A Synchronized Graphical and Source Code Editor for RDF Vocabularies

Alexandra Similea

Matriculation number: 2776909

November 25, 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 Thesis Structure

Chapter 2

Related Work

In this chapter, we will focus on previous work that is related 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

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 [1]. 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 syntax tree. The resulting tree is merged with the one belonging to the code

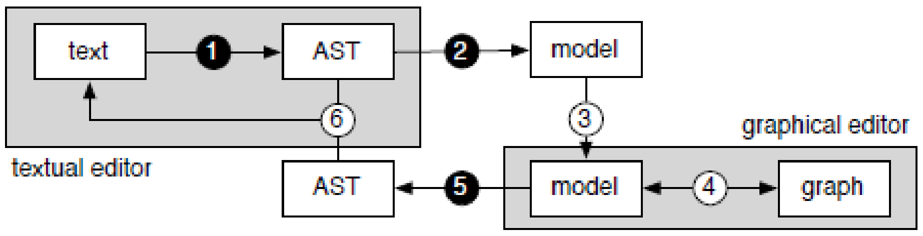


Figure 2.1: Steps involved in the synchronization process: 1) parsing, 2) tree to model transformation, 3) model merge, 4) graphical changes propagation, 5) model to text transformation, 6) tree to text printing.
Figure taken from [1].

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 semantic 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 will consider the idea of having an underlying model, but, in our case, it should be the same for the two views and it should represent the only connection between them. Both textual and graphical data shall be parsed directly into the common model and an update process needs to be triggered.

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 be committed only when they pass certain rules of correctness, the editor

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

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, Notation3 and N-Triple, while the export supports, besides the already men-

[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`

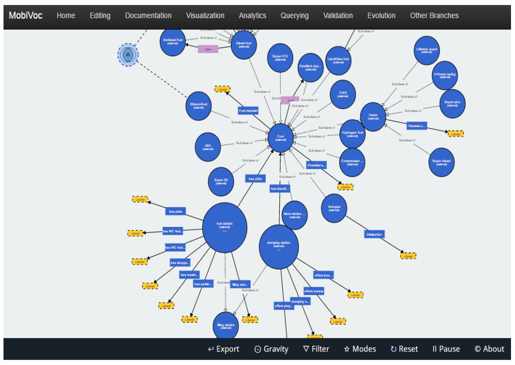
[7]`http://protegewiki.stanford.edu/wiki/OntoViz`

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

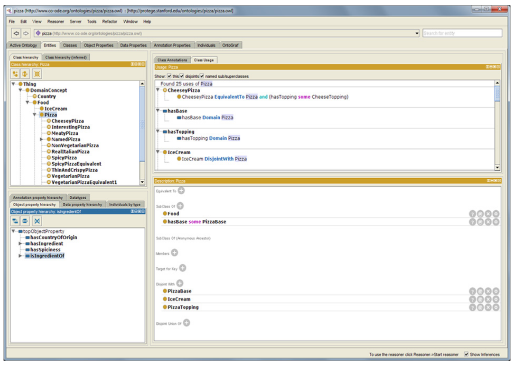
[9]`https://www.w3.org/2001/11/IsaViz/`

tioned formats, also SVG and PNG.

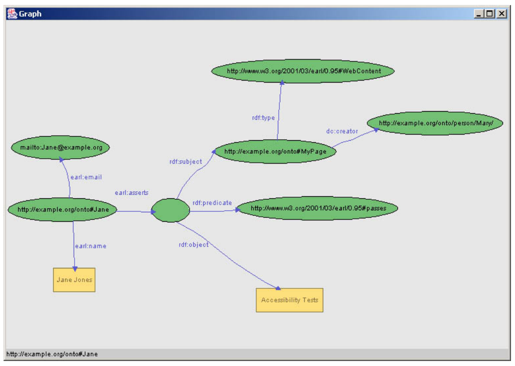
4. RDFauthor is a tool for viewing, creating and querying RDF instance data. It extracts structured information from RDFa-enhanced websites and creates an edit form based on this data. The RDF data model is presented graphically as a directed graph [4]. A visualization example is shown in Figure 2.2(d). 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.



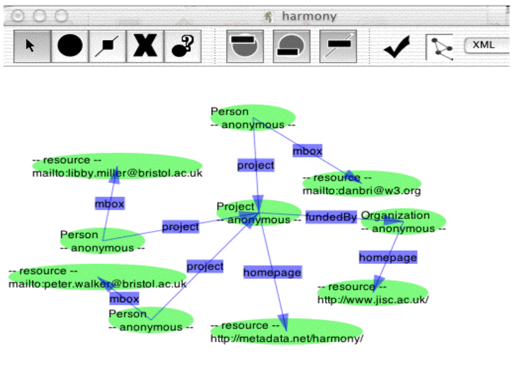
(a) Vocol



(b) Protégé


Graph
http://www.w3.org/2001/03/earl/0.95#WebContent
rdf:type
dc:creator
http://example.org/onto/person/Mary
mailto:Jane@example.org
rdf:subject
http://example.org/onto#MyPage
earl:email
earl:asserts
http://example.org/onto#Jane
rdf:predicate
http://www.w3.org/2001/03/earl/0.95#passes
earl:name
rdf:object
Jane Jones
Accessibility Tests
http://example.org/onto#Jane

(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[10], in order to further customize nodes and edges depending on certain attributes.

