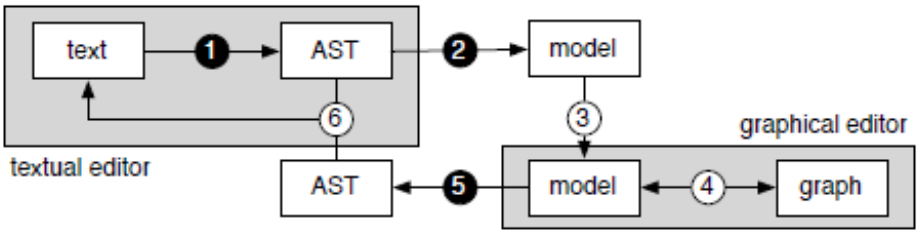


Figure 2.1: Steps involved in the synchronization process: 1) parsing, 2) tree to model transformation, 3) model merge, 4) graphical changes propagation, 5) model to text transformation, 6) tree to text printing.
Figure taken from [1].

syntax tree. The resulting tree is merged with the one belonging to the code view, which will be turned into text.

This approach is not feasible in our case as some steps do not need explicit implementation due to the numerous already existing frameworks for parsing and visualizing RDF data. Therefore, we are not bound to use abstract syntax trees, as RDF data can be easily parsed into an array of triples (one good example is the *N3.js* Javascript library[1]). Moreover, we do not have to use one model for each view. In fact, we did implement the idea of having an underlying model, but, in our case, it is the same for the two views and represents the only connection between them. Both textual and graphical data is parsed directly into the common model and an update process is triggered. More details on this can be found in the Implementation chapter.

2.2 Vocabulary Editors

In this section, we will present a list of tools for authoring and editing RDF vocabularies. Some of them feature only textual editors with the possibility to visualize the result, while the others enable the users to also edit the graphical display. None of them, though, synchronize the changes without user interaction.

1. Vocabulary collaboration and build environment (VoCol)[2] offers a collaborative environment for building ontologies. Vocabulary files storage and versioning control are achieved through repository services like GitHub, GitLab and BitBucket. VoCol comes with a Turtle editor where files can be loaded from the repository and be modified. The changes can

[1] `https://github.com/RubenVerborgh/N3.js`
[2] `https://github.com/vocol/vocol`

be committed only when they pass certain rules of correctness, the editor featuring syntax validation done by tools like Rapper[3] or Jena Riot[4]. An important observation to be made at this point is that our implementation has as prerequisite the Turle editor offered by Vocol, together with its repository services. Furthermore, this environment offers the possibility of visualizing vocabulary elements using WebVOWL[5]. More details can be found in [2]. Figure 2.2(a) shows an example of the interface. The graphical view is not editale, though, and can be updated only after the changes were comitted to the repository.

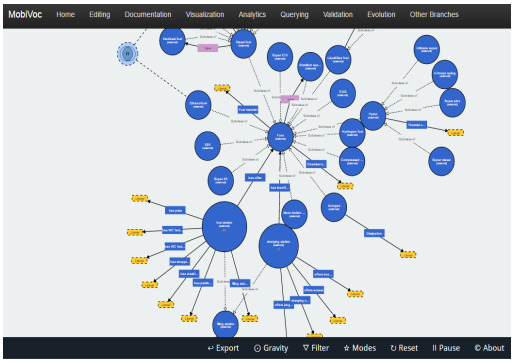
2. Protégé[6] is an ontology and knowledge base editor created by Stanford University and actively supported by the Protégé community. The tool allows the definition of classes, class hierarchies, variables, variable-value restrictions, relationships between classes and the properties of these relationships [3]. Ontologies can be uploaded and downloaded in various formats such as RDF/XML, Turtle, OWL/XML and OBO. Protégé's functionality can be divided into three area: creating ontologies, creating data using the ontology and querying the data. Moreover, the software comes with visualization packages (OntoViz[7], EZPAL[8] and others) and it can create a graphical user interface from the ontology, that is, forms with fields corresponding to elements in the ontology [4]. An example can be viewed in Figure 2.2(b). Protégé features a web extension - Web-Protégé, which aims to better support the collaborative development process in a web environment. The tool provides support for simultaneous editing, where a change made by an user is immediately seen by the other users [5]. Both editors enable modifying the graphical view and the changes can be exported into text files having the previously mentioned formats. Therefore, there is no support for immediate synchronization.

