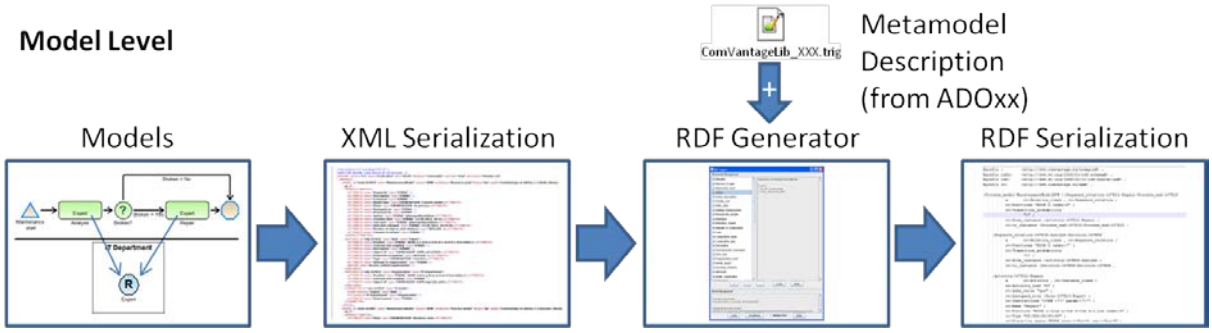
RDF Export

This document provides a simple manual on how to use the new and improved RDF Export. Some knowledge about RDF and SPARQL is required to properly understand certain parts of this manual.

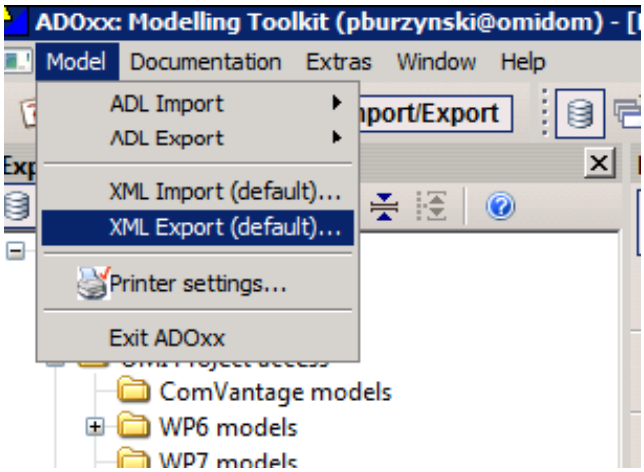
In order to run it a proper java installation is necessary. Java can be obtained at http://www.oracle.com/technetwork/java/javase/downloads/index.html (last accessed 18. Feb. 2013). To run a Java application the JRE is needed (JDK is necessary for development). The implementations have been compiled for Java 1.6 and should also work with newer versions.

Please not that this document has been created on a Computer using the "German" language, therefore certain buttons can have a different label (e.g. "Ja" instead of "Yes" or "Öffnen" instead of "Open" etc.).

1 General Procedure



1. Create the models in the Prototype
2. Export the desired models as XML

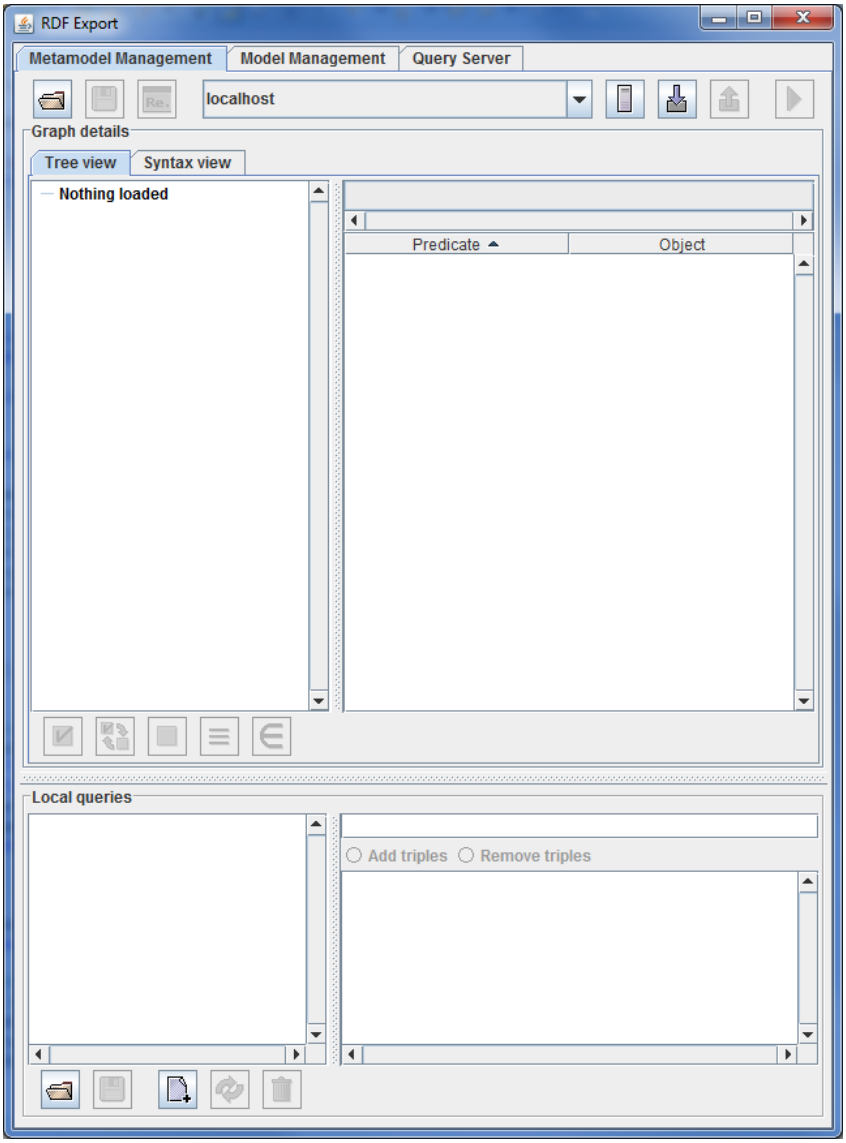


3. Run the .jar file of the RDFExport

IMPORTANT: The "adoxml31.dtd" has to be in the same folder as the "RDFExport.jar"!

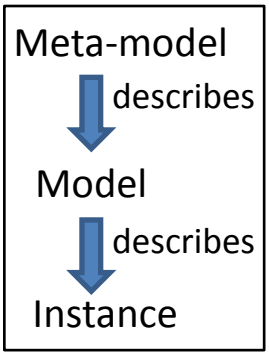
2 Using the RDF Export

When starting the RDF Export you are greeted with the following window:



It tool is separated into three major components which can be accessed through the tabs at the top:

1. **Metamodel Management** – allows loading metamodel descriptions from RDF (TriG, TriX or Turtle) or from an ADOxx 1.3 XML describing the metamodel that is transformed into RDF statements about the metamodel.
2. **Model Management** – allows loading model descriptions from RDF (TriG, TriX or Turtle) or from an ADOxx 1.3 XML describing the model(s) that is transformed into RDF statements about the model. For the transformation data from the **Metamodel Management** component is required (i.e. a metamodel has to be loaded).
3. **Query Server** – allows executing Select queries on a server and shows the result as well as the duration of sending the request and receiving the response.



The recommended procedure is to first use the **Metamodel Management** to load data/information about the metamodel and then use the **Model Management** to transform the model data into RDF.

2.1 General User Interface

The user interface has been designed to use icons visualizing the functionality that hides behind them as intuitively as possible. Additionally tool-tips are shown when moving the mouse over a button. The major functionalities for each of the three components can be found close to the top of the window. The components are also composed out of smaller parts/panels (e.g. **Tree view** in the **Metamodel Management** component) which in turn can have their own buttons. Some of those panels are reused throughout different components (e.g. **Tree view** is used in both **Metamodel Management** and **Model Management**). Their component independent features are described in subsection of this section, while the component specific features are covered in the sections corresponding to those components. Furthermore dialogs are used to communicate with the user and inform about successes and errors.

