A Synchronized Graphical and Source Code Editor for RDF Vocabularies

Alexandra Similea

Matriculation number: 2776909

January 8, 2017

Master Thesis

Computer Science

Supervisors:

Prof. Dr. Sören Auer

Niklas Petersen

Institut für Informatik III

Rheinische Friedrich-Wilhelms-Universität Bonn

Declaration of Authorship

I, Alexandra Similea, declare that this thesis, titled "A Synchronized Graphical and Source Code Editor for RDF Vocabularies", and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. Except for such quotations, this thesis is entirely my own work. I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Acknowledgements

I would like to express my sincere gratitude towards my supervisor, Niklas Petersen, who proposed the topic of this work and guided me closely throughout the thesis implementation.

I would also like to thank Prof. Dr. Sören Auer and Dr. Steffen Lohmann for valuable pieces of advice and suggestions. Moreover, I truly appreciate the constructive feedback given by the participants in the user study.

I am grateful, as well, to the University of Bonn for giving me the oportunity to study abroad and have an amazing international experience.

Last but not least, I would like to thank my family for supporting me throughout my life and studies.

Contents

List of Figures

Abstract

The shortage of graphical editors for RDF vocabularies motivated the development of our work. In this thesis, we propose a web client for visualizing and editing ontologies represented as RDF graphs. Additionally, the graphical editor is synchronized with a text-based editor. As a result, the changes performed on one view are instantly propagated to the other view, without the need of user interaction in order to keep the two sides updated. Another feature of the editor is graph clustering, which facilitates browsing RDF graphs even when they become large in size. Furthermore, the goal to provide a user-friendly application is validated in a user study. We also evaluated the time performance by conducting a series of tests.

Keywords: RDF editor, RDF visualization, textual and graphical synchronization, RDF clustering

Chapter 1

Introduction

The Semantic Web is based on the idea of interconnected data that is both machine readable and machine understandable. In this way, the data is not only displayed to the users, but it is also processed by applications, in order to provide more intuitive results and comply to the growing needs of the current era of information.

The common language for representing information about resources in the Semantic Web is RDF (Resource Description Framework). It is particularly intended for enabling information exchange between applications without loss of meaning. RDF represents metadata about Web resources in terms of simple properties and property values [1]. As a result, it is possible to create semantically rich data models, made up of triples (subject-predicate-object), where subjects and objects are entities, and predicates indicate relationships between those entities [2]. Such models that define terms and concepts describing a specific area of knowledge are known as vocabularies (or ontologies).

Ontologies play an important role in the development of Semantic Web, as they represent a way to give information a well-defined meaning [3].Their main purpose is to achieve data integration by providing shared conceptualizations of certain application domains. As a consequence, an increasing number of people in modern knowledge societies come into contact with ontologies. They are no longer exclusively used by ontology experts but also by other user groups, ranging from domain experts to non-expert users [4].

RDF data can be represented textually using different formats, such as RDF/XML, JSON or Turtle. The latter stands for Terse RDF Triple Language [5] and it provides a better human readability in contrast to the other formats. However, when it comes to ontology development, using the previously mentioned text formats requires a certain level of technical knowledge. Since ontologies are often authored by domain experts who lack this kind of

knowledge, a more intuitive method of development needs to be provided.

RDF data is visually representable as well, due to its interconnectivity among the defined entities. A triple forms a graph with two nodes (the subject and the object), connected by an edge (the predicate). Therefore, an ontology can be viewed as a graph structure, enabling users to quickly grasp the defined concepts and relations. While many graph-based ontology visualization tools have been developed, only few support direct editing of the visual representation. Graph visualizations of ontologies are currently mainly used for presenting and exploring ontologies, but not as an entry point to engage with ontology editing.

1.1 Contributions

Our work aims to lower the barrier for domain experts to engage with ontology development by providing the following:

1. A visual editor that enables direct editing of the visualized graph. The user can import an already existing ontology and modify its structure or they can build one from scratch using the features offered by the editor. The graph elements (nodes and edges) are identifiable through a text label, that is, the URI of the entity they represent. They can be created or deleted and also edited by changing the text contained within their label.

2. A synchronization module meant to synchronize the visual editor with a code editor. This means that the changes performed on one of the editors are immediately propagated to the other one, without any user interaction. The module aims to assist the teaching process by providing the textual representation of each element in the RDF graph. Moreover, experienced users who are familiar with the Turtle syntax for RDF might prefer code editing, so instant visualization of the textually developed models may be helpful as well.

3. A clustering feature that provides a meaningful visualization of large RDF graphs. Based on similarity measures of the elements and also on topological aspects of the graph, this functionality groups nodes together into other node elements called clusters, in order to produce a more clear and easy to navigate graph structure.

1.2 Thesis Structure

The remainder of this document is structured as follows. In the Related Work chapter, we present a theoretical approach for synchronizing editors that feature both a textual and a graphical part, then we introduce a few already existing editors for RDF data, followed by a list of approaches aiming to solve the problems raised by visualizing semantic data. In the Requirements chapter, we list the problems that led to the development of our editor, together with the requirements that needed to be fulfilled with respect to graphical editing, synchronization and intuitive visualization. In the Implementation chapter, we start by presenting the project on which our editor is based, next we explain the high-level architecture of our solution and then we provide a step-by-step description of the developed modules. In the Evaluation chapter, we first determine how the implementation requirements were met, second, we analyze the time performance of the graphical functions and third, we show the results of a user study that we conducted in order to assess the usability of our editor. Finally, in Conclusions, we summarize our work, ending with a set of suggestions for future improvement.

Chapter 2

Related Work

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

2.1 A Synchronization Approach

An approach which is meant to ease the work with languages that feature both textual and graphical syntax has been investigated by van Rest et al. in [6]. This work also suggests ways to overcome common synchronization problems such as error recovery and layout preservation.

The main idea of the method is having two interconnected underlying models, one for each editor. While the graphical changes can be pushed directly into the corresponding model, the textual side needs an intermediary - abstract syntax trees (AST), which can be regarded as tree representations of the syntactic structure of the code. 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 then this view is updated accordingly. The inverse synchronization process starts with pushing the graphical changes into the model, followed by its transformation into an abstract syntax tree. The resulting tree is merged with the one belonging to

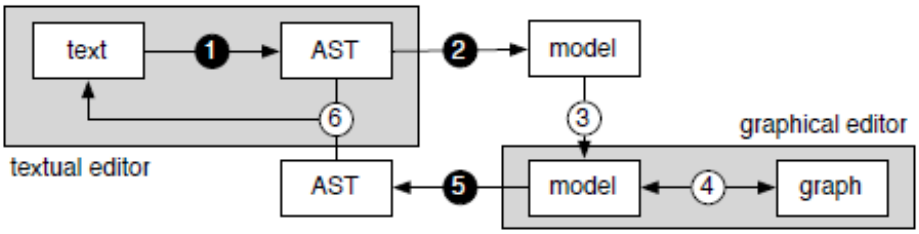


Figure 2.1: Steps involved in the synchronization process: 1) parsing, 2) tree to model transformation, 3) model merge, 4) graphical changes propagation, 5) model to text transformation, 6) tree to text printing. Figure taken from [6].

the code view, which will be turned into text.

This approach is not completely applicable in our case as some steps do not need explicit implementation. For example, we are not bound to use abstract syntax trees, as parsing RDF data is already supported by numerous tools (one such framework is the *N3.js* Javascript library[1]). Moreover, we do not have to use one model for each view. In fact, we will consider this idea, but, in our case, a more feasible approach is having a centralized underlying model for both views: the textual and graphical data shall be parsed directly into the common model, followed by an update process responsible for propagating the changes to the other view.

2.2 Vocabulary Editors

In this section, we will present a few 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. IsaViz[2] is a visual environment for browsing and authoring RDF models as graphs. This tool is offered by W3C Consortium [7], 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. However, it does not provide any clustering or other abstraction of the graph visualization, which quickly results in large RDF graphs that are hard to read and handle. Figure 2.2(a) shows an example of the interface.

[1] `https://github.com/RubenVerborgh/N3.js`
[2] `https://www.w3.org/2001/11/IsaViz`

The changes occurring in the graphical view can be synchronized with text only through file export. The tool allows importing ontologies in formats like RDF/XML, Notation3 and N-Triple, while the export supports, besides the already mentioned formats, also SVG and PNG.

2. Protégé[3] is an ontology 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 [7]. 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[4], EZPAL[5] and others) and it can create a graphical user interface from the ontology, that is, forms with fields corresponding to elements in the ontology [8]. 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 [9]. Both editors enable modifying the graphical view, i.e., the fields in the previously mentioned forms, and the changes can be exported into text files having the previously mentioned formats. This means, however, that immediate synchronization is not supported.

3. OntoSketch [10] takes a different approach for editing ontologies: The RDF graph elements can be drawn on a tablet computer using pen and paper-like interactions. The sketches are then converted into RDF triples that can be exported and further edited in other tools. The editor is available only as an Android application for tablets. Although this might assist domain experts to actively participate in ontology modeling, the aforementioned limitations remain.

4. Vocabulary collaboration and build environment (VoCol) [11] 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 code editor for Turtle format, where files can be loaded from the repository and be modified. The changes can be committed only when they pass certain rules of correctness, the editor featuring syntax validation, implemented with tools like Rapper[6] or

[3] `http://protege.stanford.edu`

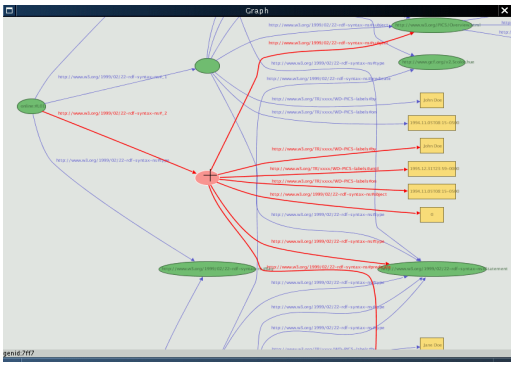
[4] `http://protegewiki.stanford.edu/wiki/OntoViz`

[5] `http://protegewiki.stanford.edu/wiki/EZPal`

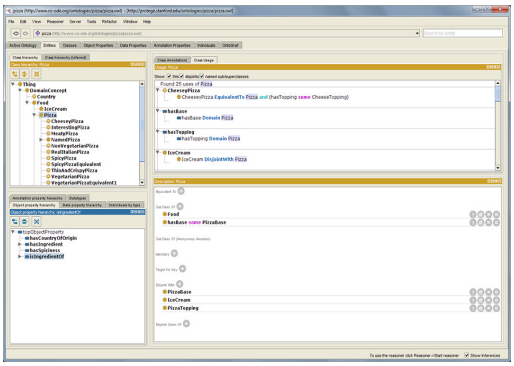
[6] `http://librdf.org/raptor/rapper.html`

Jena Riot[7]. Furthermore, this environment offers the possibility of visualizing vocabulary elements using WebVOWL[8]. Figure 2.2(d) shows an example of the interface. The graphical view is not editable, though, and can be updated only after the changes were committed to the repository. The entire code base is open source and available on GitHub[9]. An important observation to be made at this point is that our implementation has as prerequisite the code editor offered by Vocol, together with its repository services.