3. IsaViz[9] is a visual environment for browsing and authoring RDF models as graphs. This tool is offered by W3C Consortium [3], but it has not been maintained since 2007. IsaViz comes with an user interface which allows creating and editing graphs, together with zooming and navigation into the model. Figure 2.2(c) shows an example of the interface. The changes occuring in the graphical view can be synchronized with text only through file export. The tool allows importing ontologies in formats like RDF/XML,

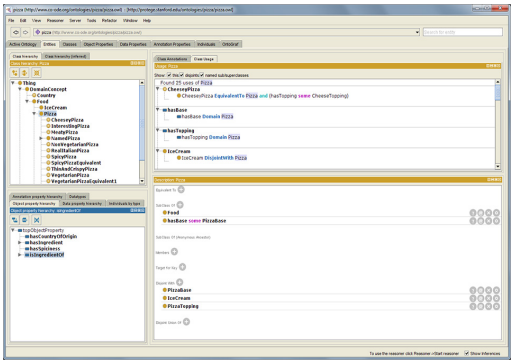
[3] `http://librdf.org/raptor/rapper.html`
[4] `https://jena.apache.org/documentation/io`
[5] `http://vowl.visualdataweb.org/webvowl.html`
[6] `http://protege.stanford.edu`
[7] `http://protegewiki.stanford.edu/wiki/OntoViz`
[8] `http://protegewiki.stanford.edu/wiki/EZPal`
[9] `https://www.w3.org/2001/11/IsaViz/`

Notation3 and N-Triple, while the export supports, besides the already mentioned formats, also SVG and PNG.

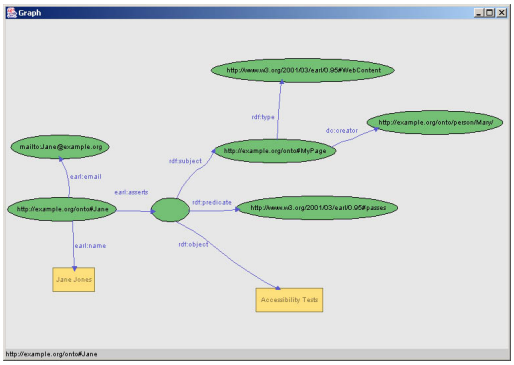
4. RDFauthor is a tool for viewing, creating and querying RDF instance data. It extracts structured information from RDFa-enhanced websites and creates an edit form based on this data. The RDF data model is presented graphically as a directed graph [4]. A visualization example is shown in Figure 2.2(d). In order to store the changes persistently in the triple store that was used to create the RDFa annotations, RDFauthor needs information about the data source (i.e. SPARQL endpoint) regarding the named RDF graph from which the triples were obtained or where they have to be updated [6]. An important contribution is that RDFauthor allows hiding the RDF and related ontology data models from novice users completely, thus allowing more people to author semantic content.



(a) Vocol



(b) Protégé



(c) IsaViz

(d) RDFauthor

Figure 2.2: Graphical user interfaces of different tools for editing RDF vocabularies.

2.3 Visualizing Semantic Data

Most of the tools that realize RDF data visualization generally work well with small models. However, semantic data describing web resources can easily reach thousands of nodes and, at this point, special techniques are needed in order to display the RDF model in such a manner that it is easy to understand and navigate.

GViz [7] is a general purpose visual environment for browsing and editing graph-based data. Its main advantage, compared to most other graph visualisation tools, is that it is easily customizable [8]. Modifying the graph layout is highly flexible, the users being able to choose the shape, color, size and other attributes of the nodes and edges. One interesting feature is the possibility to define callbacks in the Tcl scripting language, in order to further customize nodes and edges depending on certain attributes.