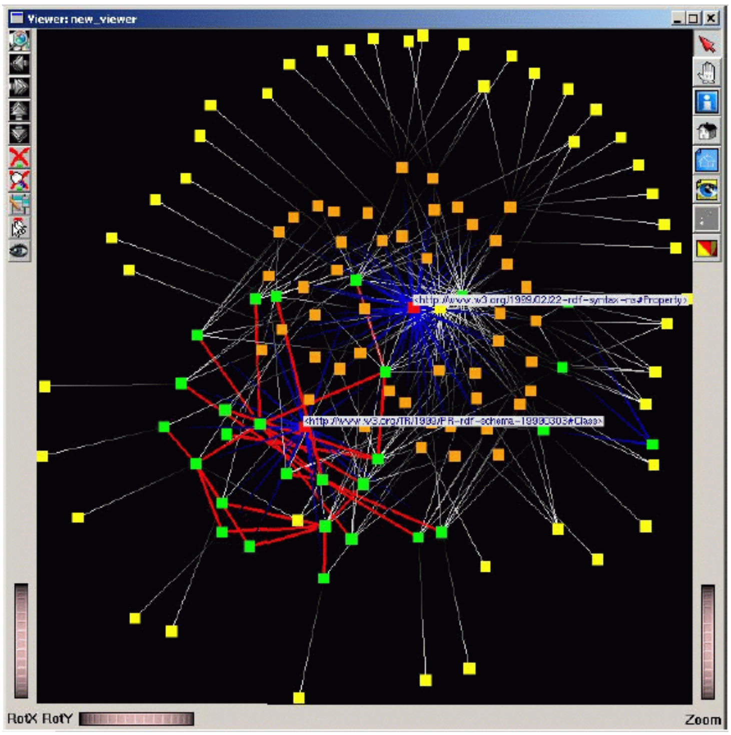

Viewer: new_viewer
<http://www.w3.org/1999/02/22-rdf-syntax-ns#Property>
<http://www.w3.org/TR/1999/PR-rdf-schema-19990303#Class>
RotX RotY
Zoom

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

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

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].

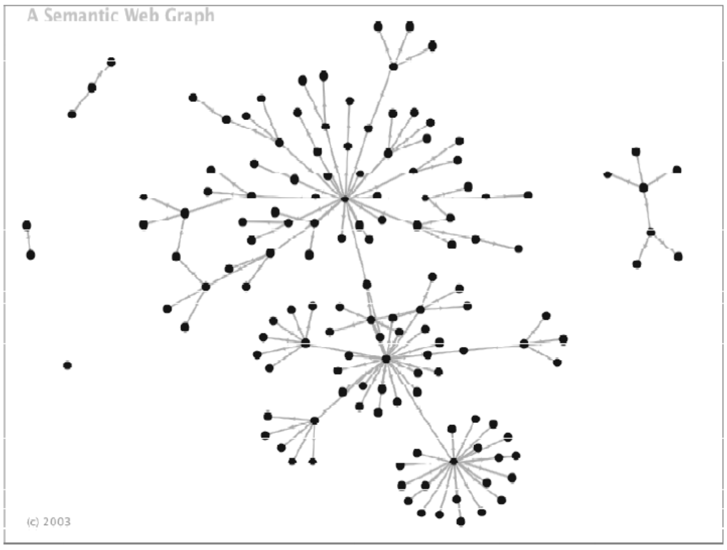


Figure 2.4: Graph layout using spring embedding. Figure taken from [9].

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 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 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 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 application:

- **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`

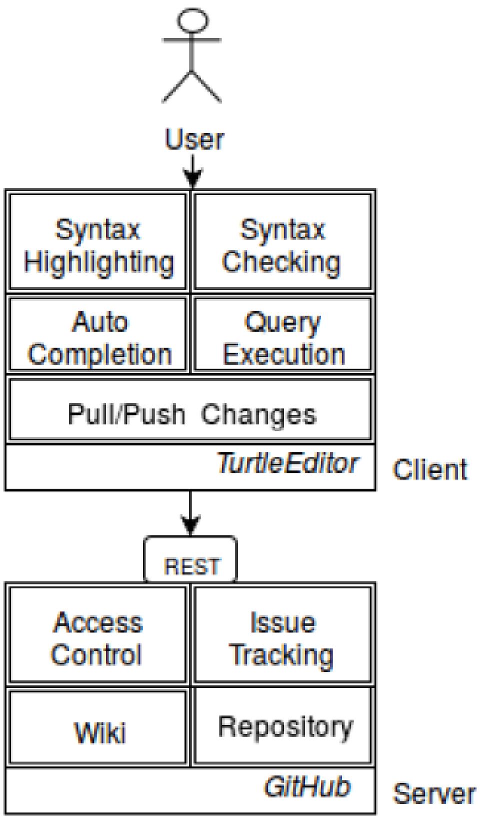


Figure 4.1: TurtleEditor architecture. Figure taken from [11].