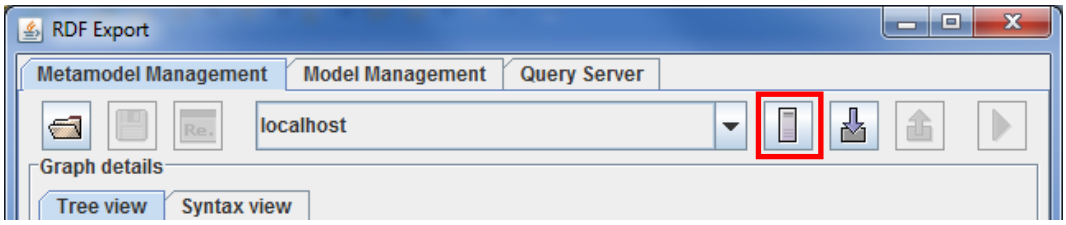
2.1.1 Icons

The following table describes what the used icons visualize:

	Opens or Loads from a file.		Saves to a file.
	Downloads from a server.		Uploads to a server.
	Opens a server configuration dialog.		Executes available/selected queries.
	Adds an item (server, query etc.)		Removes the selected item(s).
	Updates the data about the item when it has been changed.		Closes the window or dialog.
	Mark the selected items.		Unmark the selected items.
	Switch the mark of the selected items.		
Metamodel management			
	Rename the loaded metamodel.		
	Set the selected item to represent equivalence (owl:sameAs).		Set the selected item to represent membership (rdf:type).
Model Management			
	Indicates to save as several separate files.		Indicates if XML transformation should strictly adhere to metamodel.

2.1.2 Server management

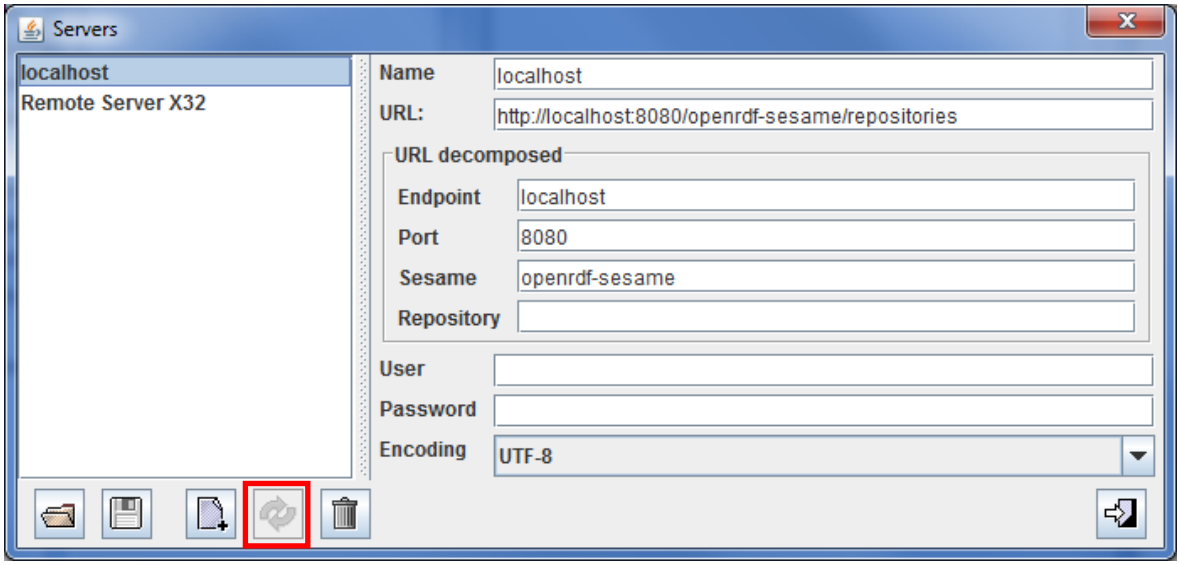
Servers can be used throughout all components. However, to properly work the available servers first have to be configured. This is achieved by opening the dialog through the “Server” button.



This button is found in all components in the top row. The drop-down box to the left of the button shows which server is currently selected and also allows changing the selected server. Note that all components use the same server configurations, and change the server in one changes the server in all of them. Pressing the button opens the dialog shown below.

The icons at its bottom (from left to right) allow:

- Adding servers stored in a file (.ser)
- Saving all servers in a file (.ser)
- Adding a new server
- Updating the information of the currently selected server (highlighted with red rectangle)
- Removing the currently selected server
- Closing the dialog



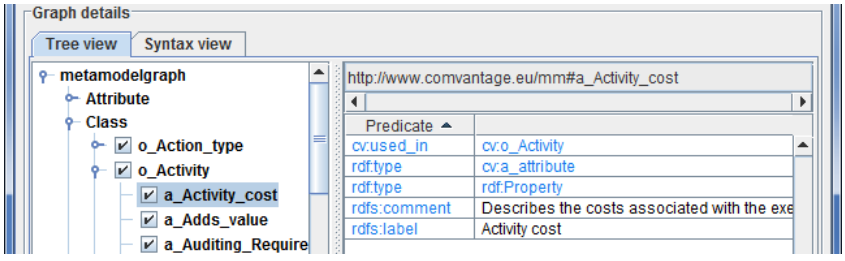
The dialog itself is split into two parts: the left part contains all of the known servers, while the right part shows the details of the currently selected server. The right part can also be used to modify the details of the selected server. Note that changes are only applied when the "Update" button (highlighted in image) is pressed. When the application is started it also checks two places for files containing server descriptions:

- The user's home directory – it will add servers from all ".ser" files from there.
- The applications directory – it will add all servers from a "autoload.ser" file. This only happens when nothing was loaded from the user's home directory.
- If no servers were loaded then a default "localhost" sesame server is added.

Note that for security reasons passwords are not stored in the files and are also not displayed when a server is selected. Also currently only Sesame servers (tested with version 2.7.9) are supported.

2.1.3 Tree view

Some components contain a **Tree view** to describe one or several graphs by arranging their contents as a tree. This assumes a certain semantic behind a graph and that it makes use of specific statements (e.g. **Metamodel Management** component assumes that attributes can be identified by being of rdf:type cv:a_attribute). Therefore the specific structure of the tree depends on the component. The image below shows an example of the **Tree view** from the **Metamodel Management** component.

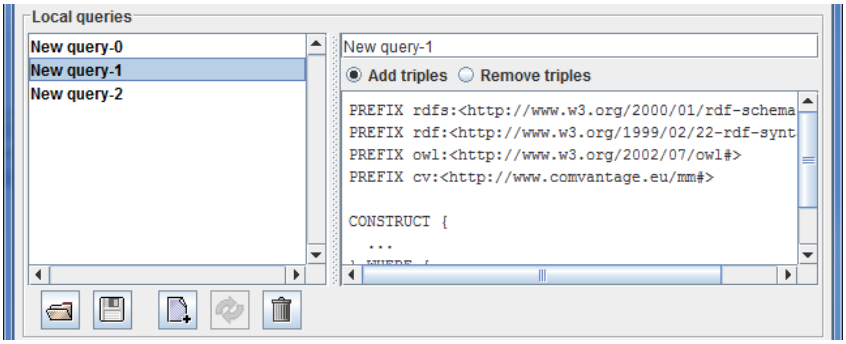


In general the elements represent non-literal rdf:Resources (with possible exceptions). When such an element is selected in the tree then the right side will show details about that element[1]. The text field at the top shows the complete URI of the element and the table below will show all known statements (in a certain context, typically the graph) in which the element is used.

The table contains one column for the predicate and one for the object of the statement, while the element selected from the list is always considered the subject. Colours are used to make the entries easier to identify, with **black** entries representing literals, **blue** entries representing non-literal resources and **green** for inverse properties, which additionally start with the '^' character. Clicking on a row in the table will select the element in the tree (that is closest to the root) representing the object (from the "Object" column) from that row. This allows navigating through the tree in a similar fashion to using hyperlinks. It is also possible to sort the entries in the table by clicking on the heading of the table.

2.1.4 Queries - Local queries and Remote queries

Some components contain a query part (e.g. **Local queries**) part which allows loading, creating and adapting queries that are used in the component. In general those queries are used to manipulate the graph. The left side of the part contains the list of all loaded queries, while the right side describes the selected query[1]. The order of the queries in the list can be manipulated through drag-and-drop. The details of the query always contain its name and the query code. Additionally, local queries which are used for graph manipulation can either be set to add triples or remove triples. The image below shows a **Local queries** part with three queries loaded and one of them selected.