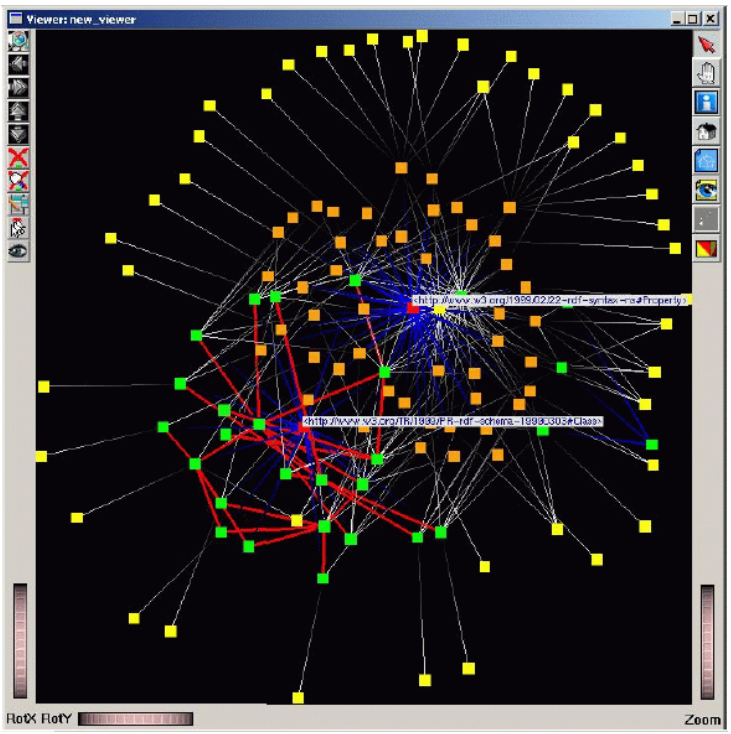


Figure 2.3: GViz ontology visualization. Figure taken from [8].

Figure 2.3 presents a visualization example, where choosing different colors makes the navigation more intuitive. Literals are depicted with yellow and displayed at the perifery as they are loose coupled, resources are green and nodes having an edge with the *rdf:Property* value are displayed in orange. Edges are also differentiated through colors, depending on their value:

rdf:type is blue, *rdfs:subClassOf* is red and the rest are white. Another important customization is not displaying the edges as arrows, but as lines fading towards the subject, in order to avoid the clutter produced by highly connected graphs. This approach proves itself very helpful when it comes to easily finding the interesting nodes, i.e. the nodes describing the web resources that the model defines, as they will always stand out due to their different color.

Another approach for intuitive graph visualization was investigated in [9]. The main idea is about using the properties between instances in order to place the related nodes near to each other, while keeping the other nodes evenly distributed. The resulting graph will give the user insight into the structure and relationships in the data model that are hard to see in text [9]. The drawing algorithm uses the spring embedding method [10], which distributes the nodes in a two-dimensional space and, at the same time, keeps the connected nodes reasonably close together. The graph is considered as a force system were each node simulates a charged particle, which causes a repulsive force, and each edge is modeled as a spring that exerts an attractive force between the pair of nodes it connects. Figure 2.4 shows an example of such layout where connected nodes are close together, yet no pair of nodes are too close to each other due to the repulsive forces acting between them [9].

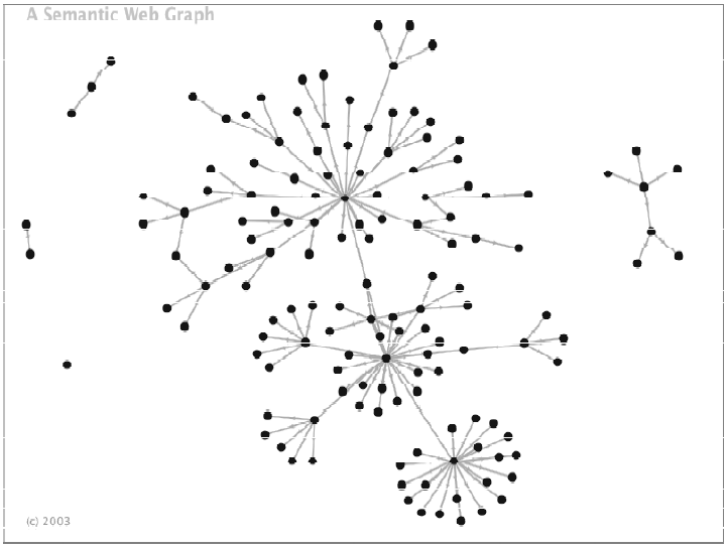


Figure 2.4: Graph layout using spring embedding. Figure taken from [9].

Chapter 3

Requirements

The development of a hybrid synchronized editor for RDF vocabularies was initially triggered by the lack of currently maintained graphical editors for semantic data. Having both a graphical and a code editor that are synchronized could be a good teaching method that would help domain experts who lack technical knowledge, with authoring and updating RDF vocabularies.

The code editing requirement has been already fulfilled as our implementation took off from the TurtleEditor project[1]. This consists of an open-source web editor, which can load files from, and commit changes to a central repository and offers features such as syntax highlighting, syntax checking and auto-completion [11].