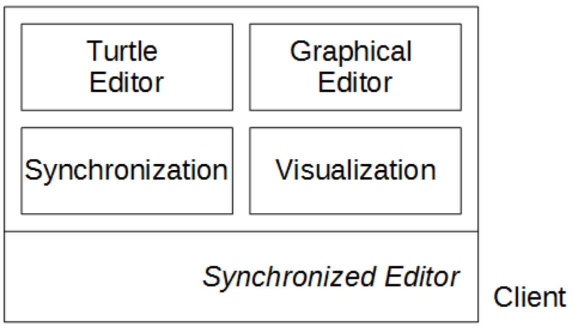


Figure 4.2: Enhanced client.

4.3 Features

The synchronized hybrid editor is implemented in Javascript and can be run using an web browser. A few Javascript libraries, such as *JQuery*[3], *vis.js*[4] and *split-pane.js*[5], are used in order to achieve certain functionalities. The role that each of these libraries plays in our implementation will be discussed in detail in what follows. The remainder of this section will explain 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 obvious 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 that will contain information about entities and the relations between them. In order to construct this object, we will make use of the functionalities offered by the *nodes* and *edges* modules. As specified in the requirements, we will draw subjects and objects as nodes, therefore, we will 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 will be 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 will be put in a second array that can be manipulated through the *edges* module. Same as above, there are several options that must or can be specified and we will start with the ones having default values:

- 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, we will have two arrays of nodes and edges that form 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 capabilities - 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. We decided to not always have it present out of visibility reasons: 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 structure is large or when a small screen device is used.

The toolbar can have different settings displayed, depending 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, the edit function depending on the type of the element that is selected (see Figure 4.3 (c) and (d)). In what follows, we will describe every button that is part of the manipulation toolbar:

1. **Add node**: clicking this button will have as effect hiding the current settings and displaying instead a *Back* button and an informative label (see 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 will 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 will be 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, 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 will be displayed.

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) and also the prohibition of introducing already exiting labels, as duplicate edges are allowed.

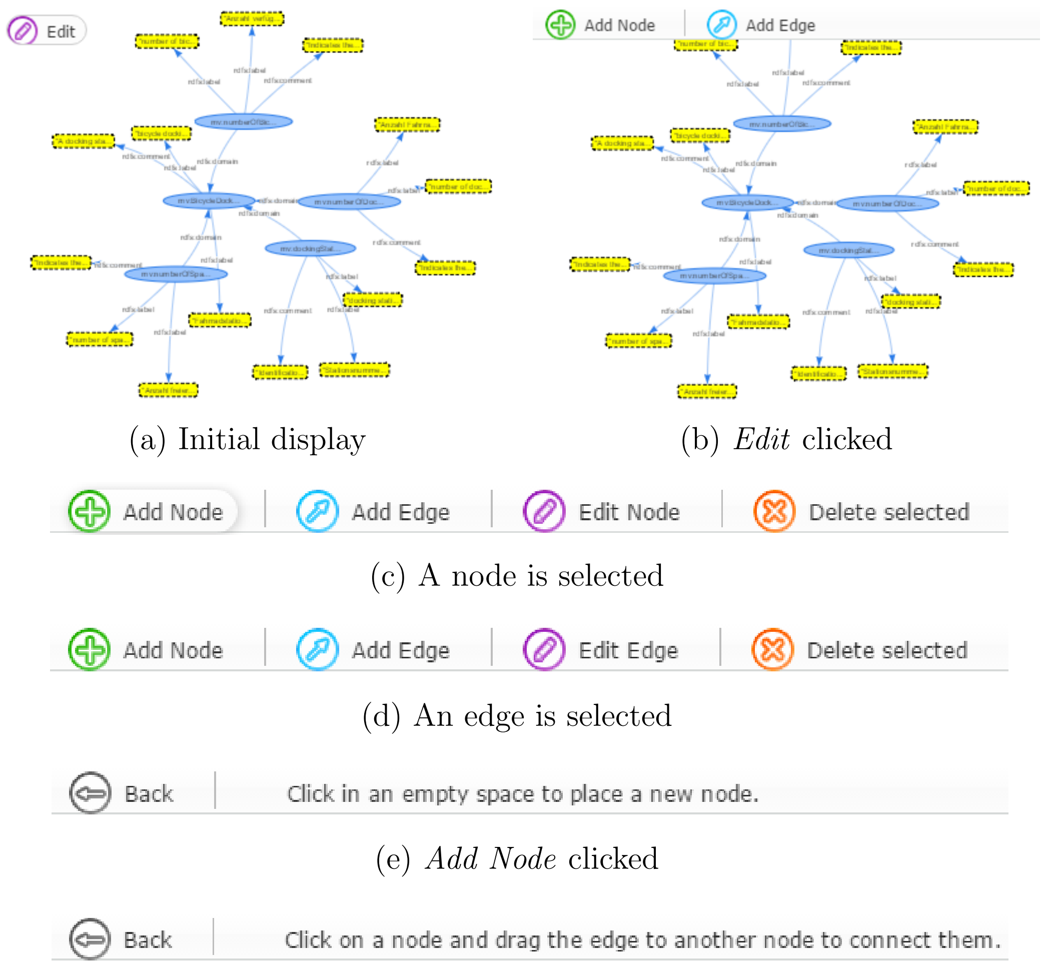
3. **Edit node**: this button is available only when a node is selected. A form similar to the one in the case of adding a node will be shown, containing the current node's label value in the label input. The user can modify it and save the new value or abort the process. All the points made for the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