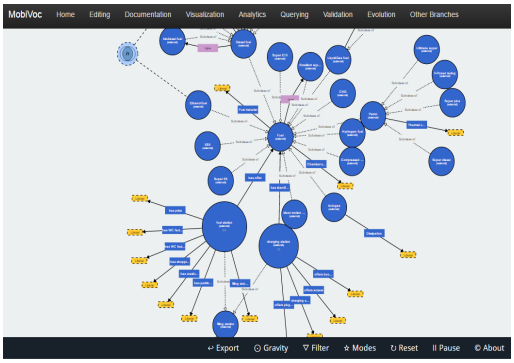
Most available tools support either visualization of existing ontologies or visual creation of new ontologies, but only few tools support both. Especially a synchronized textual and visual editing approach is currently not provided by the available RDF-based ontology editing tools to the best of our knowledge.



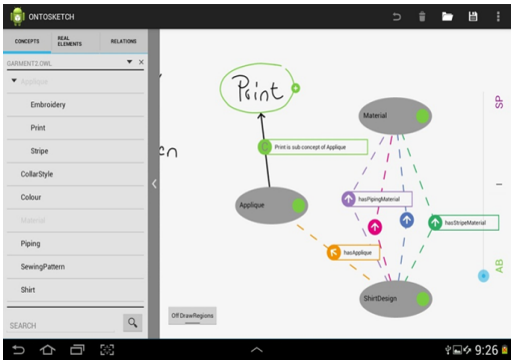
(a) IsaViz



(b) Protégé



(c) Vocol



(d) OntoSketch

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

[7] https://jena.apache.org/documentation/io
[8] http://vowl.visualdataweb.org/webvowl.html
[9] https://github.com/vocol/vocol

2.3 Visualizing Semantic Data

RDF data visualization tools are usually optimized for 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 [12] is a general purpose visual environment for browsing and editing graph-based data. Its main advantage, compared to most other graph visualization tools, is that it is easily customizable [13]. 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[10], in order to further customize nodes and edges depending on certain attributes.

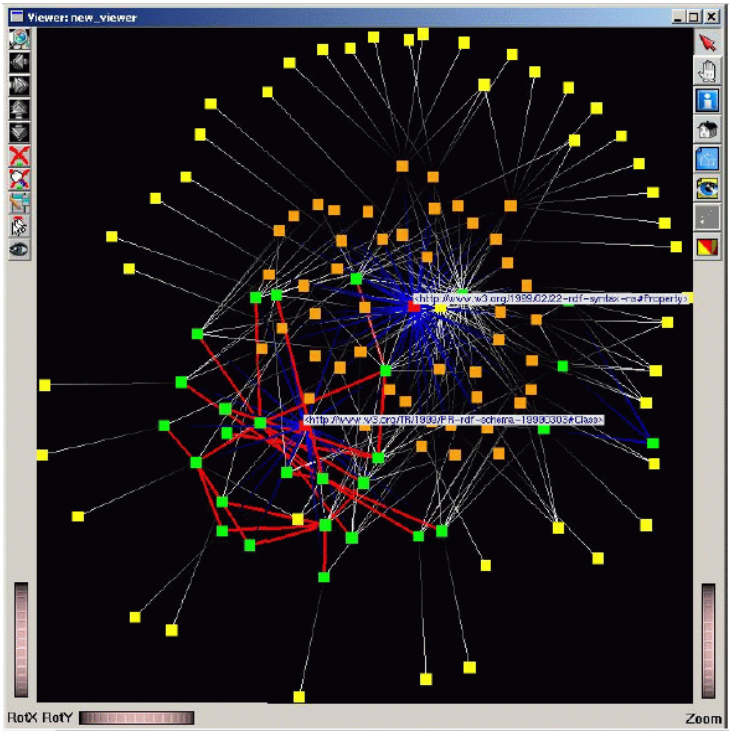


Figure 2.3: GViz ontology visualization. Figure taken from [13].

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 periphery as they are loose coupled, resources are green and nodes having an edge with the *rdf:Property* value are displayed in or-

[10] `https://www.tcl.tk`

ange. Edges are also differentiated through colors, depending on their value: *rdf:type* is blue, *rdfs:subClassOf* is red and the rest are white. Another 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 [2]. This method uses 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 [2]. The drawing algorithm uses the spring embedding method [14], 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 a force system where 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 [2].

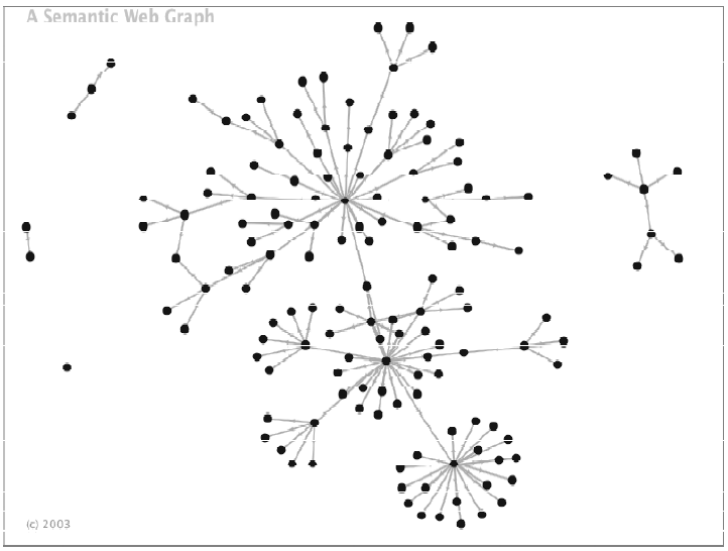


Figure 2.4: Graph layout using spring embedding. Figure taken from [2].

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 to 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 [15].

The chapter is structured as follows: we start with highlighting the graphical editor requirements, then, we show what is demanded from the synchronization module and, finally, we explain what is needed for a good visualization of a graph which is based on 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 shall 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), may 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 because it eliminates the need of user interaction for keeping both views updated and it becomes easier to detect possible mistakes in the model as code modifications are 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 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 the 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 the associated triple from the code.

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 in the graphical view 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 contained in 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 hard to manage, a trivial solution is grouping similar nodes together. So a first requirement that would facilitate the data visualization is clustering. This implies finding the appropriate criteria that determine the similarity and, at the same time, taking into account topological aspects. In what follows, we will define the nodes having exactly one neighbor 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, meaning that clusters can 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 are certain nodes which are highly connected, 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 premise 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 discuss in detail the construction of the synchronized hybrid editor. Our implementation comes on top of the already existing TurtleEditor [15] so we start with presenting its features. Then, we explain the high-level architecture and, finally, we provide a step-by-step description of the implemented modules.

4.1 Preliminaries

The code editing requirement has 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 [15].

The following is a list of features the TurtleEditor comes with and that will be, as well, part of our final application:

- **code editing**: implemented using the *CodeMirror*[1] Javascript library, which 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**: realized with CodeMirror as well, with the help of 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

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

found (i.e., it 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. Once this occurs, 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 input 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 [15].

4.2 Architecture

Figure 4.1 shows the high-level architecture of the TurtleEditor. The project basically consists of a web client supporting 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 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`

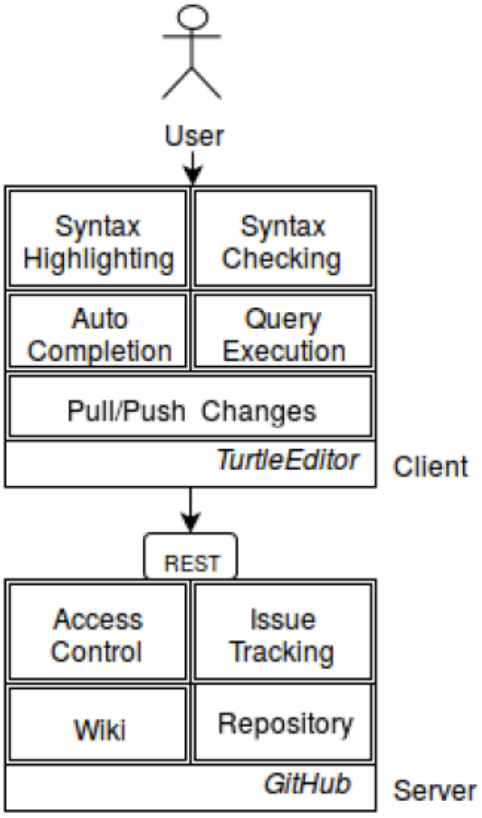


Figure 4.1: TurtleEditor architecture. Figure taken from [15].

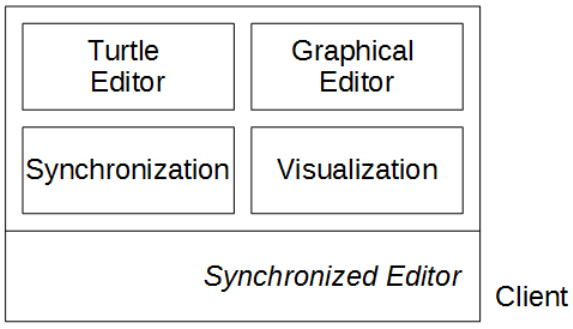


Figure 4.2: Enhanced client.

4.3 Modules

The synchronized hybrid editor is implemented in Javascript and can be run using a web browser. A few Javascript libraries, such as *JQuery*[3], *vis.js*[4] and *split-pane.js*[5], are employed in developing certain functionalities. The role each of these libraries plays in our implementation is discussed in detail in what follows. The remainder of this section explains how the requirements listed in Chapter 3 have been approached.

4.3.1 Graphical Editing

The concepts defined using Turtle code are basically interrelated entities so the apparent structure that can be employed in any graphical functionality is a graph or, in other words, a network. One mature Javascript library that supports manipulating such a structure is *vis.js*. It is an open-source project licensed under Apache 2.0 and MIT and its first working version was released in 2013, since then being constantly upgraded. Our implementation uses version 4.16.1, released in April 2016.

Currently, *vis.js* consists of five components that enable data manipulation and interaction: DataSet, Timeline, Network, Graph2d and Graph3d. We are going to make use of the Network component in conjunction with the DataSet, which was designed to easily handle large amounts of dynamic data. The Network assures the realization of all the graphical functionalities of the editor as it supports a high degree of visual customization and comes with a number of modules that enable a broad manipulation of and interaction with the data.