The chapter is structured as follows: we will start with highlighting the graphical editor requirements, then, we will show what is demanded from the synchronization module and, finally, we will explain what is needed for a good visualization of a graph which is built using semantic data.

3.1 Graphical Editing

Having only a code editor turns out to be insufficient for authors lacking technical background, as they would still be bound to learning the syntax of the language in which the vocabulary is or needs to be written. Therefore, enabling editing through a graphical view is mandatory when it comes to ensuring intuitiveness and ease of understanding.

In order to fully enable creating and editing vocabularies in a graphical manner, a set of operations need to be made available to the graphical user interface through different types of forms. The following functions are to be supported:

[1] `https://github.com/vocol/vocol/tree/master/TurtleEditor`

1. Creating nodes as a representation for subjects or objects
    - Creating literals has to be differentiated from entities which are defined by URIs.
    - For a successful creation, the user has to specify a label, that is, the URI of an entity (the prefixed version has to be accepted too) or a literal itself.
    - The newly created node has to be easily identifiable in the graph.
    - Creating duplicate nodes needs to be prohibited.
2. Editing nodes
    - This assumes modifying the node's label.
    - Introducing a new label which is equal to another node's label has to be prohibited.
3. Deleting nodes
    - Edges that are linked to the node also have to be deleted.
    - Deleting a node may imply leaving other nodes disconnected from the graph. The user has to be prompted regarding keeping or discarding these nodes.
4. Creating edges as a representation for predicates
    - For a successful creation, the user has to specify a label, that is, the URI of the property (the prefixed version has to be accepted too).
    - The edge has to link two nodes or a node to itself, but in the latter case, the user has to be prompted if this is the actual intention, since this is a rather unusual case.
    - Creating duplicate edges is allowed.
5. Editing edges
    - This assumes modifying the edge's label.
    - Introducing a new label which is equal to another edge's label is allowed.
6. Deleting edges
    - Deleting an edge may imply leaving certain nodes disconnected from the graph. The user has to be prompted regarding keeping or discarding these nodes.

3.2 Synchronization

Creating or updating vocabularies with the help of a graphical editor and, later on, exporting the modifications to a file, like in the case of IsaViz (see Section 2.2), can be insufficient when teaching purposes are involved, as it is cumbersome to track each graphical change into text. Having an instant synchronization with a code view turns out to be more effective because the user can immediately spot the modified line of code and learn step-by-step. At the same time, supporting the inverse synchronization (from code to visual) is also important as it eliminates the need of user interaction for keeping both views updated and it becomes easier to spot possible mistakes in the model when the code modifications can be instantly visualized.

The two-way synchronization shall follow certain rules for keeping the model consistent and preventing the propagation of errors between the two views. Therefore, for updating the graphical side as a result of textual modifications, the syntax check function of the TurtleEditor shall be used. As a result, the synchronization process shall be triggered only when the code changes do not introduce any error. For updating the text view, we need a set of rules that apply for each of the graphical editor functions that we presented in the last section:

1. Creating a node - no update shall occur as the new nodes will be floating around, unlinked to the graph (no new triples are introduced until they get connected to other nodes).
2. Editing a node - update the corresponding triple in the text view.
3. Deleting a node - remove from code all triples containing this node as a subject or an object.
4. Creating an edge - introduce in the code the triple formed as a result of linking two nodes.
5. Editing an edge - update the corresponding triple in the text view.
6. Deleting an edge - remove from code the associated triple.

In order to better track the synchronization updates, the two editors shall be simultaneously visible so a split view is required. Moreover, each node shall be easily trackable in the code so a click event should determine the code editor update its view to the line containing the corresponding triple, together with the highlight of the associated term. The highlights shall also be seen on the scrollbar as this is useful when the node is associated with multiple triples.

3.3 Visualization

Visualizing semantic data is not a trivial task as an ontology can easily reach thousands of triples (see Section 2.3). Therefore, certain functionalities are required in order to make browsing the graph easier and more intuitive.

When a graph reaches a certain size and it becomes unmanageable, the obvious solution that comes to mind is grouping similar nodes together. So a first requirement that would ease the data visualization is clustering. This implies finding the appropriate criteria that determines the similarity and, at the same time, taking into account topological aspects. In what follows, we will define the nodes that have exactly one neighbour as outliers. We considered sufficient to cluster only the outliers together with the node that they are linked to, as this is equivalent to group a subject and all its properties. The graph clustering shall be possible until there are no more outliers, this meaning that clusters would be grouped together with other clusters. From our observations, this also makes sense semantically, as most of the times the outlier clusters represent subclasses of the cluster node they are linked to.