Click in an empty space to place a new node.

Click on a node and drag the edge to another node to connect them.

(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 with respect to 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 will be 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 which is responsible of keeping the views synchronized. Figure 4.4 shows a broad view of how it works. The model always keeps two versions of itself: the current one, created when one view was modified and the changes were not yet reflected in the other view, and the one before, when both views were consistent with the model. Therefore, when textual changes are detected, they are propagated into the current model, which will compare itself with the version before and then transmit the detected changes into the graphical view. Similarly, the visual modifications are sent to the model, which will create an updated version of itself that will be further turned into text. In what follows, we will describe in detail each way of the synchronization process.

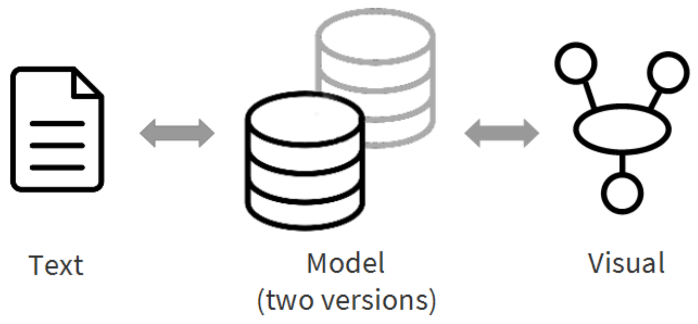


Figure 4.4: A broad view of the synchronization process.

As mentioned in Preliminaries, the code editor is managed by the *Codemirror* library. Changes detection is handled by this library, which will trigger 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 when no syntax error are detected, the trilples that were found are returned. We will fetch these triples and store them into an array that will play the role of our 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.

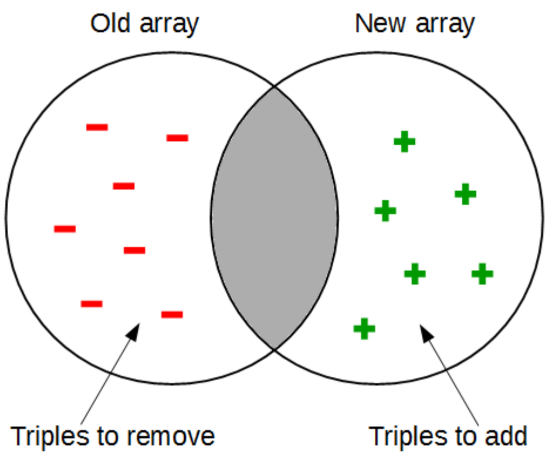


Figure 4.5: Symmetrical difference of the old and new array of triples.

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 this step was realized). For this, we will do a symmetrical difference on the two arrays (see Figure 4.5). The triples that are in the old array but not in the new one need to be removed and, accordingly, the triples that are in the new array but not in the old one need to 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. Un 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 the short version of its URI), 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 interacts with the graphical editor functions presented in Subsection 4.3.1. Depending on this, different operations (insert, update, delete) are executed on the model represented by the array of triples. After the model is updated and we have a new "current version", we need to translate the array of triples into Turtle code. This is handled by the *N3.js* library, which features a *Writer* object that serializes one or more triples (given as an array) as an RDF document, where the default format is Turtle. One observation is that the triples are written in the order they stored in the array so, in order to keep the code consistent, we have to preserve the triples' order into the model.

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

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. We could not find an intuitive way to represent into code a node that is disconnected so 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. As for the label of a new node, it will be shown as the short version of its URI (with shrinked prefix) if any abbreviation provided. If the value is provided in short version but with no 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 will be 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, we can determine the type of each of the two nodes, the convention being 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 will be inserted at the end of the array and, therefore, it will appear at the end of the code view.