The initial drawing of the graph expects a DataSet object containing information about entities and the relations between them. In order to construct this object, we make use of the functionalities offered by the *nodes* and *edges* modules. As specified in the requirements, we draw subjects and objects as nodes, therefore, we put their information in the same array that can be manipulated through the *nodes* module, which has several mandatory and optional properties. Some of the properties that we chose to leave with their default values are:

- shape (for URI entities): oval
- border (for URI entities): continuous

[3] `https://jquery.com`
[4] `http://visjs.org`
[5] `https://github.com/shagstrom/split-pane`

- color (for URI entities): blue with a darker shade for the border
- font: *14 Arial*; its size will determine the size of the node (also depending on the amount of text contained in the label)

The properties which receive specific values are:

- shape (for string literals): rectangular with rounded corners
- border (for string literals): dashed (inspired by WebVOWL)
- color (for string literals): yellow with black border (also inspired by WebVOWL)
- id: the URI or the literal value
- label: the URI with shrinked prefix (if any abbreviation is defined) or the literal value; the text is cut if longer than 15 characters
- title: present only when the label gets cut; contains the non-cut version of the label

The predicates are drawn as edges so their information is put in a second array that can be manipulated through the *edges* module. Same as above, there are several settings that must or can be specified. The options having default values are:

- shape: continuous, acts as a spring when physics simulation is on
- color: same as the default color for node border
- font: same as for node

The options that we customized ourselves are:

- direction: arrow pointing towards the object
- smoothing: continuous (for performance reasons)
- id: the URI of the predicate
- label: the URI with shrinked prefix (if any abbreviation is defined)

After creating these objects, there are two arrays of nodes and edges forming the DataSet, which will be passed at network initialization. Besides this, a set of options with extra customizations for each module can also be passed. We will discuss the *layout* and *physics* module in the Visualization subsection.

What concerns us related to graphical editing is the *manipulation* module, due to its features - it supplies an API and an optional GUI for altering the data in the network. When the manipulation system is enabled through the options given at network initialization (which will always be the case in our implementation), an *Edit* button is shown in the top left corner of the graphical view (see Figure 4.3(a)). If this button is clicked, then a toolbar is displayed, containing multiple manipulation settings for the graph elements. The toolbar can be closed in order to go back to the state when only the *Edit* button is displayed, in order to keep the complexity of the user interface low. As it can be observed in Figure 4.3(b), when the toolbar is shown, the height of the view dedicated to the graph display gets reduced, which most of the times is not desired, especially when the visualized structure is large or when a small screen device is used.

The settings available on the toolbar depend on the user interaction with the graphical view. When no elements of the network are selected, only the *Add Node* and *Add Edge* buttons are available. When a selection is made, extra two buttons are displayed, for editing or deleting the graph element. The edit function depends on the type of the selection, as shown in Figure 4.3 (c) and (d). In what follows, we describe every button that is part of the manipulation toolbar:

1. **Add node**: clicking this button has as effect hiding the current settings and displaying instead a *Back* button and an informative label, as in Figure 4.3(e). The user is required to click an empty space in the graphical view in order to choose a position for the new node. Additionally, a form will appear, asking for a text value that is going to represent the node's label. The user can choose to type in and proceed with the node creation or abort the entire process. The position and the label are further passed to a callback function which will handle introducing the new information both in the underlying data structure and in the graphical network. There are a few points to be explained here regarding the value of the label:
   - it can be an URI in either plain format, or with shrinked prefix, or with no prefix. In the latter case, the base prefix will be prepended if there is any given in the code view.

- if the new node is desired to be a string literal, then the text has to be enclosed in quotes. A tooltip is offered in order to make this option clear.
- if the newly introduced label already belongs to one of the existing nodes, the creation process will not be triggered and an error message is displayed instead.

2. **Add edge**: this function is pretty similar to the previous one. Clicking it will determine a transformation of the toolbar as in Figure 4.3(f), informing the user that the new edge has to be dragged from one node to another. When this is done, a form will appear asking for a label. What differs from adding a node is the information passed to the callback function: the ids of the newly linked nodes instead of the position. In addition, the prohibition of introducing already exiting labels is not present anymore, as duplicate edges are allowed.

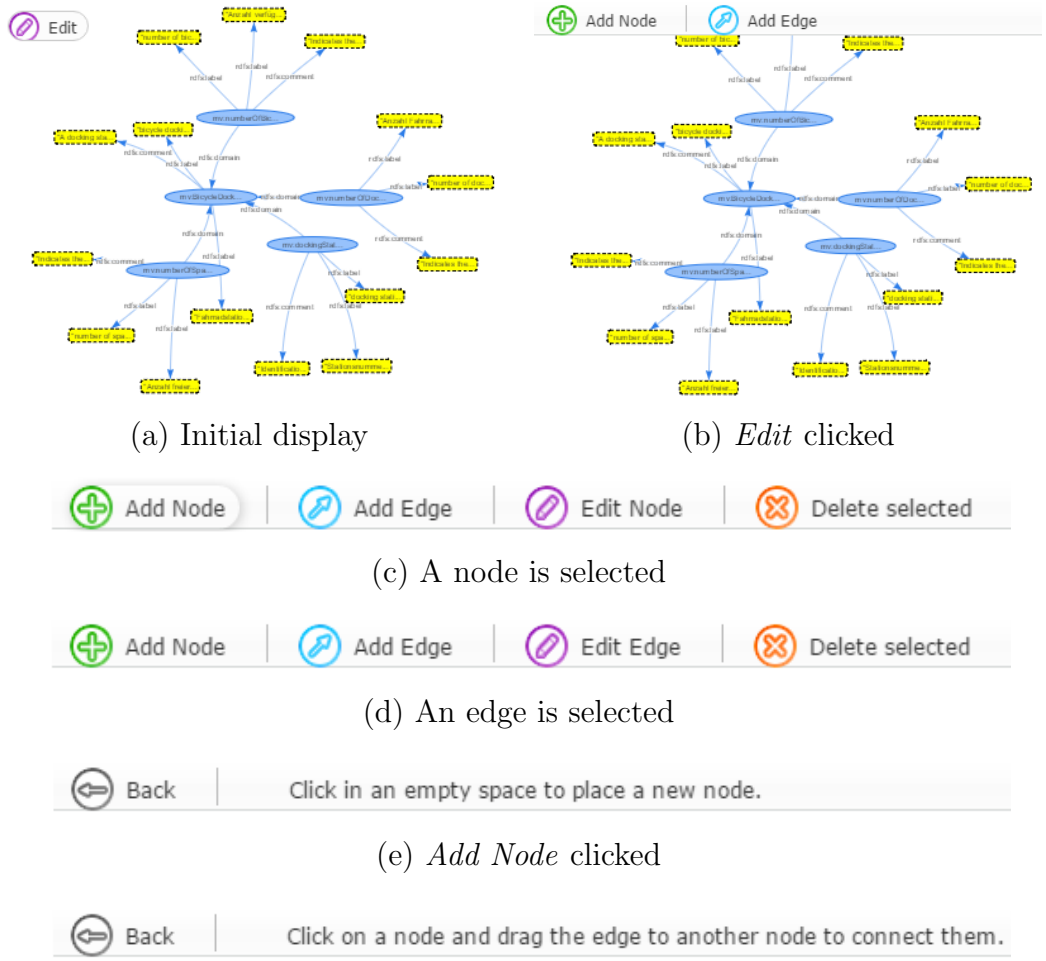
3. **Edit node**: this button is available only when a node is selected. A form similar to the one in the case of adding a node will be shown, containing the current node's label value in the label input. The user can modify it and save the new value or abort the process. All the points made for the *Add Node* function regarding the label value, apply here as well.

4. **Edit edge**: this button is available only when an edge is selected. Like in the previous case, editing an edge actually refers to modifying its label and so, for the new value, the same points apply as in the *Add Edge* case.

5. **Delete selected**: this button is available only when a network element is selected. When a node is deleted, the edges that are linked to the node also disappear. Deleting a node or an edge may imply leaving other nodes disconnected from the graph so clicking this button will trigger the display of a form asking the user if these nodes should be kept or not. We believe this approach might be useful when the user desires to form other triples with these nodes so recreating them will not be necessary in this case. The form offers three possibilities: keeping the possible "orphaned" nodes, discarding them, or aborting the entire deletion process. When the operation is carried on, the ids of all the nodes that need to be deleted are passed to a callback function that will handle removing them from both the underlying data structure and the graphical network.

What has been discussed up to this point regards the network initialization when a set of data is given, i.e., when a file is loaded in the text editor. A graph can also be drawn from scratch and the synchronization module will assure populating the code view with the corresponding elements. However, prefixes can only be added textually.

For every function presented above, extra procedures are carried on with respect to synchronization. We will elaborate this aspect in the following subsection.



(a) Initial display

(b) *Edit* clicked

(c) A node is selected

(d) An edge is selected

(e) *Add Node* clicked

(f) *Add Edge* clicked

Figure 4.3: Different states of the graphical manipulation toolbar, depending on the user interaction with the graphical interface.

4.3.2 Synchronization

The synchronization module was designed to keep both editors updated, meaning that the changes in one view are automatically reflected in the other view, without user interaction. The updates are made only if the modifications pass certain rules of correctness, as a measure to prevent the propagation of errors between the two editors.

The implementation of this functionality took off from the idea of maintaining an underlying model as a common representation for the content of each editor. The logic always keeps two versions of the model: the new one, created when one view is modified and the changes are not yet reflected in the other view, and the one before, when both editors were consistent with the model. As a result, when textual changes are detected, the new model is constructed and a comparison with its older version is performed. In case any changes are detected, they will be transmitted to the graphical view. Similarly, the visual modifications are used to build a new model which will be further turned into text. Figure 4.4 displays the general synchronization process. In what follows, we will describe in detail how the changes propagation occurs on each side.

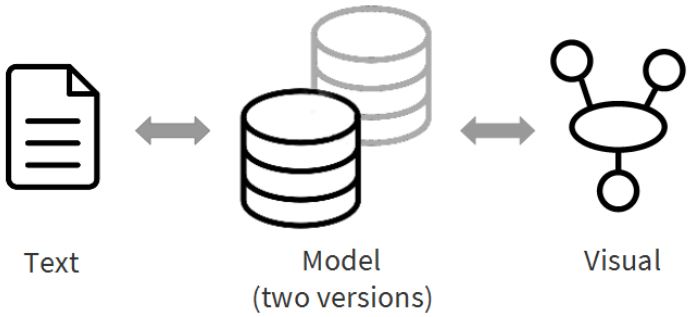


Figure 4.4: A broad view of the synchronization process.

As mentioned in Preliminaries, the code editor is managed by the *Codemirror* library. Changes detection is handled by this library, which triggers an event every time the editor content is modified. The propagation of these changes towards the model relies on the parsing functionality offered by the *N3.js* library. With each change event, the Turtle code is parsed and in case the new version is syntactically valid, the triples that were found are returned. Then, these triples are fetched in order to be stored into an array that will play the role of the underlying model. A triple is basically a Javascript object with three fields ("subject", "predicate" and "object"), where the value of each field is a string representing the URI of an entity.