Not far from the query part are usually five buttons located that allow (in the above image from left to right):

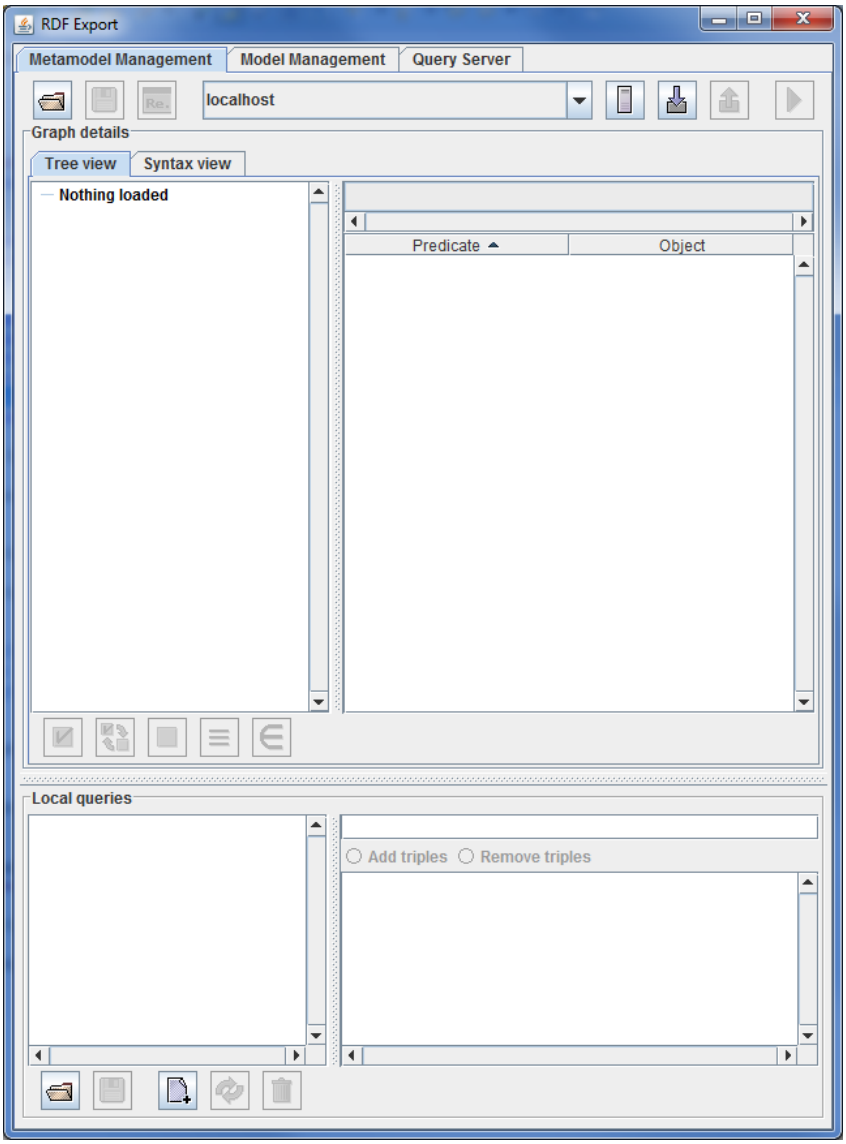
- Loading queries from files (filename used as query name)
- Saving selected queries to files (select folder where to save them)
- Adding a new query
- Updating the currently selected query[1] (when changes have been made)
- Removing the selected queries

It is often possible to select a folder when loading queries, which will then search recursively the folder and all of its children for queries to load and then add them in alphabetical order. Note that the value of a query does not change unless the update button has been clicked. The button becomes available if some part of the query is changed. Therefore changes can be cancelled by simply selecting a different query in the list.

[1] If more than one element is selected then only the details of the first element (i.e. closest to the top) are shown.

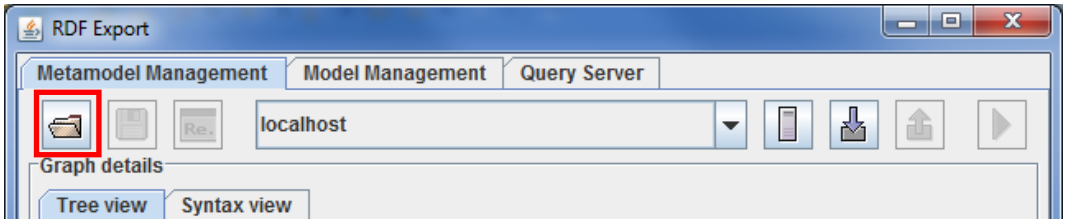
2.2 Metamodel Management

The **Metamodel Management** component allows transforming metamodel descriptions from XML to RDF, retrieving metamodel RDF descriptions, manipulating them to a certain degree and storing the descriptions.



2.2.1 Loading from a file

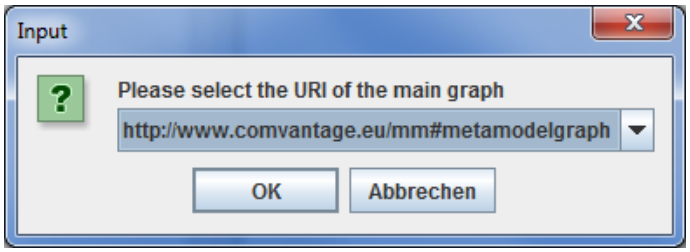
The description of the metamodel can be loaded in the **Metamodel Management** component by using the "Open" button.



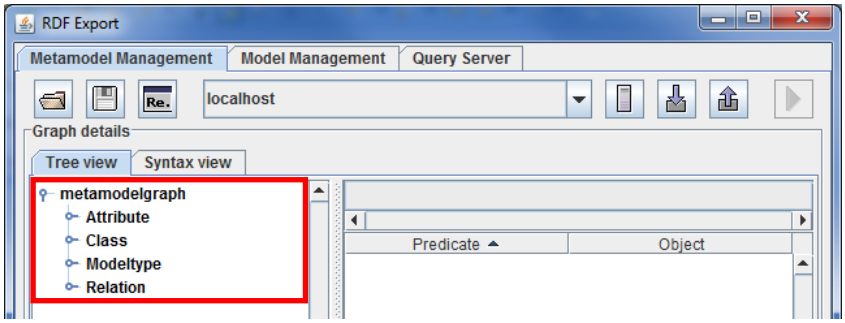
This in turn will open up a file selection window. Here select from which file to load the descriptions. Different types are supported like TriG and XML among others. A TriG file (ComVantageLib_XXX.trig) containing the metamodel description for one of the ComVantage prototypes has been provided together with the RDF Export. Should an XML file be selected then the user is asked to specify a graph name and the metamodel described in the XML it will be transformed into RDF first. If the XML

file does not follow the required structure (i.e. the structure of a modelling method exported from ADOxx 1.3) an error will be shown[2].

If the selected source file contains more than one graph then a dialog will ask which of those graphs should be used for the metamodel description (as in the image below).[3]

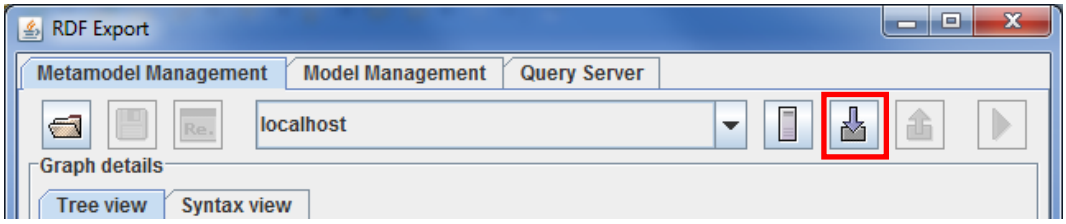


If a graph has been loaded then a success dialog appears and the **Metamodel Management** component should change to something resembling the following image (the red rectangle highlights the change):



2.2.2 Downloading from a server

It is also possible to download a graph describing the metamodel from a server. This requires at least one properly configured server (see section 2.1.2). The server to use for the download has to be selected from the drop-down list at the top, which contains all of the configured servers. After the right server has been selected press the "Download" button to retrieve the data from the server.



If the selected server stores more than one graph then a dialog will ask which of those graphs should be used for the metamodel description.

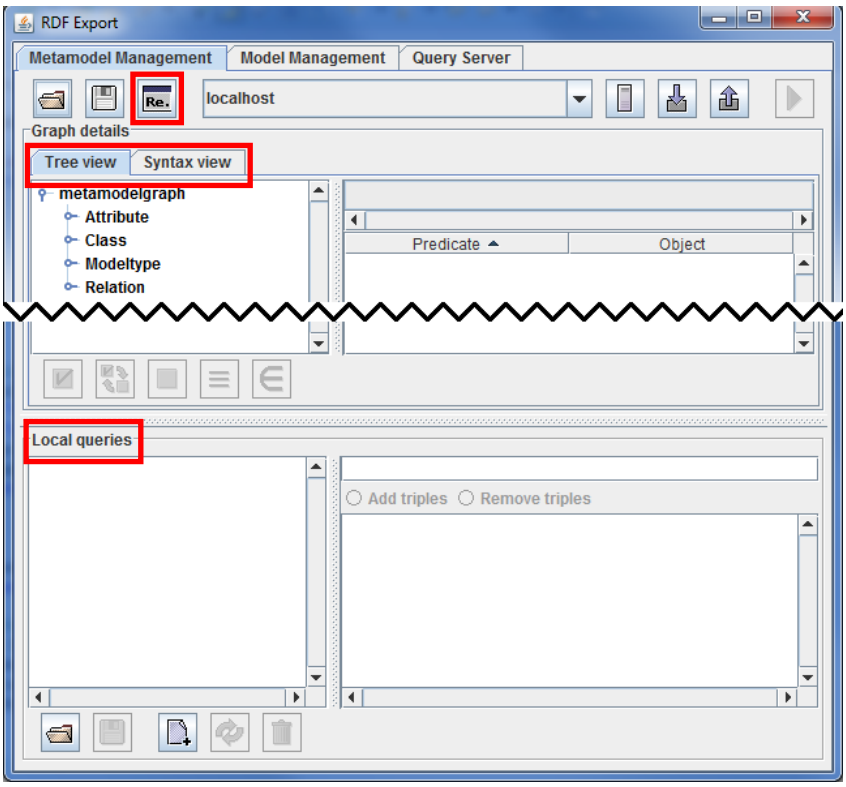
2.2.3 Managing the metamodel

Some limited functionality to manage the metamodel is available in the application. The **Metamodel Management** component uses different parts for this task:

- A "Rename" button to rename the currently loaded metamodel and change the default namespace. (at the top next to the "Save" button)
- A **Tree view** that shows the attributes, classes, modeltypes and relations described in the metamodel. (accessible through the tab in the upper part)

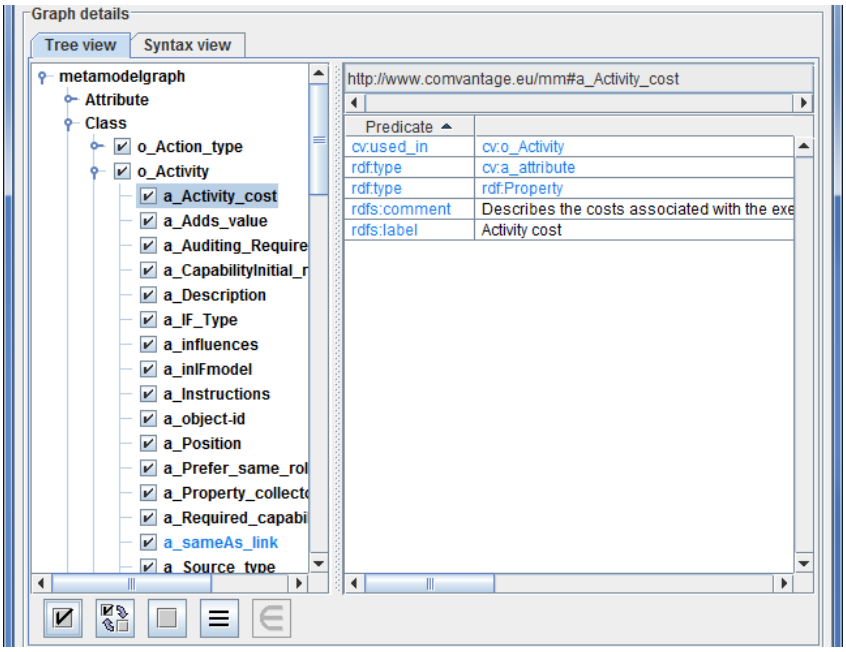
[2] In this case the error usually complains with a "XML Parse Exception: Missing <classes> element".
[3] Other windows might appear depending on the selected file type, but they should be self explanatory.

- A **Syntax view** that shows the current metamodel description in TriG format. (accessible through the tab in the upper part).
- A **Local queries** panel that shows loaded queries. (always present at the bottom)



The “Rename” button should be used to change the namespace and name of the graph containing the description of the metamodel. A dialog will ask to enter a new URL for the graph and everything before the first ‘#’ character will be used as the namespace and everything after that as the name of the graph.

The **Tree view** shows some of the information of the metamodel description that was properly interpreted (see section 2.1.3). Elements are grouped into attributes, classes, modeltypes and relations and can be found as children of those in the tree. If certain elements (like classes or relations) make use of attributes, then those attributes will also be shown as children of those. Note that even though the same attribute might be found in several places of the tree it is still the same attribute and changing one changes them all. The image below shows how the **Tree view** could look when the attribute “a_Activity_cost” is selected.

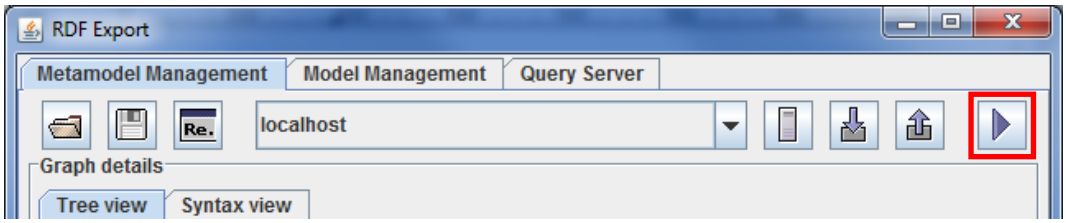


The buttons at the bottom of the **Tree view** can be used to manipulate the data of the graph. Certain elements (classes, attributes etc.) can either be selected or not using the first three buttons seen in the image above. Elements that are not selected will have certain statements about them removed (e.g. for an attribute it removes the statement about its rdf:type being cv:a_attribute) when the graph is saved or requested by another component (e.g. **Model Management** component). This is useful to remove implementation specific or "debug" attributes that are not necessary in the RDF serialization. Additionally, if an attribute is selected, the fourth button (read as "equivalence") allows changing if the attribute should be used to determine the URI for objects[4]. This is visualized by the attributes text changing to the colour **blue**. The firth button (read as "membership") is available when a relation is selected and allows changing if the relation should be used to indicate membership to a type[5]. This is visualized by the relations text changing to the colour **green**. Note that all of those changes are only applied when the graph is saved, uploaded or retrieved by another component.

The **Syntax view** simply shows the information of the metamodel description as RDF code using TriG syntax. While it cannot be edited, it is possible to select the text (or parts of it) and copy & paste it to other tools if so desired. Alternatively it could be saved in a file with the desired format.

The **Local queries** panel allows to specify queries that can then be executed on the graph locally (see section 2.1.4). The **Metamodel Management** component only supports "Construct" queries, which generate triples that are either added or removed. Please note that the supported query concepts are restricted by the library used in the implementation and therefore not all queries that are valid SPARQL can necessarily be executed in this application. It is not recommended to use queries to add statements in the form of "?x rdf:subPropertyOf owl:sameAs" or "?x rdf:subPropertyOf rdf:type", since those will be overwritten by the adaptations from the **Tree view** (i.e. if an attribute is not set as "equality" in the **Tree view**, visualized by a blue font-colour, then any "rdf:subPropertyOf owl:sameAs" statement attached to that attribute will be removed when the graph is used)**.** The queries are executed by pressing the "Run queries" button at the top.

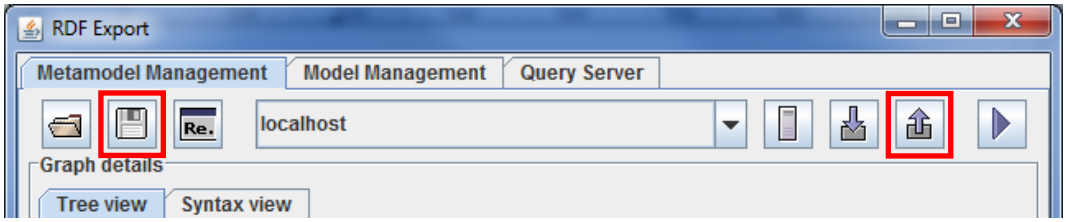
[4] This adds a statement about the attribute being rdfs:subPropertyOf owl:sameAs.
[5] This adds a statement about the relation being rdfs:subProeprtyOf rdf:type.



This executes all of the queries in the **Local queries** panel in the order they appear in the list, with the ones closer to the top being processed before queries closer to the bottom.