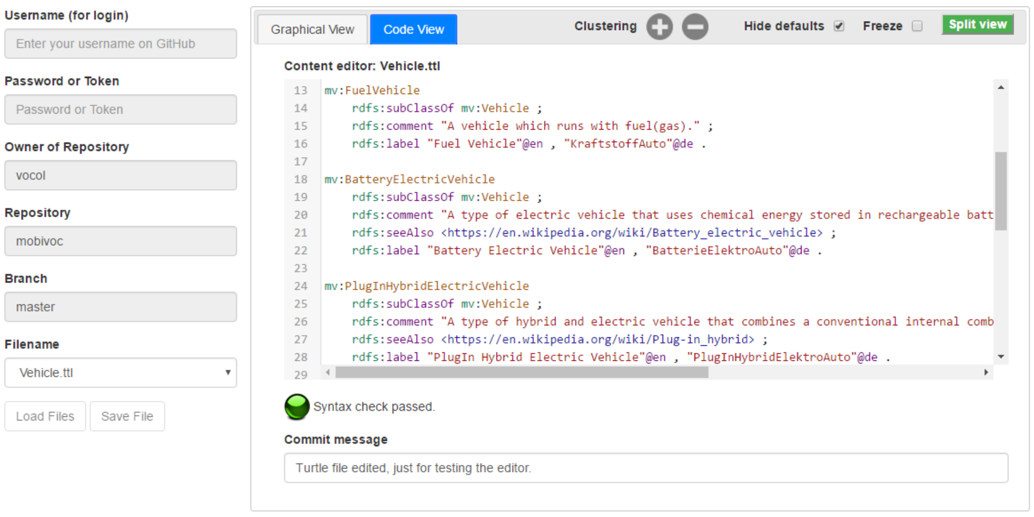
3. Editing a node or an edge - this operation refers to the modification of an entity's URI. The triple in which it is contained is searched in the model and, depending on its type (subject, predicate or object), the corresponding field of the triple receives the new value (the existing URI string is 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 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 have to be simultaneously visible. Initially, we followed the interface layout offered by TurtleEditor (an example can be found in [11]) and we created a tabbed view so that the user can switch



Content editor: Vehicle.ttl

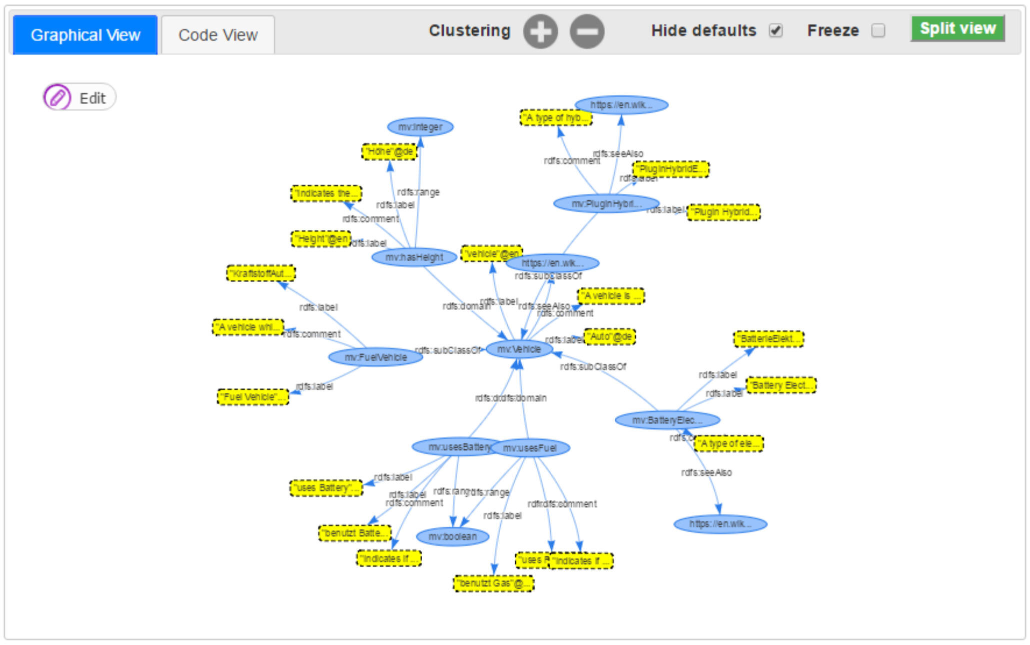
```
mv:FuelVehicle
    rdfs:subClassOf mv:Vehicle ;
    rdfs:comment "A vehicle which runs with fuel(gas)." ;
    rdfs:label "Fuel Vehicle"@en , "KraftstoffAuto"@de .

mv:BatteryElectricVehicle
    rdfs:subClassOf mv:Vehicle ;
    rdfs:comment "A type of electric vehicle that uses chemical energy stored in rechargeable batt
    rdfs:seeAlso <https://en.wikipedia.org/wiki/Battery_electric_vehicle> ;
    rdfs:label "Battery Electric Vehicle"@en , "BatterieElektroAuto"@de .

mv:PlugInHybridElectricVehicle
    rdfs:subClassOf mv:Vehicle ;
    rdfs:comment "A type of hybrid and electric vehicle that combines a conventional internal comb
    rdfs:seeAlso <https://en.wikipedia.org/wiki/Plug-in_hybrid> ;
    rdfs:label "PlugIn Hybrid Electric Vehicle"@en , "PlugInHybridElektroAuto"@de .
```

Syntax check passed.