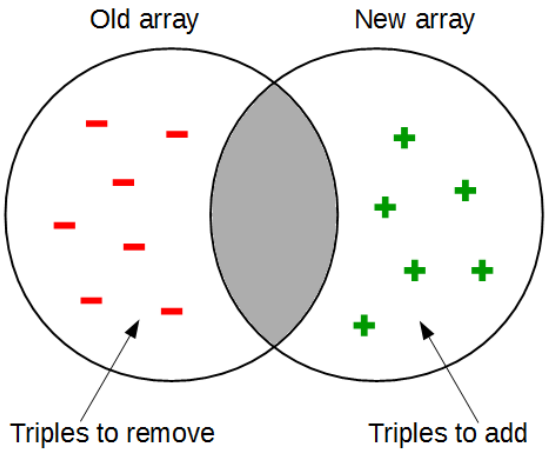


Figure 4.5: Symmetrical difference of the old and new array of triples.

In order to send the changes to the graphical view, we need to compare the array obtained at the parsing step with its older version (the one before these changes were performed). For this, a symmetrical difference is calculated between the two arrays (see Figure 4.5). The triples that are in the old array but not in the new one will be removed and, accordingly, the triples that are in the new array but not in the old one will be added. An observation to be made here is that there can be three types of changes occurring in the code: insert, update and delete. A special case is the *update*, which will be treated as a composite operation of a delete followed by an insert. Therefore, the triples that were updated will be first removed from the graph and then added with the new values. An update process basically means changing the URI of an entity and since the URIs actually represent the node ids in the network, they cannot be modified. Therefore, in order to keep the consistency between a node's id and its label (which is usually its URI in prefixed form), we considered necessary to remove the nodes that had their URI modified and re-add them with another id represented by the new value.

When updating the graphical view due to textual changes, a set of rules are involved, depending on the operation that is being executed:

1. Removing triples:

    - subjects and objects (represented by nodes) are removed only if they have exactly one edge or if they are disconnected from the graph (no edge)
    - predicates (represented by edges) are always removed

2. Adding triples:

    - subjects and objects are added only if they do not exist already
    - predicates are always added (the array cannot contain duplicate triples)

The inverse synchronization (from visual to text) is triggered every time the user changes the structure of the network. Depending on the modification, different operations (insert, update, delete) are executed on the model, represented by the array of triples. The result is a new version of the model that needs to be further translated into Turtle code. This operation is handled by the *N3.js* library, which features a *Writer* object that serializes one or more triples (given as an array) into an RDF document, where the default format is Turtle. One observation is that the triples are written in the order they are stored in the model so, for keeping the code view consistent, the triples' order must be preserved within the array.

Below, we will explain how pushing changes into the model works in the case of each graphical modification:

1. Creating a node - the node is inserted graphically into the network but no updates occur on the model as the node has no edges yet, therefore it forms no triples. Due to the lack of a reasonable representation in the code of a node that is disconnected, we considered that the best approach, in this case, is to make this element known to the code view as soon as it gets linked to another node. Regarding the label of a new node, it will be shown as the short version of its URI (with shrinked prefix) if any abbreviation is provided. If the value is given in short version but without a prefix, the base prefix will be prepended, if any available in code. Otherwise, the label will be shown as introduced by the user. If the label exceeds 15 characters though, the text is truncated and the complete value will be available as a tooltip.

2. Creating an edge - linking two nodes through a new edge is equivalent to creating a new triple. Depending on the direction of the edge, the type of each of the two nodes can be determined. The convention is that at the base of the arrow is the subject and at the tip - the object. Drawing an edge from a literal is prohibited through an error alert, as literals can never be subjects. Once we have determined the elements of the triple, we can insert it into the model. In order to avoid having the same subject appear multiple times in the code, and thus preserve the Turtle layout, we considered the following: when inserting a new triple

into the array, we search for the last triple having the same subject and we introduce the new one exactly after it. If no such triple was found, the new value is inserted at the end of the array and, therefore, it will appear at the end of the code view.

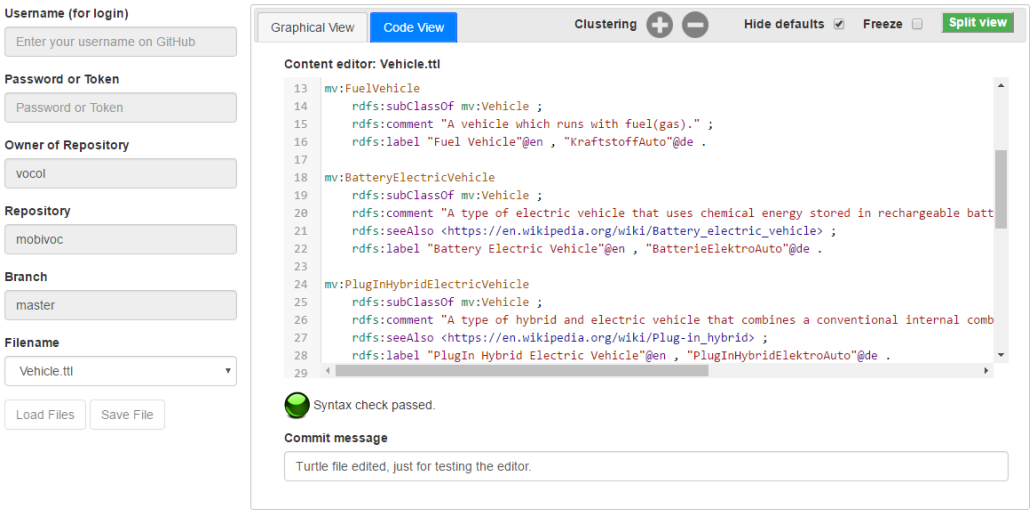
3. Editing a node or an edge - this operation refers to the modification of an entity's URI. The triple in which it is contained is searched in the model and, depending on its type (subject, predicate or object), the corresponding field of the triple receives the new value (the existing URI string is replaced with the given one).

4. Deleting a node - when a node is removed, its edges also disappear from the network. As an edge defines a triple, the number of triples to be removed from the model is equivalent to the number of edges that get deleted together with the node. After these triples are identified and the model is updated, the result is that the code view will erase all triples containing the deleted node as a subject or an object.

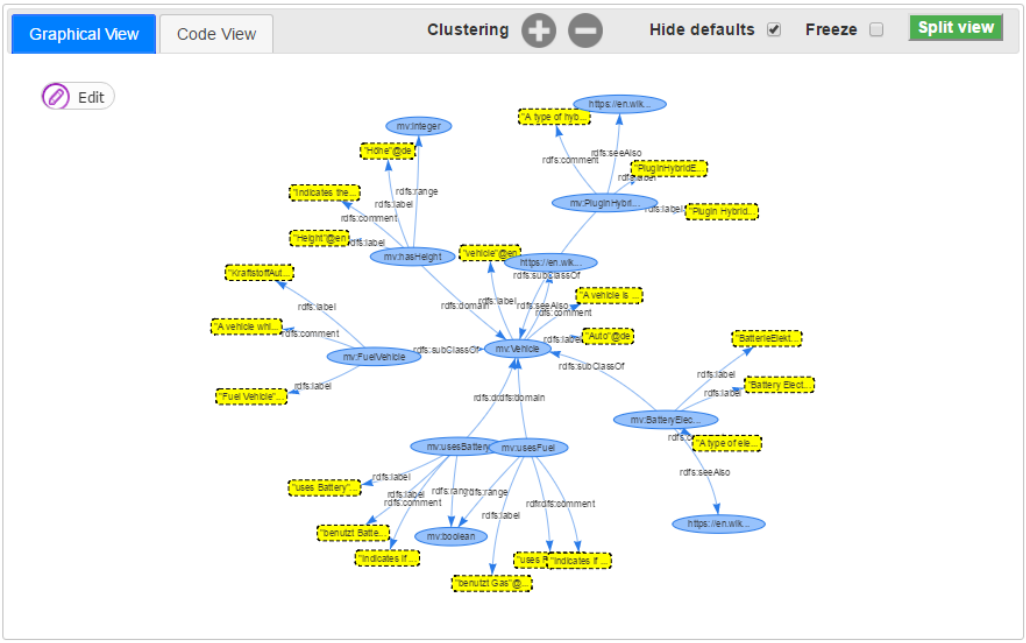
5. Deleting an edge - this is equivalent to removing exactly one triple, therefore one element gets erased from the array.

Some of the operations performed on the array of triples can be quite expensive when the graph is large (thousands of nodes and edges). The symmetrical difference, in particular, can reach a quadratic complexity. In order to improve the time performance, we considered other data structures for storing the triples. Since each triple in the model is unique, we thought of using sets. After performing some operations on this data structure with triple objects, we concluded that the costs vary to a very low extent so we decided to continue working with arrays due to their ease of use in Javascript. The only difference was made by structures that are optimized to work with RDF data as triple arrays. One such object is offered by the *N3.js* library. It is called a *store* and it allows storing triples in memory and finding them fast. The downsides are that it does not allow inserting a triple at a certain index and it does not preserve the triples order as they are parsed from the code. As a compromise, we decided to use stores only for calculating the symmetrical difference and continue performing the other operations on arrays.

In order to easily track the results of the synchronization operations presented above, the two editors need to be simultaneously visible. Initially, we followed the interface layout offered by TurtleEditor (an example can be found in [15]) and we created a tabbed view so that the user can switch between the two editors. Figure 4.6 shows the user interface when each of the tabs are active.



(a) Code view



(b) Graphical view

Figure 4.6: Graphical user interface of the two editors in tabbed view.

Since the synchronization operates instantly, the user should be able to supervise changes in both views in parallel. Therefore, we also implemented a split view that can be activated through the green button placed on the top right corner of the tabbed view. This removes the left side containing the forms needed for GitHub interaction and puts together the views that were previously accessible only in tabs. The editors are separated by a movable bar so each view can be extended in width as much as the screen allows. In order to go back to the tabbed view, a green button is available on the top right corner, as shown in Figure 4.7. The splitting functionality is handled by the *spli-pane.js* library[6], which looks for HTML containers having set a certain CSS class that defines them as components of the view.

Another feature available in the split view is term highlighting. When a node is selected in the graphical view, the effect is that all of its occurrences in the code are marked with a light-green background. In addition, the text editor updates its view so that the first line where the entity appears can be visible. Moreover, the highlights are available on the scrollbar as well, enabling the user to easily navigate between multiple occurrences of a term (see Figure 4.7).This functionality is implemented as follows: when the user clicks on a node in the network, an event is triggered and data regarding the selected entity is passed to a handler function. Then, the label of the entity is searched within the code using pattern matching and the corresponding pieces of text are marked employing some of the functionalities offered by *Codemirror*. The highlights on the scrollbar are implemented using the same library with one of its add-ons - *matchesonscrollbar.js*.