2.2.4 Saving to a file or uploading to a server

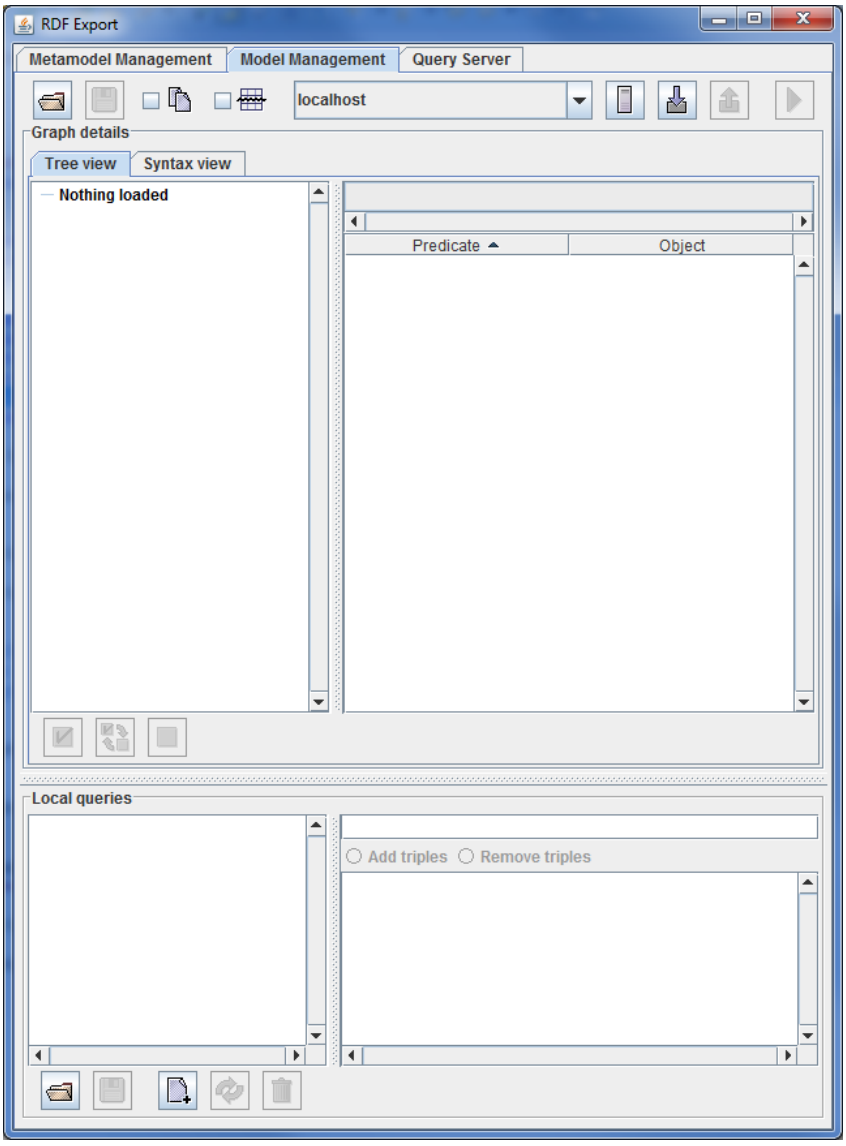
The RDF description of the metamodel can also be saved to a file or uploaded to a server. Use either the “Save” button or the “Upload” button to access those functionalities.



Both are very similar to their “Load”/”Download” counterparts. While saving opens a file selection dialog, uploading uses the server selected in the drop-down list. Both apply the changes specified in the Tree view before saving/uploading. Uploading currently uses the HTTP-PUT method, which means that any previous graphs with the same name are replaced.

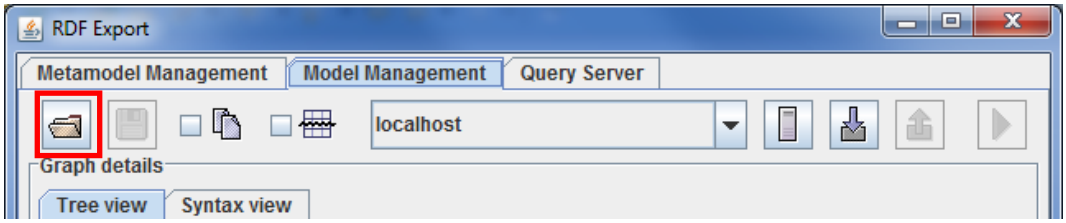
2.3 Model Management

The **Model Management** component allows transforming model descriptions from XML to RDF in accordance to a specific metamodel description, retrieving model RDF descriptions, manipulating them to a certain degree and storing the descriptions. It looks and works (for the most part) very similar to the **Metamodel Management** panel.



2.3.1 Loading from a file

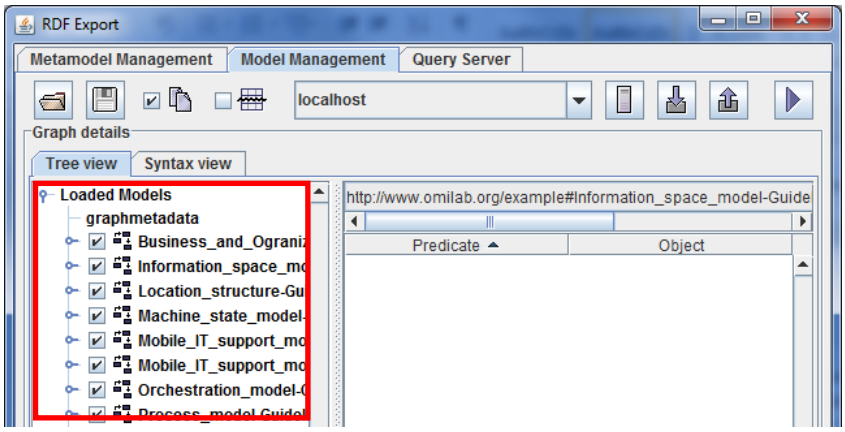
The description of one or several models can be loaded in the **Model Management** component by using the “Open” button.



This in turn will open up a file selection window. Here select from which file to load the descriptions. Different types are supported like TriG and XML among others. An example TriG and XML file (RDFExport_Example_XXX.*) containing several public models using the ComVantage metamodel has been provided together with the RDF Export. If an XML file is selected then the user is asked to

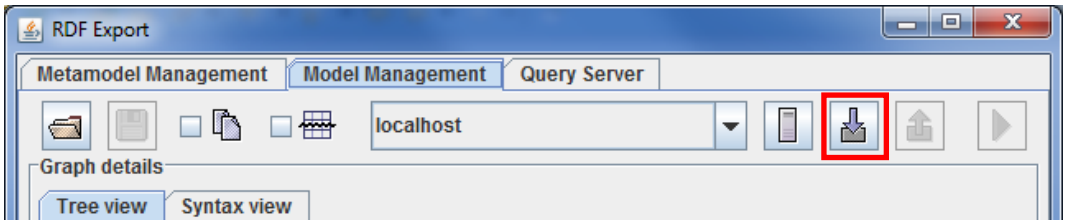
specify a namespace to be used and the models described in the XML it will be transformed into RDF. This transformation however requires that a metamodel description is loaded in the **Metamodel Management** component first. The transformation of the model is then based on the loaded metamodel description. If the XML file does not follow the required structure (i.e. the structure of a modelling method exported from ADOxx 1.3) or no metamodel description is loaded an error will be shown. An error also occurs when the DTD for the XML could not be found.

If the graphs have been loaded then a success dialog appears and the Model Management component should change to something resembling the following image (the red rectangle highlights the change):



2.3.2 Downloading from a server

It is also possible to download graphs describing models from a server. This requires at least one properly configured server (see section 2.1.2). The server to use for the download has to be selected from the drop-down list at the top, which contains all of the configured servers. After the right server has been selected press the “Download” button to retrieve the data from the server.

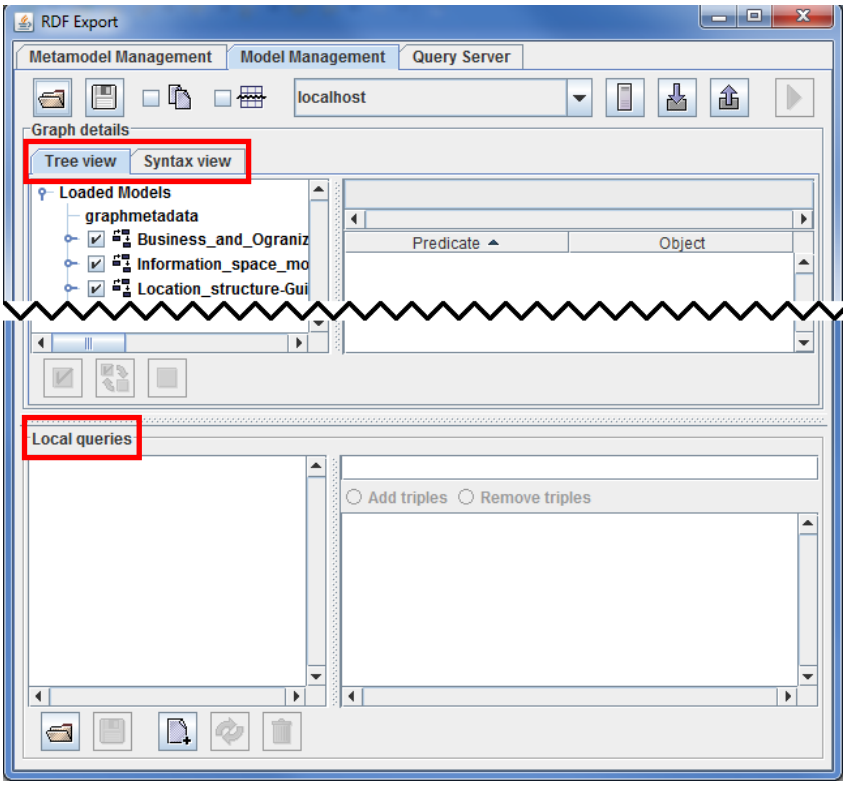


This will download all of the graphs found on the server.