Commit message

Turtle file edited, just for testing the editor.

(a) Code view



(b) Graphical view

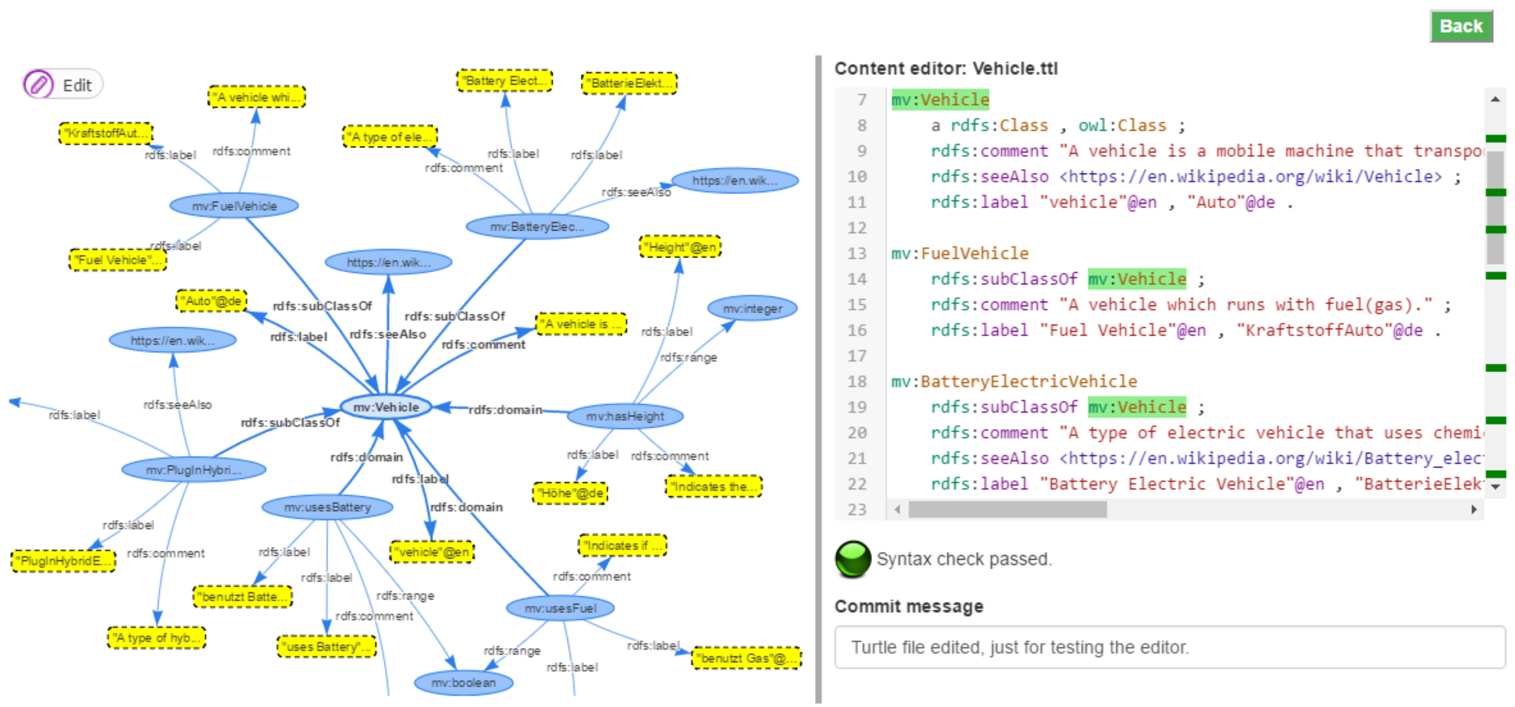
Figure 4.6: Graphical user interface of the two editors in tabbed view.

between the two editors. Figure 4.6 shows the user interface when each of the tabs are active.

We decided that having shown just one editor at a time is not enough for experiencing the synchronization functionalities so we implemented a split view that can be accessed through the green button placed on the top right corner of the tabbed view. The split view basically eliminates the left side containing the forms needed for GitHub interaction and puts together the views that previously were 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. This means that when a node is selected in the graphical view, all of its occurrence in the code are marked with a light-green background and the text editor updates its view so that the first line where the entity appears can be visible. Moreover, the highlights are also available on the scrollbar so that the user can easily move between occurrences (see Figure 4.7). How this works: when the user clicks on a node in the network, an event is triggered by *vis.js* and data about 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`



Content editor: Vehicle.ttl

```
mv:Vehicle
    a rdfs:Class , owl:Class ;
    rdfs:comment "A vehicle is a mobile machine that transpo
    rdfs:seeAlso <https://en.wikipedia.org/wiki/Vehicle> ;
    rdfs:label "vehicle"@en , "Auto"@de .

mv:FuelVehicle
    rdfs:subClassOf mv:Vehicle ;
    rdfs:comment "A vehicle which runs with fuel(gas)." ;
    rdfs:label "Fuel Vehicle"@en , "KraftstoffAuto"@de .

mv:BatteryElectricVehicle
    rdfs:subClassOf mv:Vehicle ;
    rdfs:comment "A type of electric vehicle that uses chemi
    rdfs:seeAlso <https://en.wikipedia.org/wiki/Battery_elec
    rdfs:label "Battery Electric Vehicle"@en , "BatterieElek
```

Syntax check passed.