[6] `https://github.com/shagstrom/split-pane`

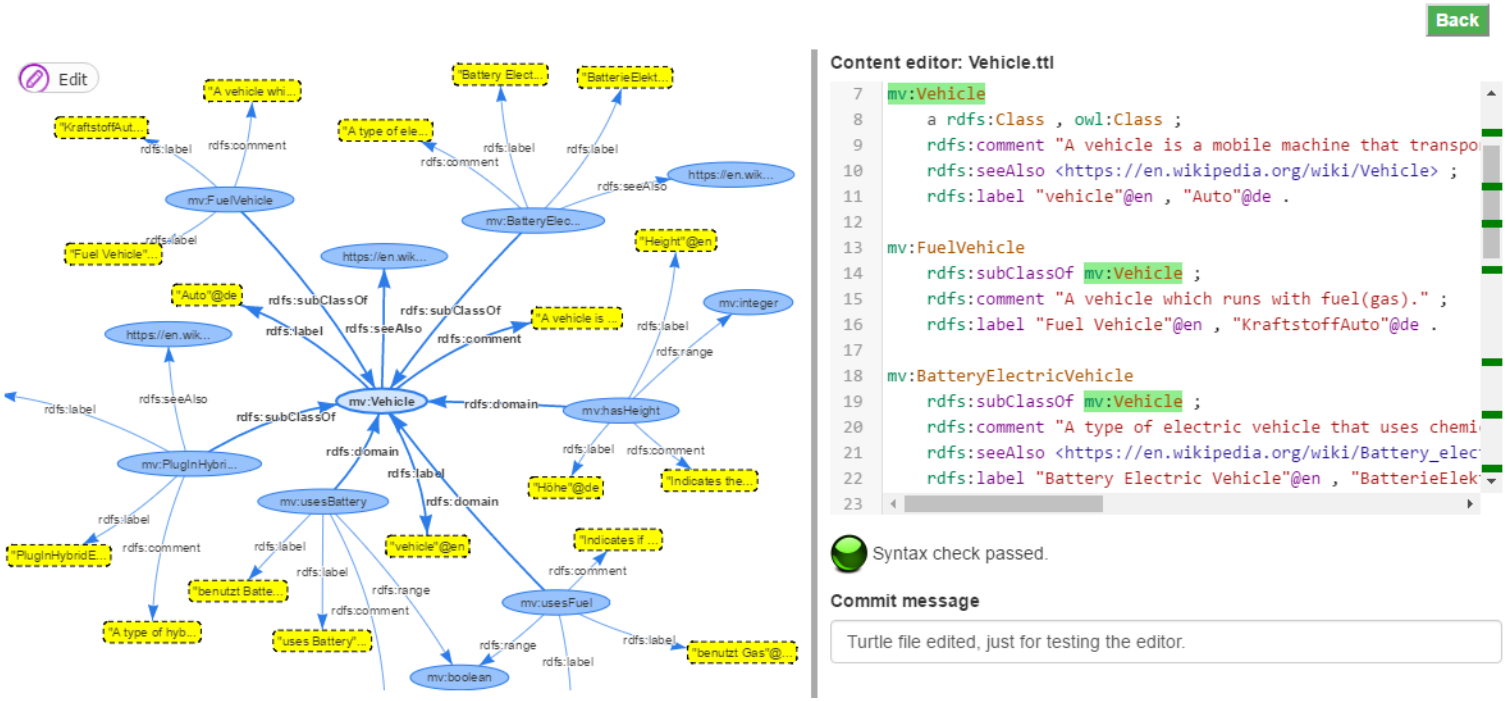


Figure 4.7: Graphical user interface of the split view.

4.3.3 Visualization

The visualization of semantic data can become a quite complex task when dealing with large vocabularies. Certain techniques need to be employed in order to keep the interaction with a graphical editor usable and intuitive. Part of this task has been taken care of by the *vis.js* library, through its *layout* and *physics* modules.

The *layout* module governs the initial and the hierarchical positioning. We chose to keep its default values: non-hierarchical, using the Kamada Kawai algorithm [16] for the initial layout. As a result, the graph elements are positioned in a symmetrical way so that the edges have approximately the same length and there are as few crossings as possible between them.

The *physics* module is concerned with the moving simulation and also with the graph stabilization, by forcing the nodes to always return to their precomputed positions, based on certain parameters. The positions are calculated using the Barnes-Hut algorithm [17], which according to the *vis.js* documentation[7], is "the fastest and recommended solver for non-hierarchical layouts". It is a quadtree based gravity model which groups together bodies that are close enough. We modified the default values of the following parameters in order to speed up the drawing of large graphs:

- gravitational constant: -2500; a negative value refers to repulsion, where the lower the value, the higher the repulsion; we decreased the gravity from its default value (-2000) in order to obtain a better visibility for large graphs.

- spring constant: 0.001; the edges are modeled as springs, where the constant refers to how "robust" the spring is; this value will determine the edges to be more "loose" (than the default value, 0.04), also for visibility purposes.

- spring length: 50; it refers to the length of the spring in resting state; we decreased it from the default value (95) is order to avoid a too wide spread of the graph into space.

The last two parameters in conjunction with the continuous smoothing of the edges (see Subsection 4.3.1) turn out to give good results in the case of large graphs, with respect to time performance and aesthetics.

The physics functionalities prove to be useful when it comes to an intuitive interaction with the graph. However, they represent additional overhead for the time performance and even burden the navigation of large graphs by

[7] `http://visjs.org/docs/network/physics.html`

generating lag and slow movements. Moreover, the user might want to manually rearrange the graph layout and drag the nodes at certain positions, without having them automatically rearranged by the physics engine. We enabled this possibility by providing a "Freeze" checkbox which is by default unchecked, meaning that the physics laws are active. This feature is available in tabbed view, as shown in Figure 4.6. For a close-up, please check Figure 4.8. This is achieved by setting the "enabled" flag of the physics module to false.

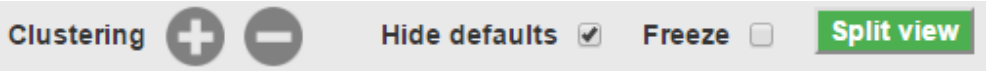
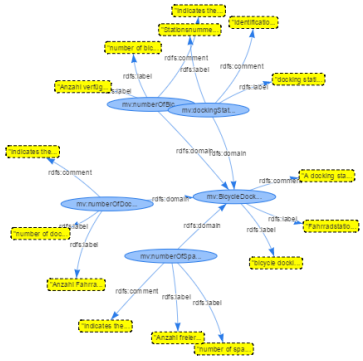


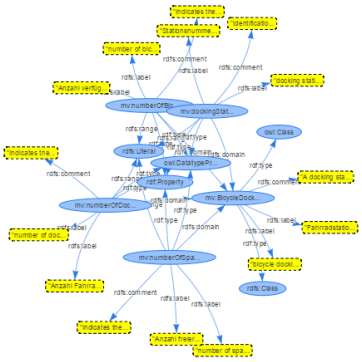
Figure 4.8: Functions of the visualization module.

Another function available in the visualization configurator is called "Hide defaults". This comes as a fulfillment of one of the requirements stating that the user should be able to hide the nodes that are highly connected and, so, generating a lot of clutter. We chose the "defaults" to be all the entities that are part of the *RDF*, *RDFS* and *OWL* namespaces and we made this clear to the user by providing a tooltip. The functionality is implicitly enabled when the page is loaded and, in order to show all the nodes, the user has to uncheck it (see Figure 4.8). When the checkbox is ticked, the default nodes are not removed from the graph, they are just hidden. This is achieved through the *nodes* module, which features a "hidden" flag for each node in the dataset. We filter the nodes by their URI and then we set this flag to true on the result set. Also, an event has to be triggered in order to have the network redrawn and commit the updates. The effect of applying this option is shown in Figure 4.9, where we provided two examples: one quite small graph (33 triples) and one large graph (1573 triples), where the nodes are clustered. As it can be observed, even for small graphs this is useful, as it makes it more clear to see the nodes that are part of the defined vocabulary. For large graphs, hiding these nodes comes as a necessity, as it is extremely cumbersome to distinguish anything among the large number of edges.

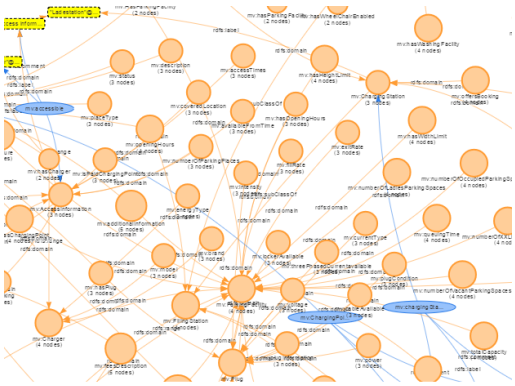
The last and most important functionality in the visualization module is clustering. Its implementation came as a requirement for improving both the interaction with large graphs (over 500 nodes) and the time performance when any graphical operation is involved. *vis.js* comes with support for clustering, offering different methods for grouping nodes together based on certain properties. Out of these, we chose clustering outliers, where an outlier represents a node with exactly one edge , i.e., one neighbor. The method basically groups together these nodes with their respective connected node



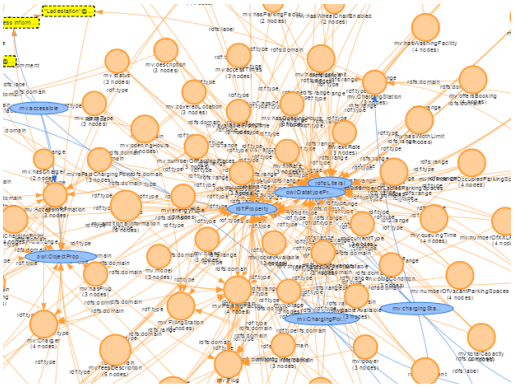
(a) Small graph, no defaults



(b) Small graph with defaults



(c) Large graph, no defaults



(d) Large graph with defaults

Figure 4.9: Hiding and showing nodes from the default namespaces on two different sized graphs.

and it can be called as long as there are still outliers in the graph, meaning that clusters can be joined together with other clusters. The graphical result of this process is that the outlier nodes will disappear from the graph, being replaced by another node which encapsulates them and plays the role of the cluster. This type of nodes can be differentiated through a set of properties:

- shape: circle.
- color: light orange with a darker shade for the border.
- label: composed of two lines, where the first line is the label of the common neighbor node for each outlier and the second line represents the number of nodes that are contained within the cluster; when two