2.3.3 Managing the model

Some limited functionality to manage the metamodel is available in the application. The **Model Management** component uses different parts for this task:

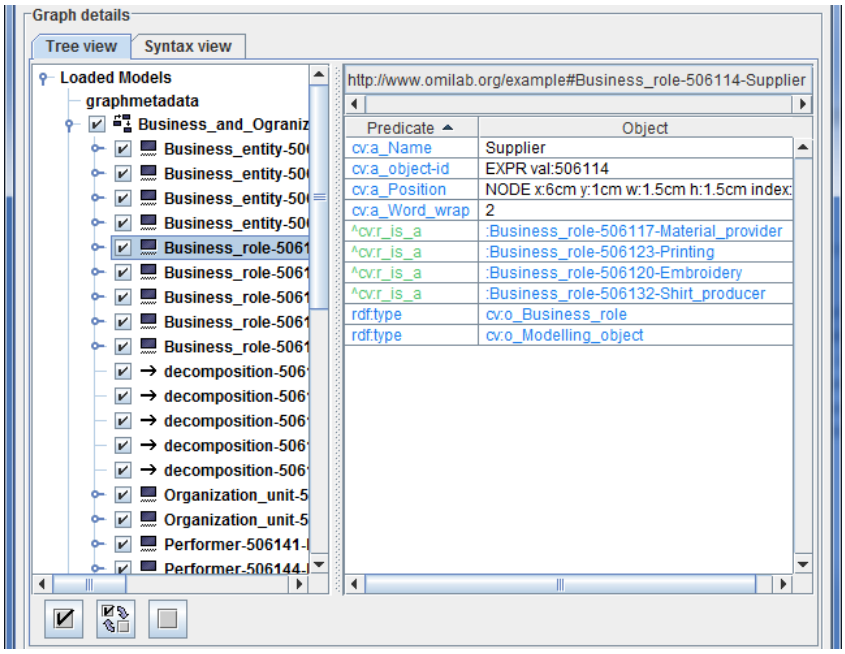
- A **Tree view** that shows the described models and the elements they contain as a hierarchy. (accessible through the tab in the upper part)
- A **Syntax view** that shows the current described models in TriG format. (accessible through the tab in the upper part).
- A **Local queries** panel that shows loaded queries. (always present at the bottom)



The **Tree view** shows some of the information of the description of the models that was properly interpreted (see section 2.1.3). The children of the root element represent all the graphs that have been loaded. If the graph represents a model then it will contain additional children. Those represent either model objects or relations that are part of the model. The model objects can further contain children representing links to other objects (e.g. “Interrefs” or to elements of a table). Different icons are used to represent different types of elements in the tree:

- is used for models.
- is used for model objects.
- is used for relations.
- is used for relations without attributes or links (Interrefs, table attributes etc.)

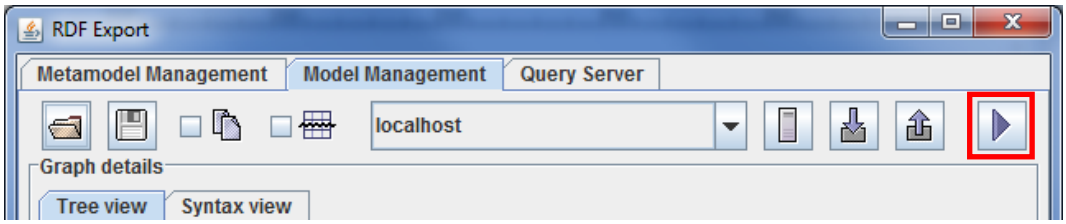
Note that even though the same element might be found in several places of the tree it is still the same element and changing one changes them all. The image below shows how the **Tree view** could look when a modelling object is selected.



The buttons at the bottom of the **Tree view** can be used to manipulate the data. Certain elements (models, model objects etc.) can either be selected or not using the three buttons seen in the image above. Elements that are not selected will have certain statements about them removed when the graph is saved or requested by another component. For models the graph with their URI as well as all statements where their URI is the subject are removed. For any type of model content (model objects, relations etc.) every statement where they are the subject or the object will be removed.

The **Syntax view** simply shows the information of the models described as RDF code using TriG syntax. While it cannot be edited, it is possible to select the text (or parts of it) and copy & paste it to other tools if so desired. Alternatively it could be saved in a file with the desired format.

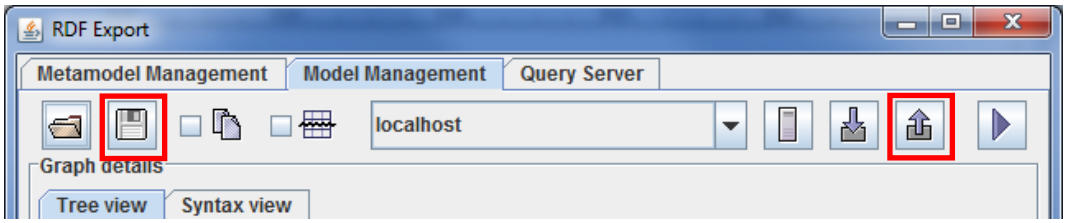
The **Local queries** panel allows to specify queries that can then be executed on the graph locally (see section 2.1.4). The **Model Management** component only supports "Construct" queries, which generate triples that are either added or removed. Please note that the supported query concepts are restricted by the library used in the implementation and therefore not all queries that are valid SPARQL can necessarily be executed in this application. The queries are executed by pressing the "Run queries" button at the top.



This executes all of the queries in the **Local queries** panel in the order they appear in the list, with the ones closer to the top being processed before queries closer to the bottom. Also the queries are executed for each graph separately.

2.3.4 Saving to a file or uploading to a server

The RDF description of the models can also be saved to a file or uploaded to a server. Use either the "Save" button or the "Upload" button to access those functionalities.

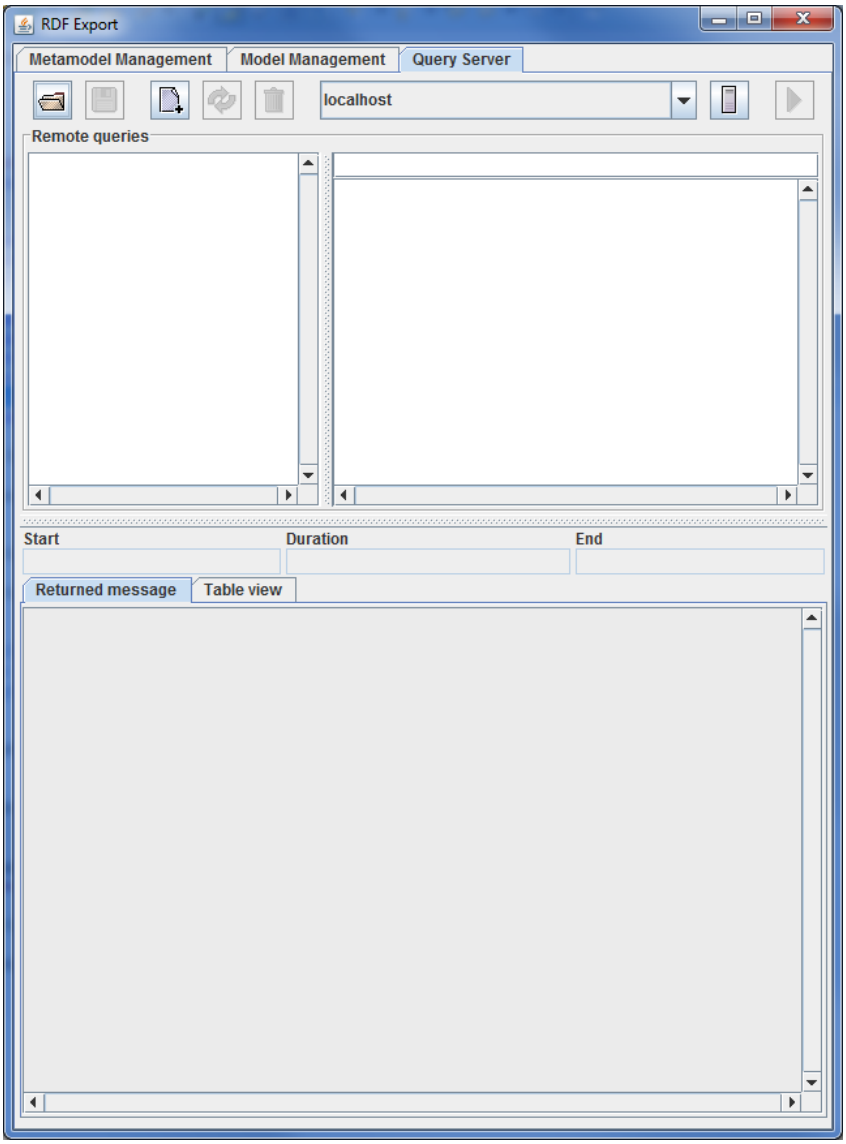


The upload is very similar to the download and requires that a server is selected in the drop-down list. Both the upload and the saving apply the changes specified in the Tree view before saving/uploading. Uploading currently uses the HTTP-PUT method, which means that any previous graphs with the same name are replaced. The description can either be saved as a file using RDF or transformed into ADOxx 1.3 XML by selecting “ADOxx XML files” as the file type. This transformation however requires that a metamodel description is loaded in the **Metamodel Management** component first since it is based on statements about the metamodel. If those are not present then the transformed models can contain less data then the original model. If “Turtle files” is selected then each graph is saved in a separate file, since turtle does not support multiple graphs in one file (for details on how files are split continue reading the next paragraph).