Commit message

Turtle file edited, just for testing the editor.

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 hierarchical positioning. We chose to keep its default values: non-hierarchical, uses the Kamada Kawai algorithm [12] for the initial layout, which positions the graph elements in an aesthetical way so that the edges have approximately the same length and there are as few crossings between them as possible.

The *physics* module is concerned with the moving simulation and also stabilizing the graph by forcing the nodes to always return to their precomputed positions, based on certain parameters. The positions are calculated using Barnes-Hut algorithm [13], which according to *vis.js* documentation, 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 basically decreased the gravity from its default value (-2000) in order to obtain a better visibility for large graphs.

- sprig 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 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 generating lag and slow movements. Moreover, the user might want to rearrange the graph layout and drag the nodes at certain locations without having them return to their precalculated positions due to active laws of physics. We enabled this possibility by providing a “Freeze” checkbox which is by default unchecked, meaning that the physics module is 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 simply by setting the “enabled” flag of the physics module to false.

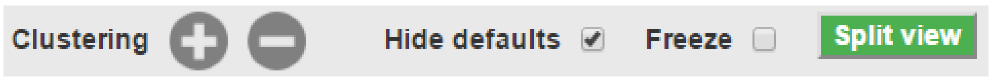
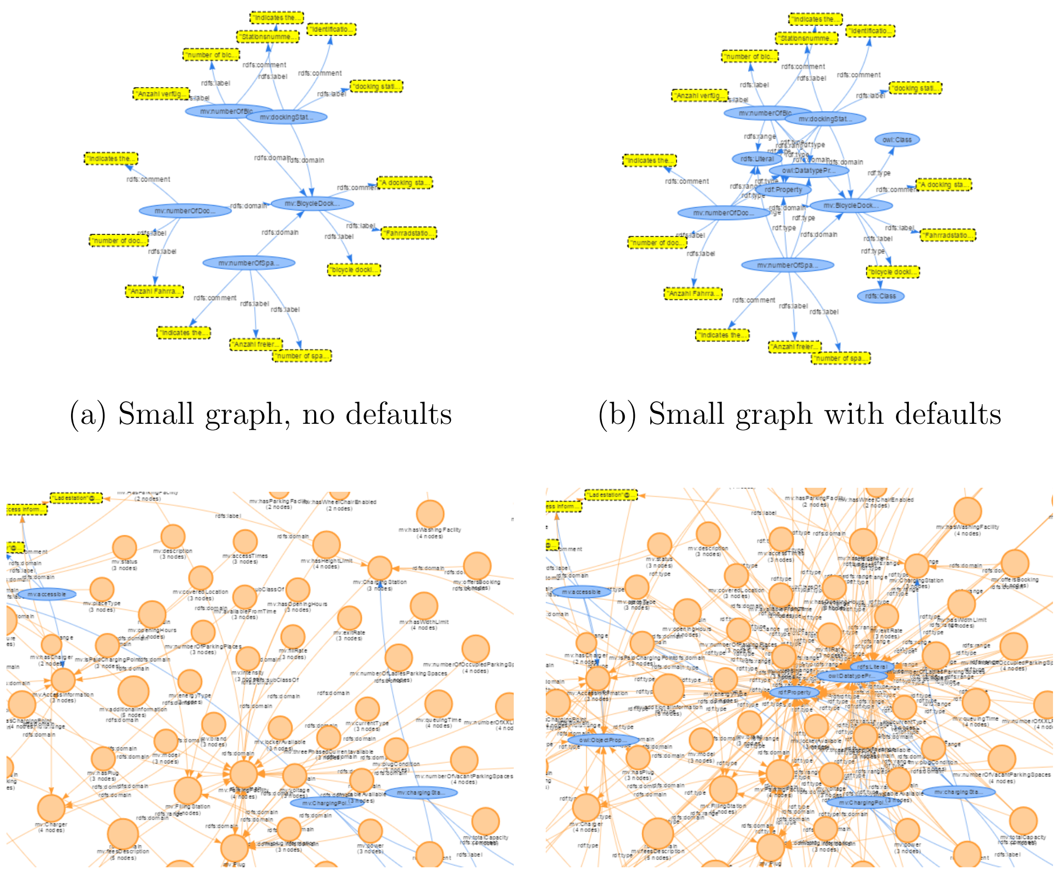


Figure 4.8: Functions of the visualization module.

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 just 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 can be visualized 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 necessity 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 and it can be called as long as there are still outliers in the graph, meaning



(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.