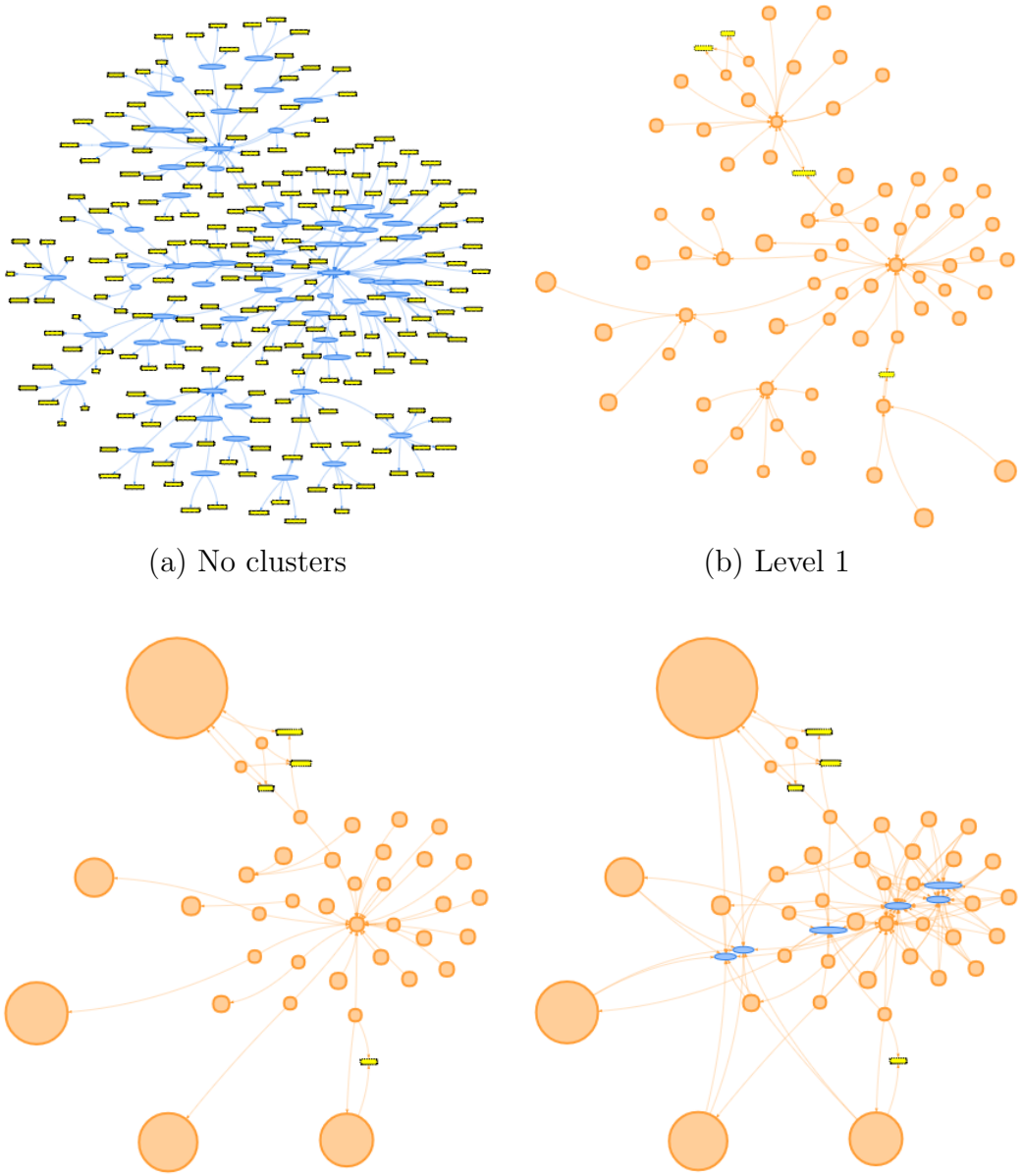
clusters are joined together, the number of children belonging to each one of them is summed. The label is displayed outside the dot shape (as opposed to normal nodes) because, by design, when the label is inside, the length of the text determines the size of the node. When it is placed outside, then the size can be calculated using different criteria.

- size: depends on the number of children and it grows linearly.
- repulsion: increases linearly with the size.

The clustering process can be manipulated using the plus and minus buttons available in the tabbed view (see Figure 4.8). The plus sign clusters any outliers currently existing in the graph, be they normal nodes or already clusters. This process can be repeated as long as there are still outliers in the network. With each repetition, one more level of clustering is added. In order to open the clusters, the minus sign can be used. The nodes will be declustered with respect to their levels, so it may be possible that the minus needs to be clicked multiple times in order to reach the state where there are no more clusters in the graph. Clicking a cluster node also removes one clustering level, the effect being that the node is opened up and the hidden graph elements inside it are set free. Figure 4.10 shows the effect of applying two clustering levels on a graph consisting of 370 triples. Please note that pressing the plus button one more time would have no effect, even though in (c) it seems like there are still outliers in the graph. This is because the default nodes are hidden but they are still part of the graph, maintaining connections with other nodes. After displaying them in (d), we see that there are actually no more outliers and this is the reason why the clustering stopped after two levels.

The clustering process is mainly handled by the *vis.js* library. We only needed to take care of the list of currently existing clusters by managing the insertions and deletions that come with every cluster level. Also, the cluster node properties presented above are set manually as we needed to override the default values in order to suit our needs with respect to intuitiveness.

One last observation needs to be made with respect to the number of clustering levels that are pre-applied when a network is initially loaded. This happens when a file is loaded in the Turtle editor or when code is pasted in the text view. Small graphs (below 500 triples) are displayed unclustered, while graphs consisting of over 500 and 1000 triples have applied one and, respectively, two clustering levels. This is done primarily for reducing the time used for the initial drawing of the network, but also for better visibility and navigation through the graph.



(a) No clusters

(b) Level 1

(c) Level 2

(d) Level 2, defaults displayed

Figure 4.10: Applying the maximum levels of clustering on a graph consisting of 370 triples.

Chapter 5

Evaluation

The synchronized hybrid editor has been evaluated in three steps. We first made a qualitative evaluation in order to determine how the requirements are met and we discovered some gaps that were subsequently corrected. Then, we evaluated our work quantitatively by analyzing the time performance of the graphical functions and of the synchronization module. Finally, we made an user evaluation that helped us assess the usability and the practicality of our editor.

5.1 Meeting the Requirements

While evaluating the functionalities of our editor with respect to the requirements formulated in Chapter 3, we discovered some mismatches and aspects that could still be improved.

Regarding **graphical editing**, the manipulation module for altering data in the network, offered by *vis.js*, pretty much covered our needs. However, we realized that, although the parsing library accepts and recognizes automatically labels surrounded by quotes as literals, the user was not informed in any way about this functionality. We decided that an enhancement of the "Add node" form is needed. Therefore, we added a tooltip that clarifies the possibility of adding two different types of nodes: an URI entity or a string literal.

The **synchronization** module did not require many subsequent modifications either. One inconsistency that we observed is concerned with updating the code view after adding in the network a new node with a label that is not quoted and also not prefixed with any namespace. In this case, we first considered the label as the node's URI and it was displayed in the text view exactly as introduced by the user. This was correct only as long as there

was no base prefix specified in the code. Later on, we added an extra check for a base prefix and, in case it was found, we prepended it to the specified label. Another observation came with regard to term highlighting. Initially, we made a plain string search by the label of the selected node, but we noticed that other entities can be matched too, although they are semantically different. This was the case of URIs that partially contained the searched string. We concluded that a match by meaning is needed and we implemented a pattern search in order to ensure that the highlights are performed exclusively for the selected entity.

The **visualization** was evaluated in multiple stages and we gradually added different improvements for enhancing the user interaction with the graphical representation of the RDF data. Among these, we mention:

- coloring the literals differently: we considered a clear visual distinction was needed between URI entities and string literals.
- truncating the nodes' labels at 15 characters: since the labels determine the size of the nodes, we decided that a certain uniformity is required for a pleasant visualization of the graph.
- hiding defaults implicitly when the network is initially drawn: as pointed out in Subsection 4.3.3, even in the case of small graphs, the visualization is noticeably clearer when nodes from the RDF, RDFS and OWL namespaces are not displayed; therefore, we decided that this feature should be implemented as a default.
- the possibility to hide the editing toolbar: we noticed that on smaller devices, the space occupied by the toolbar could negatively impact the network navigation, so we concluded that the user shall be able to temporarily suppress it.
- increasing repulsion as the size of a cluster grows: initially, the repulsion's value was static, but since the size of the node was calculated dynamically (it increases with the number of children), a certain discrepancy could appear in the graph display; therefore, for improving the visibility, we decided that computing the repulsion dynamically (with respect to the number of contained nodes) is also required.
- tooltips for hiding default nodes and freezing the network movements: additional explanations were considered useful for users which are not familiar with RDF data and methods of displaying it.

5.2 Time Performance

The web editor has been evaluated quantitatively by analyzing the time performance of the initial network load and of the two-way synchronization. The evaluation tests were run using the Google Chrome web browser (version 54), in an environment with an i5 CPU with two physical cores of 2.66 GHz each, 4 GB RAM and 64-bit operating system (Windows 8.1).

In order to perform these measurements, we chose five files containing ontologies of different sizes, as follows:

- O1 - 27 triples
- O2 - 77 triples
- O3 - 178 triples
- O4 - 513 triples
- O5 - 1573 triples

While evaluating the time needed for the initial load of a network (that is, when a file is loaded in the TurtleEditor), we noticed that, in the case of larger ontologies (O3, O4 and O5), approximately 70% of the time is consumed by the physics module, which uses a multiple-step method to calculate the forces exerted between nodes. As a result, we considered three scenarios:

1. physics module is enabled with default values.
2. physics module is enabled with custom values, which are intended to improve the time performance, as explained in Subsection 4.3.3; also clustering is applied for ontologies larger than 500 triples.
3. physics module is disabled.