There are also additional options that can be set for saving as a file using the two checkboxes to the right of the “Save” button. When the left checkbox is selected then each graph will be stored in a separate file. In this case the name of each saved file consists of a prefix (which is the part of the filename specified in the dialog) and the graphs name. When the right checkbox is selected then the transformation to XML checks if an attribute is used in a specific class (by querying the metamodel description) before creating it. If it is not checked then the implementation only checks if the attribute is present in the metamodel description at all. Having more attribute elements in the ADOxx 1.3 XML does not prevent it from being properly imported, since incorrect attributes are ignored for the most part (a warning at the end might appear).

2.4 Query Server

The **Query Server** component allows executing SPARQL queries on a remote SPARQL endpoint and shows the result and time it took to execute the query. Currently only queries that return a result of type “application/sparql-results+xml” are supported (i.e. select, but not construct).



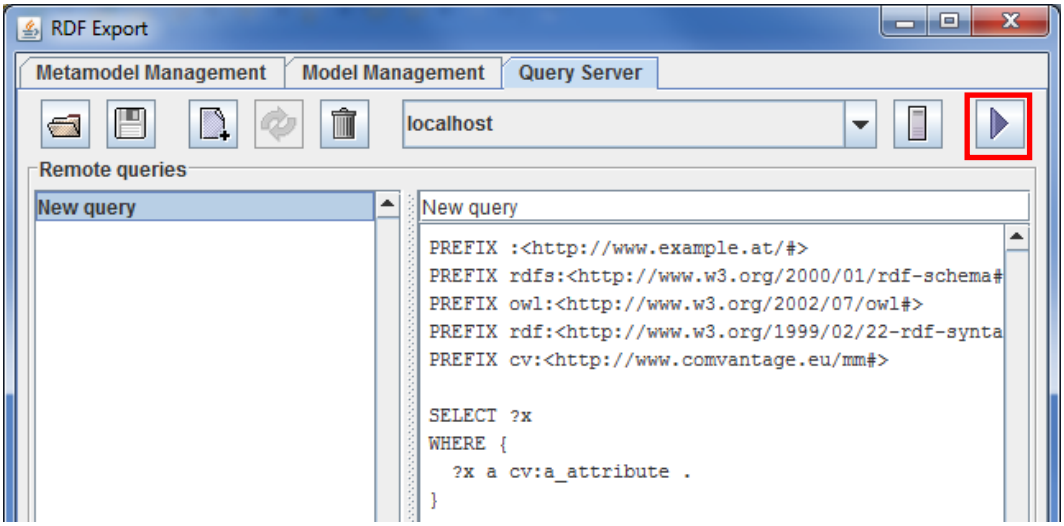
2.4.1 Loading, managing and saving queries

The loading, managing and saving of queries is already described for the most part in section 2.1.4. The panel for those tasks is found at the top “half” of the application. One difference is the buttons which are located above the list instead of below. Also when a query is added using the “Add” button the component tries to contact the currently used server and retrieves all of its prefixes, which are automatically added to the query. If no prefixes are found on the server then a set of predefined prefixes is used.

2.4.2 Executing queries

In order to execute queries at least one properly configured server (see section 2.1.2) is necessary. The server to use for the query has to be selected from the drop-down list at the top, which contains all of the configured servers. Also only one query can be executed at a time, which is simply selected from the list. When a query is selected simply press the “Run query” button. Note that the content of the text area on the right is sent as the query and not the value stored in the query object. The

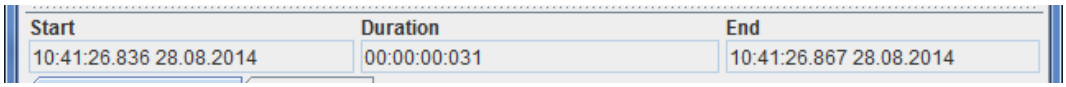
following image shows a simple example query and highlights (red rectangle) the “Run query” button.



Executing the query results one or several dialogs denoting the success and/or any errors. The results received from the server are then shown in the bottom “half” of the application. The result contains three different panels with information:

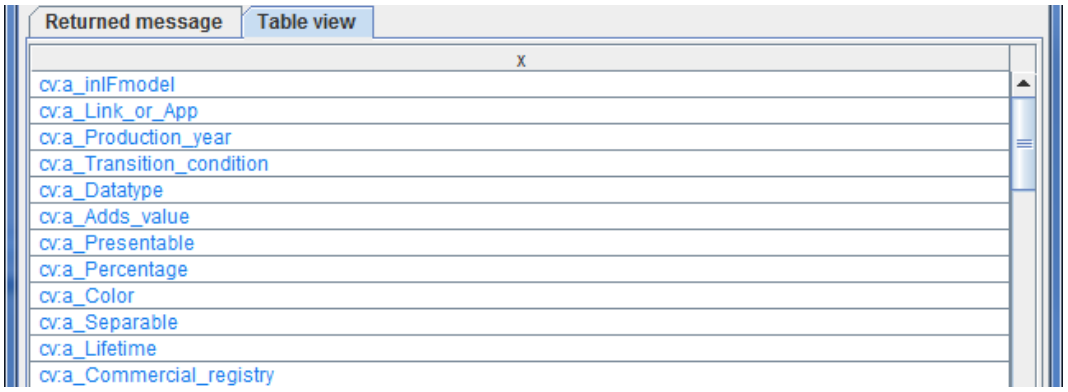
- The **Time panel** providing information about how long the execution of the query took. (always visible)
- The **Returned message** panel which contains the message/content returned from the server. (accessible through a tab)
- The **Table view** which shows the returned message/content as a table. (accessible through a tab)

The **Time panel** provides three pieces of information: when sending the query has started, when the result has been received and the difference between those points in time. Times are presented in the format of “Hours : Minutes : Seconds : Milliseconds”, while the date uses the format of “Day : Month : Year”. Note that the results also contain the time it takes to create the message, send it to the server and receive the result. The following image shows one example of the **Time panel** for the query used above.



The **Returned message** panel shows the message that has been returned by the server. Since only “application/sparql-results+xml” types of result messages are currently supported they should always follow that structure. The following image shows an example for the query used above.

The **Table view** parses the returned message and shows the result in the form of a table. The columns represent the different variables from the returned message while each row represents a single result. Similar to the **Tree view** (see section 2.1.3) colours are used to make the entries easier to identify, with **black** entries representing literals, **blue** entries representing non-literal resources. The following image shows an example for the query used above.



2.5 Procedure for serialization of models as Linked Data

This application was mainly created to allow the serialization of models as Linked Data and from there to further link them to other data. This is achieved by transforming models from a propriety format into RDF descriptions and then serializing those in a desired RDF serialization format. This implementation focuses on transforming models from the ADOxx 1.3 XML serialization format. If the goal is simply to serialize the models as Linked Data, without the desire of managing any metamodel descriptions or further changing the model descriptions in the application, then the procedure is as follows:

1. Load the corresponding metamodel description in the **Metamodel Management** component. In the simplest case this involves simply loading a .trig file (otherwise see section 2.2.1).
2. Load the XML in the **Model Management** component. This can simply be achieved by using the XML export of the ADOxx 1.3 tool (more details see section 2.3.1).
3. Save the data from the **Model Management** component in the desired format or directly upload to a server (more details see section 2.3.4).