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: dot.
- 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

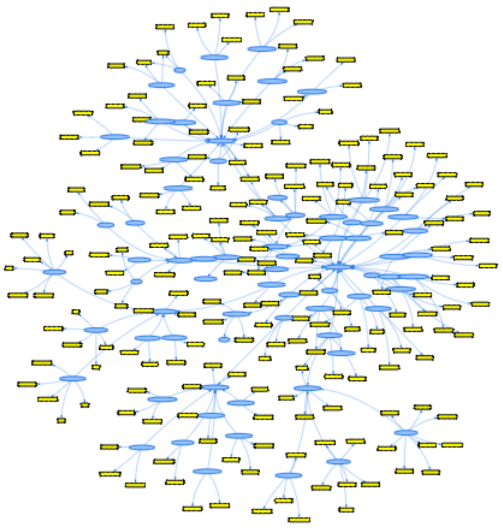
one of them is summed. The label is displayed outside the dot shape (as opposed to normal nodes) because, by design, when the label is inside, the length of the text determines the size of the node. When it is placed outside, then the size can be calculated using different criteria.

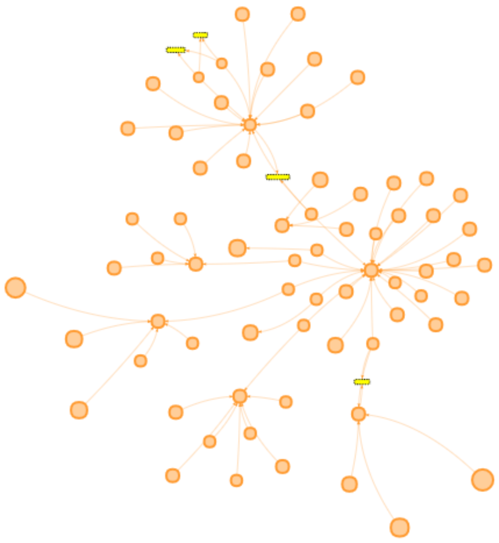
- size: depends on the number of children and it grows linearly.
- repulsion: increases linearly with the size.

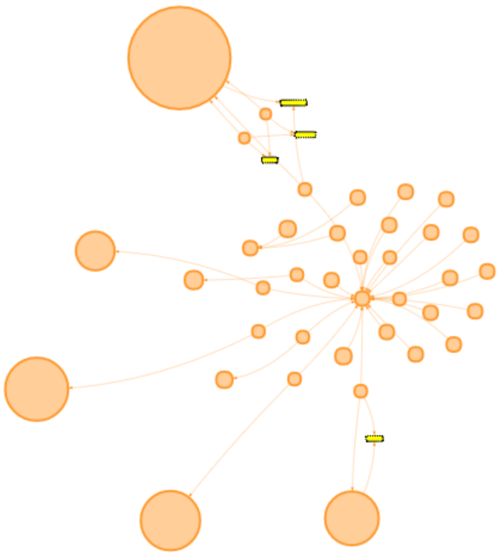
The clustering process can be manipulated using the two plus and minus buttons available in the tabbed view (see Figure 4.8). The plus sign will cluster any outliers currently existing in the graph, be they normal nodes or already clusters. It can be clicked and yield an effect (by adding one more clustering level) as long as nodes with one edge are still present in the network. 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 needed that the minus is clicked multiple times in order to reach the state where there are no more clusters in the graph. Clicking a cluster node will also have the effect of opening it up by removing one clustering level. 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 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 defaults in order to suit our needs with respect to intuitiveness.

One last observation is made regarding the way networks are initially displayed when a Turtle file is loaded into the editor. 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

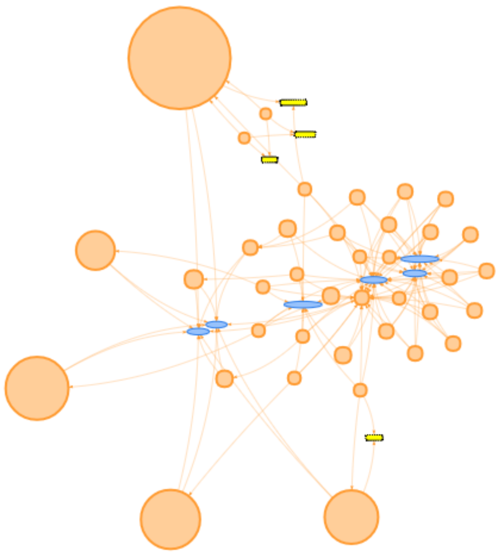

(d) Level 2, default nodes displayed

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 later corrected. Then, we evaluated our work quantitatively by analysing the time performance of the graphical functions and of the synchroization module. Finally, we made an user evaluation that helped us assess the usability and the practicality of our editor.

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.

[12] T. Kamada and S. Kawai, “An algorithm for drawing general undirected graphs,” *Information Processing Letters*, vol. 31, no. 1, pp. 7–15, 1989.

[13] J. Barnes and P. Hut, “A hierarchical O(N log N) force-calculation algorithm,” *Nature*, vol. 324, no. 4, pp. 446–449, 1986.