The results of our measurements, for each of the three scenarios, are presented in Figure 5.1. In the third case, when the physics are completely absent, most of the time is consumed by operations like parsing the Turtle code, manipulating the resulting array of triples and initializing the network with these data. Although it yields by far the best results with respect to time performance, we chose the second scenario for our final implementation. The reason for this decision is that, when the algorithms within the physics module are not applied, the resulting graph is almost unreadable as many nodes are overlapping each other due to the lack of repulsion forces. Moreover, after the network load, the physics can be disabled in order to decrease

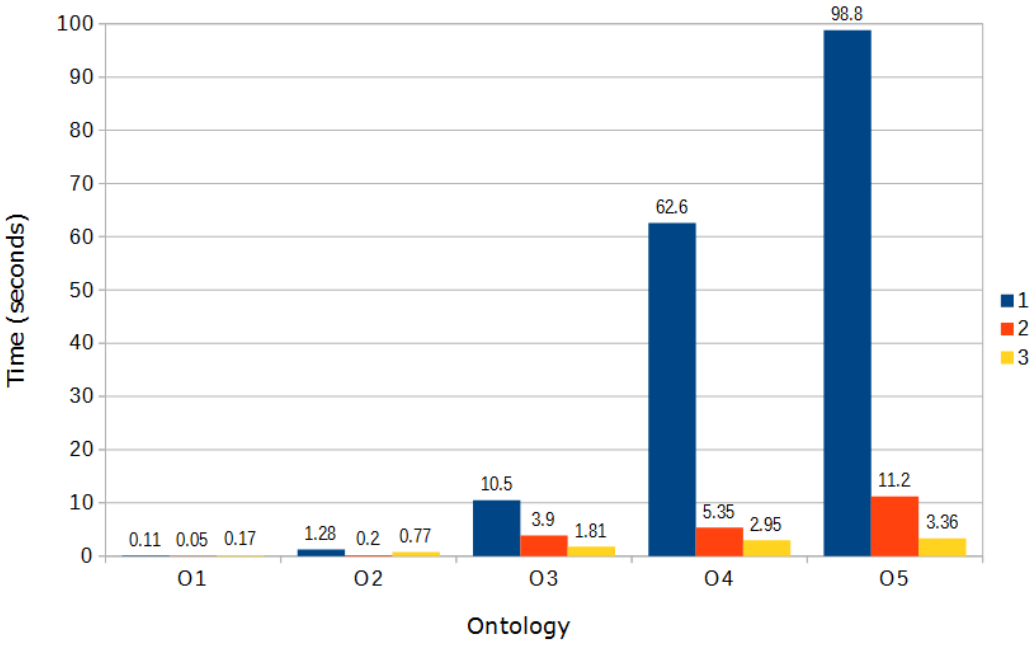


Figure 5.1: The time (in seconds) taken for the initial network load in each of the three scenarios, grouped by ontology.

the memory consumption of the web browser. At the same time, the layout that was previously calculated when the module was active, is kept.

In order to evaluate the time needed for updating the graphical view as a result of a textual modification, we considered two situations regarding the data structure that is used for performing the symmetrical difference between the old and the new triples. As discussed in Subsection 4.3.2, this operation is the most expensive one when it comes to propagating changes towards the visual side. The two structures are: plain Javascript arrays and N3 Stores (offered by the *N3.js* library). The code change performed for each situation was erasing a letter from an entity's URI. The results are shown in Figure 5.2. As we can observe, the time does not vary too much in the case of the first four ontologies. Only for the largest ontology there is a considerable difference (almost one second), which is the reason why we chose to proceed with N3 Stores in our implementation.

The visual to text synchronization is a fast operation for all of the five ontologies. We measured the time taken when the label of a node is edited in the graphical view. The resulting time did not vary significantly among the cases, the difference between O1 and O5 being of only 200 milliseconds. Figure 5.3 shows how this time is distributed among the five ontologies.

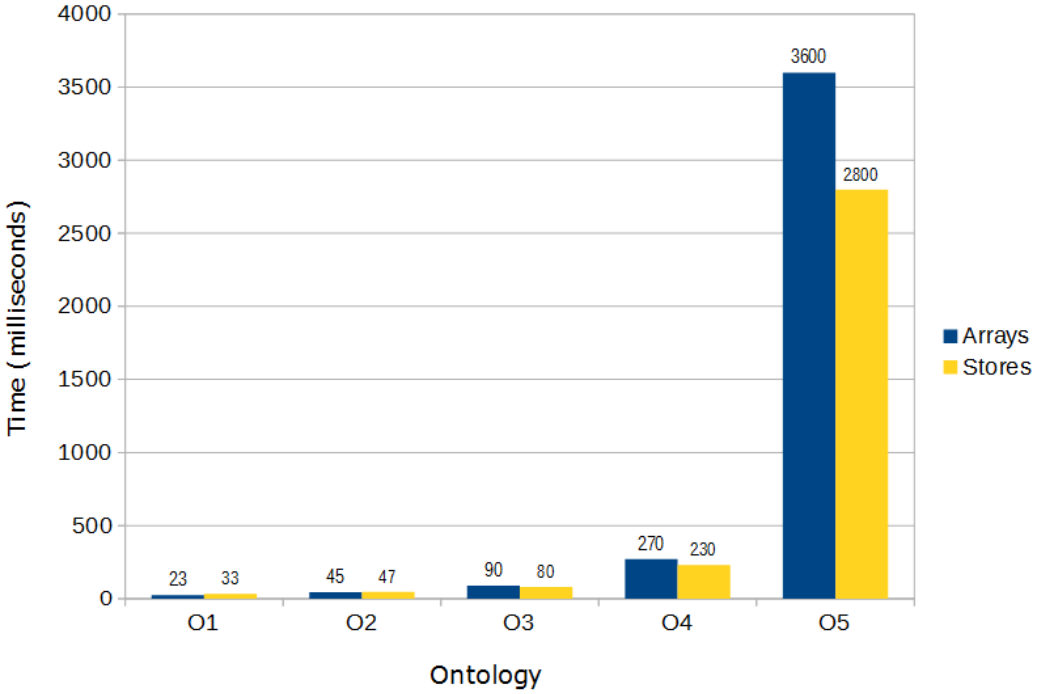


Figure 5.2: The time (in milliseconds) needed to perform the text to visual synchronization with two different data structures for the model.

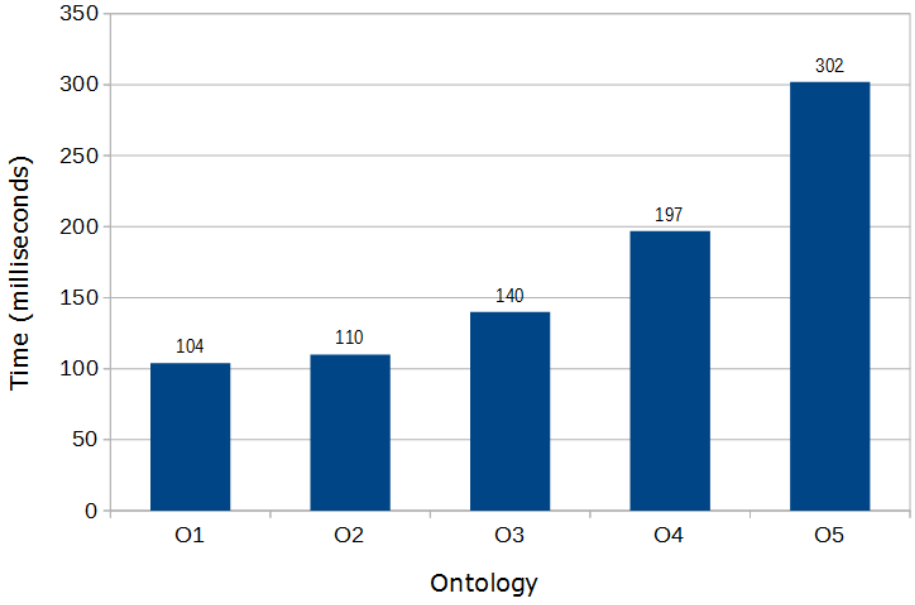


Figure 5.3: The time performance (in milliseconds) of the visual to text synchronization, depending on the ontology size.

5.3 User Evaluation

In order to objectively evaluate the usability and practicality of our editor, we conducted a user study where the participants had to perform several tasks and then respond to a few questions that would assess their interaction with the application.

The experiment was done sequentially, meaning that the activity of each participant was observed and assisted individually, in order to better identify which functionalities may raise usability problems. At the same time, we were able to give hints when the user got stuck on a certain task and, hence, we made sure that the entire test is approached. We noticed that the users were generally identifying the same problems so we decided to end the experiment after the sixth participant, as the gathered data was already sufficient.

Previous experience and technical knowledge play a significant role in the ability to perform this exercise. Therefore, we selected exclusively subjects having a degree in computer science. On top of that, we made a survey with regard to their age and expertise in the semantic web field, placing emphasis on the area of ontology design. We asked each participant to estimate a time period since they got familiar with these domains and, in case they modeled any ontologies before, to specify what types of editors they used. All subjects that confirmed working with a graphical editor, indicated Protégé as their choice. The results are presented in Table 5.1.

Participant identifier	**Age**	**Semantic web**	**Ontology design**	**Editors type**
P1	24	4 years	2 years	graphical and code
P2	34	1 year	2 months	graphical and code
P3	29	1 year	5 months	only code
P4	29	4 years	3 years	graphical and code
P5	35	10 years	9 years	graphical and code
P6	29	6 months	-	-

Table 5.1: Background data about the user study participants.

The evaluation test consisted of nine tasks which aimed to bring a set of modifications to an already existing ontology. They were structured as follows: first three tasks for familiarizing with the interface, tasks 4 - 8 for

using the graphical editor functionalities and the last one to be performed with the code editor:

1. Load *Country.ttl*. Go to the graphical view and add another language label for *mv:Country* (e.g. "Pays"@fr).
2. Uncheck "Hide defaults" in order to see the nodes from the RDF, RDFS and OWL vocabularies.
3. Click "Split view".
4. Define a new property by adding a node (e.g. "Prop1") and make it of type (rdf:type) *rdf:Property*. Note that the *rdf:Property* node already exists.
5. Click on the new node to see it in the text.
6. Make *mv:Country* the property's domain (rdfs:domain) and *rdfs:literal* its range (rdfs:range). Note that the *rdfs:literal* node already exists.
7. Add a description for the property.
8. Modify the node's label (e.g. "Prop2").
9. Remove the property.

We noticed that the speed to which the participants performed these tasks is correlated to their level of experience, especially when previous ontology design is involved. On average, the time needed to solve the entire test amounted to approximately 15 minutes. Figure 5.4 shows how this time is distributed among the participants.