While visualizing large graphs, we noticed that there exist certain nodes which are highly connected, this meaning that they have more edges than the others. Usually, these nodes are part of namespaces like *rdf*, *rdfs* and *owl*. Therefore, we consider another requirement enabling the possibility to hide these nodes and their edges as this would remove the clutter that they generate.

Another solution that would eliminate the clutter is increasing the distance between nodes. However, this proves itself to be unfeasible as it would burden the graph navigation due to its wide expansion in space. Also considerable is the idea of duplicating the literals for each subject, used by VoCol (see Section 2.2), because this would make clustering cleaner - each literal would become outlier and be grouped with its subject, otherwise it will always appear in the graph, no matter how many clustering levels are applied. We will not consider this requirement, though, because it would violate our supposition that each node is unique.

The graph visualization shall be ruled by certain laws of physics that determine a level of gravitation between nodes, similar to what was explained in Section 2.3. Therefore, these rules will decide how nodes will be floating around and how far they will be from each other so, even if they are dragged by the user, they will always come back to their predetermined position. We considered that, at some point, the user will need to move the nodes out of different reasons (e.g. grouping, easier browsing etc.) so another requirement we are stating is the possibility to disable the physics, i.e., freezing the nodes as they are dragged to certain positions.

Chapter 4

Implementation

In this chapter, we will discuss in detail the construction of the synchronized hybrid editor. Our implementation comes on top of the already existing TurtleEditor [11] so we will start with presenting its capabilities. Then, we will explain the high-level architecture and, finally, we will provide a step-by-step description of the implemented features.

4.1 Preliminaries

Our work did not start from scratch as the code editing requirement had been already fulfilled by the TurtleEditor project, implemented in Javascript. This is an open-source web client which incorporates a code editor supporting features like syntax highlighting, syntax checking and auto-completion. Besides this, communication with external sources can also be realized by loading files from and committing changes to a central repository [11].

The following is a list of features (according to [11]) that the TurtleEditor comes with and that will be, as well, part of our final implementation:

- **code editing**: done using the *CodeMirror*[1] Javascript library. It also supports syntax highlighting for more than 100 languages, including Turtle - the language that our hybrid editor will use in its code view for designing vocabularies.
- **auto completion**: also achieved with CodeMirror, through its add-on *hint*, which requires defining the namespaces internally. Once a certain event is triggered (a keyboard combination, in our case - *Ctrl+Space*), the look-up process is started and, if the namespace is found (i.e., it

[1] `https://codemirror.net`

was previously defined), a list of available terms will be displayed in order to choose from.

- **validation**: implemented using the *N3.js*[2] Javascript library, which supports parsing Turtle code and detects possible syntax errors. When this happens, the faulty line is highlighted in red and a tooltip with additional information is provided via a red dot placed besides the line number.

- **repository communication**: realized by using the REST interface provided by the GitHub repository. The user can provide a repository source that will be checked out and a dropdown menu will be populated with the available vocabulary files so that they can be browsed and selected for editing.

- **access control**: this feature is needed for accessing and writing to private repositories. The user can log in with credentials or by using a generated personal access token to authenticate with the GitHub REST API [11].

4.2 Architecture

Figure 4.1 shows the high-level architecture of the TurtleEditor. The project basically consists of a web client which supports the features presented in the previous section. Since repository communication is also included, the diagram displays the server side too, which actually represents the repository hosting service together with the services that it is supposed to provide: access control, issue tracking, a wiki for the documentation and the version-controlled repository itself.

Our hybrid editor is entirely developed on the client-side and it represents, basically, an enhancement of the web client. On top of the features supported by the TurtleEditor, we add the implementation of the requirements presented in Chapter 3: graphical editing, synchronization of the two editors and the visualization module which includes various functionalities for displaying semantic data in a meaningful way. The enhanced client, which represents our contribution, is displayed in Figure 4.2