While observing the participants' activity, we noticed that most of them encountered two problems. First was the edge drawing - except P1, they all expected that the edge will appear if the nodes that need to be linked are clicked consecutively. The actual method (clicking the first node and holding the click while dragging the edge to the other node) appeared unexpected, even though there was an explanation in the manipulation toolbar, as it was shown in Figure 4.3(f). The second problem was differentiating between an URI node and a string literal - except P6, the participants omitted the tooltip explaining that, in order to add a literal, its label has to be enclosed in quotes. Besides these two problems, for participants P1, P3 and P5, it was not clear that, when drawing an edge, its direction will imply which node is the subject and which is the object. Also, P2, P3 and P5 found it cumbersome to locate the already existing nodes in the graph in order

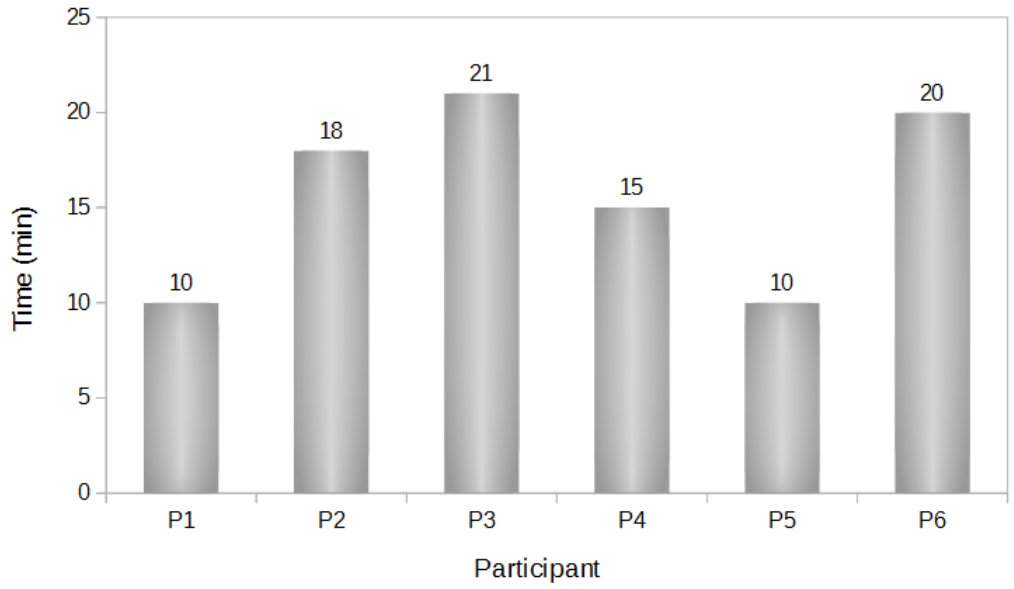


Figure 5.4: Time (in minutes) taken by each participant for solving the test.

to link them with the newly added ones. Finally, P6 had difficulties in figuring out how to edit the graphical view - the "Edit" button needed to be clicked in order to display the manipulation toolbar, as show in Figure 4.3 (a) and (b). To draw a conclusion, we noticed that previous experience with other graphical tools might also have a negative effect, as it is expected that there are certain standards and every editor would work in the same way, which currently is not the case. Nevertheless, problems that were widely encountered among the participants indicate that some functionalities need to be changed in order to provide an usable tool which can become largely used within the ontology design communities.

After performing the tasks, the participants were asked to answer two questions. The first one consisted of three statements where the user had to indicate how much they agree with and the second one was an open-answer question:

1. How much do you agree with each of the following statements?
   (a) The graph visualization represents the information in an understandable way.
   (b) The graphical editor is intuitive to use.
   (c) The synchronization helps with understanding the code representation in Turtle.

2. Do you have any suggestions? (e.g. features you would improve / add)

The level to which the participant agrees to the statements in the first question could be expressed by choosing one of the following affirmations:

- Strongly agree (4)
- Somewhat agree (3)
- Neutral (2)
- Somewhat disagree (1)
- Strongly disagree (0)

In parentheses, there is the score we assigned to each affirmation in order to easily assess the participants' ratings by visualizing them in a chart and, finally, by calculating an average. Statements (a) and (c) obtained an overall score of 92%, while (b) got only 70%. The study results can be viewed in Figure 5.5.

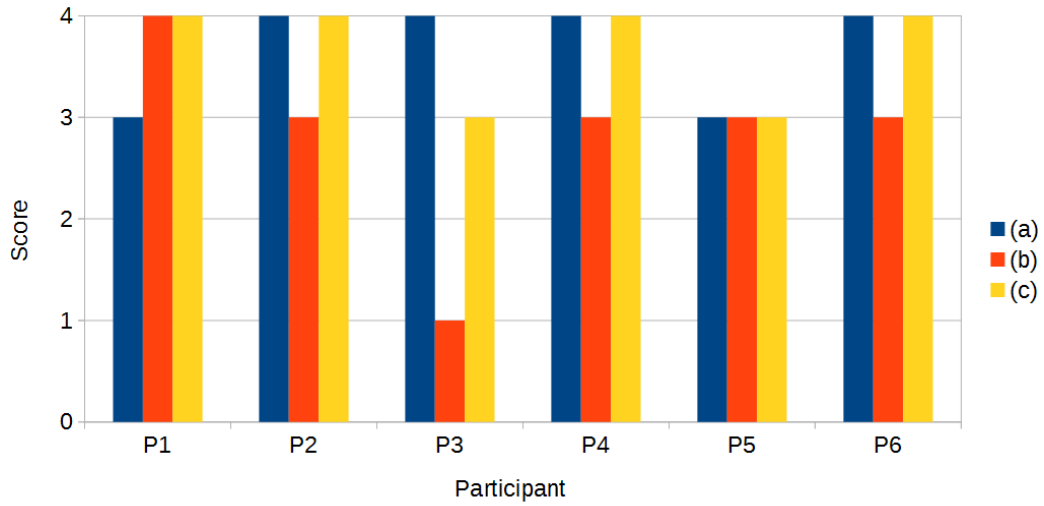


Figure 5.5: The scores obtained by each of the three statements in the first question, grouped by participant.

The answers to the second question helped us gather a set of suggestions for improvements and further development of the editor. The list below presents the recommendations made by the participants in the study:

- P3, P5 and P6 - another way for drawing the edges. P3 suggested that edges could be drawn by clicking the two nodes consecutively. P5 and

P6 preferred having a button available on the node's border, which will trigger edge drawing on click.

- P2 and P3 - another method to differentiate adding an URI node from a string literal. For example, not needing to include the label's text in quotes, rather having a dropdown menu or a radio button for choosing the type of the node.
- P4 and P5 - improvement for node locating in the graphical view. Similarly to the way text is highlighted when a node is clicked, the nodes or the edges also need to be focused depending on the position of the cursor in the text. Alternatively, a search functionality can be implemented for the graphical view, in order to be able to locate nodes or edges by their label.
- P1 and P5 - the options available in tabbed view ("Clustering", "Hide defaults" and "Freeze") should also appear in split view.
- P1, P2, P4 and P5 - auto-completion when typing the label of a node or edge in the graphical view.
- P1 and P4 - improvement for prefix handling as currently it is not possible to add nodes in the graph with prefixes that were not defined in the code.
- P1 - semantic check in both editors for preventing the creation of semantically incorrect triples.
- P1 - filters for different types of nodes in the graph (e.g. hiding all literals).
- P1 - support for typed literals as currently only string literals are differentiated from other nodes.
- P4 - disabling the graphical editing when there are errors in the code.
- P5 - having the ability to copy and paste graph elements.
- P5 - direct editing in the graph. When a node is clicked, editing its label should be enabled directly. In this way, there will be no need to click an edit button in order to modify the label in a form.

These suggestions will be part of the future work for further enhancing the hybrid editor.

Chapter 6

Conclusions and Future Work

In this work, we presented a graphical editor for RDF vocabularies, which synchronizes its content with a code side, provided by the TurtleEditor project. We motivated the necessity of our work by emphasizing the importance of ontology editing in the development of Semantic Web, given the fact that this process is often conducted by domain experts who lack technical knowledge in this field. A graphical editor is an effective method for lowering the barrier with respect to ontology development, especially when there is a lack of such currently maintained tools. The decision for a web application is motivated by the goal to enable domain experts to participate immediately without the need for any software installation on their computers.

By developing a visual editor and offering hybrid editing functionality, a couple of issues needed to be addressed. When having both a graphical and a code editor, the content synchronization comes as a requirement for assisting the teaching process and enhancing the user experience. This has been solved by maintaining a central model, which is basically a translation of the content of each editor into a machine understandable structure that is easy to manipulate. Another issue that needed to be approached was visualizing large RDF graphs because, after a certain threshold, the generated graph becomes hard to manage and navigate. This problem has mainly been solved by implementing a clustering module, which groups nodes having one edge together with their respected connected neighbor. This proves to clear the visualization, especially when clusters can be merged with other clusters.

The editor has been evaluated by analyzing the time performance yielded by different graphical functions. The initial load of the network had a particularly big impact on the user experience and determined us to make a series of modifications. First, we changed the parameters of the module managing the graph layout and the forces exerted between nodes. On top of this, we concluded that the possibility of disabling this module entirely has to be

available to the user, in order to further speed up the interaction. Moreover, we saw another possibility to improve the memory consumption of the browser by making use of the clustering module. When less network elements have to be drawn, also less forces need to be calculated between the nodes, resulting in noticeable time savings. Consequently, RDF graphs exceeding a certain size are initially loaded in a clustered form. Time consumption can be further decreased by finding another method to detect errors in the Turtle code. Currently, the entire code is parsed with each textual modification that is performed by the user. As an improvement, these changes could be isolated to a few lines of code and further logic could be implemented in order to determine the modified triples.

Besides the time performance, we also assessed the usability of the editor by conducting a user study that helped us gather a list of suggestions for future development.

The future work includes further improvement of the time performance and solving the usability issues discovered by the participants in the study. The entire code base is open-source[1] and free to be used and extended by any interested party. Moreover, the hybrid editor will soon be integrated into the larger VoCol environment as a new version of the TurtleEditor, further facilitating, in this way, the collaborative ontology development.

[1] `http://editor.visualdataweb.org`

Bibliography

[1] F. Manola and E. Miller, "RDF Primer." `https://www.w3.org/TR/2004/REC-rdf-primer-20040210`. Accessed: 30-Dec-2016.

[2] P. Mutton and J. Golbeck, "Visualization of Semantic Metadata and Ontologies," *Seventh International Conference on Information Visualization (IV03), IEEE*, pp. 300–305, 2003.

[3] T. Berners-Lee, J. Hendler, and O. Lassila, "The Semantic Web," *Scientific American*, vol. 284, no. 5, pp. 34–43, 2001.

[4] S. Lohmann, S. Negru, F. Haag, and T. Ertl, "Visualizing Ontologies with VOWL," *Semantic Web Journal*, 2015.

[5] D. Beckett, T. Berners-Lee, E. Prud'hommeaux, G. Carothers, and L. Machina, "RDF 1.1 Turtle." `https://www.w3.org/TR/turtle`. Accessed: 30-Dec-2016.

[6] 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.

[7] 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.

[8] D. Steer, "MEG Client Software Review." `http://www.ukoln.ac.uk/metadata/education/regproj/review`. Accessed: 27-Oct-2016.

[9] 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.

[10] M. Brade, F. Schneider, A. Salmen, and R. Groh, "OntoSketch: Towards Digital Sketching as a Tool for Creating and Extending Ontologies for Non-Experts," in *13th Int. Conf. on Knowledge Management and Knowledge Technologies (I-KNOW)*, pp. 9:1–9:8, ACM, 2013.

[11] 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.

[12] A. Telea, A. Maccari, and C. Riva, "An Open Toolkit for Prototyping Reverse Engineering Visualization," *IEEE EG VisSym '02*, pp. 241–250, 2002.

[13] 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.

[14] 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.

[15] 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.

[16] T. Kamada and S. Kawai, "An Algorithm for Drawing General Undirected Graphs," *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[17] J. Barnes and P. Hut, "A Hierarchical O(N log N) Force-Calculation Algorithm," *Nature*, vol. 324, no. 4, pp. 446–449, 1986.