[2] `https://github.com/RubenVerborgh/N3.js`

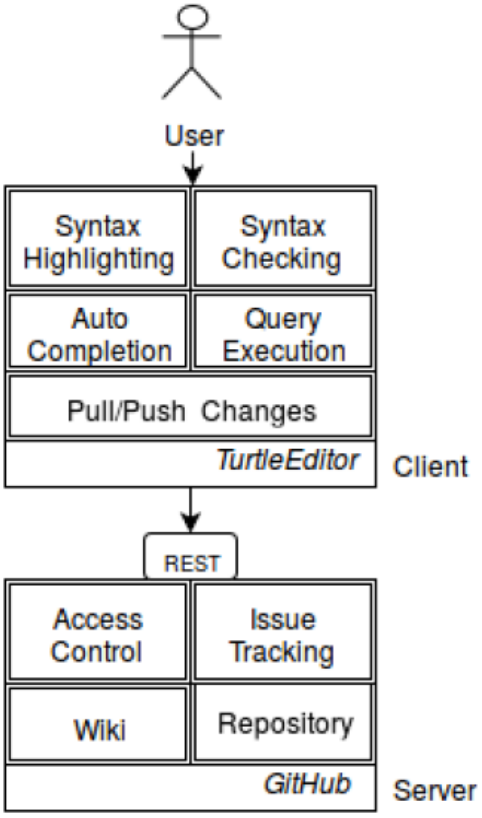


Figure 4.1: TurtleEditor architecture. Figure taken from [11].

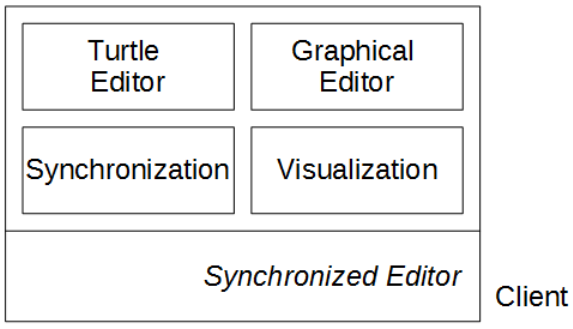


Figure 4.2: Enhanced client.

4.3 Features

1.

Chapter 5

Evaluation

5.1 Meeting the Requirements

5.2 Time Performance

5.3 User Evaluation

Chapter 6

Conclusions and Future Work

Bibliography

[1] O. van Rest, G. Wachsmuth, J. Steel, J. G. Süß, and E. Visser, "Robust real-time synchronization between textual and graphical editors," *Proceedings of ICMT '13*, vol. 7909, pp. 92–107, 2013.

[2] L. Halilaj, N. Petersen, I. Grangel-González, C. Lange, S. Auer, G. Coskun, and S. Lohmann, "VoCol: An integrated environment to support vocabulary development with version control systems," *1st Smart Data Innovation Conference (SDIC)*, 2016.

[3] B. Kapoor and S. Sharma, "A comparative study ontology building tools for semantic web applications," *International Journal of Web and Semantic Technology (IJWesT)*, vol. 1, no. 3, pp. 1–13, 2010.

[4] D. Steer, "Meg client software review." `http://www.ukoln.ac.uk/metadata/education/regproj/review`. Accessed: 27-Oct-2016, Last updated: 07-Jun-2002.

[5] T. Tudorache, J. Vendetti, and N. F. Noy, "Web-protégé: A lightweight owl ontology editor for the web," *5th OWL Experiences and Directions Workshop (OWLED 2008)*, 2008.

[6] S. Tramp, N. Heino, S. Auer, and P. Frischmuth, "Rdfauthor: Employing rdfa for collaborative knowledge engineering," in *Knowledge Engineering and Management by the Masses; 17th International Conference, EKAW 2010, Lisbon, Portugal* (P. Cimiano and H. Pinto, eds.), October 2010.

[7] A. Telea, A. Maccari, and C. Riva, "An open toolkit for prototyping reverse engineering visualization," *IEEE EG VisSym '02*, pp. 241–250, 2002.

[8] A. Telea, A. Frasincar, and G.-J. Houben, "Visualisation of rdf(s)-based information," *Proc. of 7th Intl. Conf. on Information Visualization (IV 2003)*, pp. 294–299, 2003.

[9] P. Mutton and J. Golbeck, "Visualization of semantic metadata and ontologies," *Seventh International Conference on Information Visualization (IV03), IEEE*, pp. 300–305, 2003.

[10] T. M. J. Fruchterman and E. Reingold, "Graph drawing by force-directed placement," *Software – Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, 1991.

[11] N. Petersen, G. Coskun, and C. Lange, "TurtleEditor: An ontology-aware web-editor for collaborative ontology development," *10th International Conference on Semantic Computing (ICSC)*, 2016.