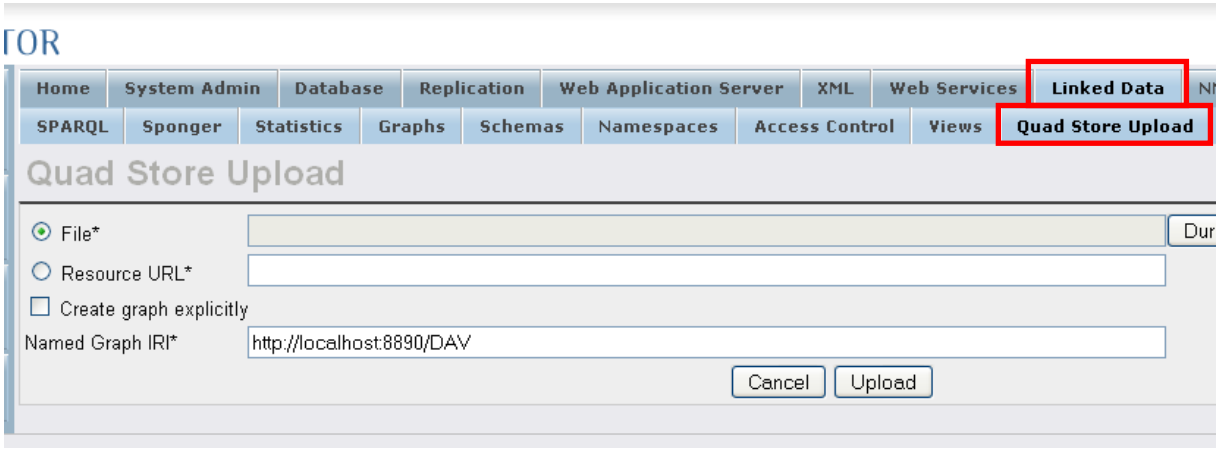
3 Use with a Quad Store

The generated results can also be published to a Quad Store like Virtuoso or Sesame.

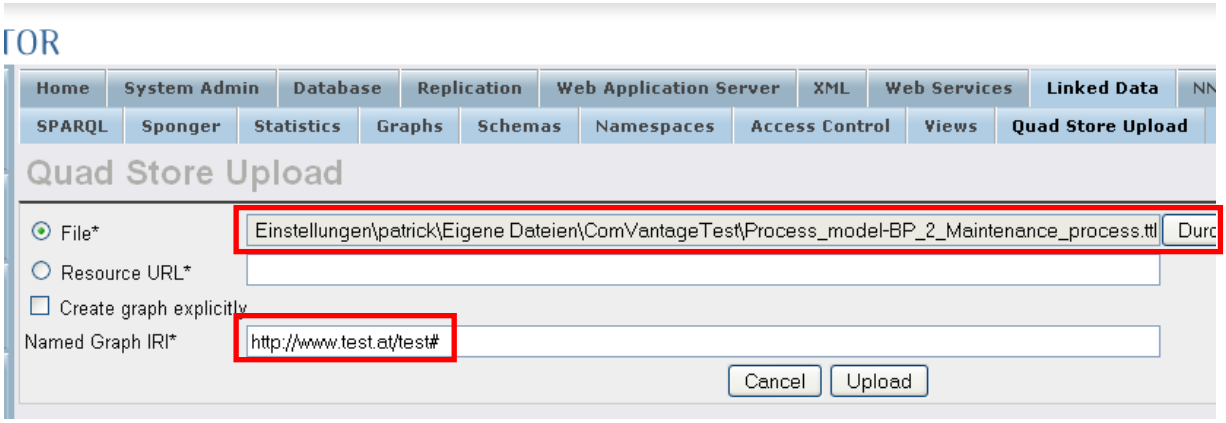
3.1 Publishing to Virtuoso

In order to publish the data in Virtuoso you have to save the files in Turtle (.ttl) format. Putting a tick next to “Multiple files” allows you to select a folder to save to instead of a file, which should make things easier. This generates one file for each graph.

Now start the Virtuoso Conductor, log in and go to the “Quad Store Upload” found in the “Linked Data” part.



Once there select one Turtle file containing a graph and specify the URI for the graph under “Named Graph IRI”. Then click on “Upload” to upload the graph.



Now repeat the previous step (select file, provide URI and click Upload) for every graph you want to publish.

Once finished, you can query the data from the SPARQL component of the Virtuoso Conductor.

3.2 Publishing to Sesame

The application provides a “Download” and “Upload” button that can be used with Sesame servers.

4 Finer details of the RDF Export

This section addresses some of the finer details of how the RDF Export works and can help in finding solutions to (more or less) exotic errors or provide some inspiration for enhancing the connectivity of the modelling method to Linked Data. It is easy to feel lost in this section, especially without knowledge about modelling, metamodelling, ADOxx, the ADOxx XML serialization, RDF, the RDF TriG syntax and some basic programming.

4.1 Transformation and structure of (meta-)model descriptions

The transformation of metamodel descriptions is used to translate the description of a metamodel from ADOxx 1.3 XML to RDF. This also means that the structure is changed to one closer resembling the metamodel description using RDF from ComVantage deliverable D3.1.2[6] (section 3.3.10 of the deliverable). However, this implementation still deviates a bit from the specification of the deliverable to strike a balance between usability and complexity of implementation and therefore omits some of the more complicated cases. Also additional concepts are added, because of the difference between the two formats.

4.1.1 Metamodel descriptions

The XML structure expected for the transformation looks as follows (note that in most cases the order is irrelevant):

- library
  - attributes
    - attribute (with name “Modi”)
      - value (containing the model type descriptions in LEO-XML[7] format)
  - classes
    - class (with attribute “name”) [optional]
      - attributes [optional]
        - xsd:any (with attribute “name” and [optional] attribute “type”)
          - facets
            - facet [optional]
    - relationclass (with attribute “name”) [optional]
      - attributes [optional]
        - xsd:any (with attribute “name” and [optional] attribute “type”)
          - facets
            - facet [optional]
    - recordclass (with attribute “name”) [optional]
      - attributes [optional]
        - xsd:any (with attribute “name” and [optional] attribute “type”)
          - facets
            - facet [optional]

Note that in the current implementation “Attribute profiles” are completely ignored. Additionally all the attributes used by model types have to be in a class with the name “__ModelTypeMetaData__”. Also it does not specifically check for the element type of the children of the “attributes” element for

[6] It can be accessed through the OMILab Repository in the Design phase. The documents name is “ComVantage Core Modelling Method Specification”

[7] The proprietary ADOxx format serialized as XML.

"class", "relationclass" and "recordclass". The implementation also does not specifically cover most of the different cases to handle tables described in D3.1.2. Instead it considers tables to be ordered lists of objects (rdf:List).

The above described structure is used to access information about the metamodel and based on this create the RDF description[8]:

1. First some standard meta²-model statements are added (cv:Model is a rdfs:Class, cv:Modelling_object is a rdf:Class etc.)
2. Then the attribute with the name "Modi" is parsed for all the used model types and for each *Model type*:
    a. Statements are added that denote: *Model type* rdf:type cv:m_Model
    b. When the model type has custom attributes
        i. Find the notebook definition and use it to process the attributes
    c. Names of all the classes used in the model type are stored in a set *Used classes*
3. For every "class" element in "classes" as *Class*
    a. If it is in *Used classes*
        i. Statements are added that denote: *Class* rdf:type cv:o_Modelling_object
        ii. All of its attributes (all children of the "attributes" element) are processed
4. For every "relationclass" element in "classes" as *Relation class*
    a. If it has attributes besides the default attributes
        i. Statements are added that denote: *Relation class* rdf:type cv:r_Modelling_relation_a
        ii. All of its attributes (all children of the "attributes" element) are processed
    b. Otherwise
        i. Statements are added that denote: *Relation class* rdf:type cv:r_modelling_relation_na
        ii. Also statements are added that it is treated as a connector in ADOxx: *Relation class* rdf:type cv:CONNECTOR (necessary for transformation of models back into XML)
5. For every "recordclass" element in "classes" as *Record class*
    a. If it is in *Used classes*
        i. Statements are added that denote: *Record class* rdf:type cv:o_Table_object
        ii. All of its attributes (all children of the "attributes" element) are processed
6. The attributes "Name" and "Version" are added to the RDF description (since they are implied by the metamodel description):
    a. Stating that cv:a_Name rdf:type cv:a_attribute
    b. And cv:a_Name rdfs:subPropertyOf rdfs:label
    c. As well as cv:a_Version rdf:type cv:a_attribute

The procedure for processing attributes is (*Parent* is used to denote the element the attribute belongs to):

1. If the type is "interref"
    a. Statements are added that denote: *Attribute* rdf:type cv:r_modelling_relation_na
    b. Also that: *Attribute* cv:used_in *Parent*
2. Otherwise if the type is "record"

[8] Variables are written in italic.

    a. Statements are added that denote: Attribute rdf:type cv:a_table_attribute
    b. Also that: *Attribute* cv:used_in *Parent*
3. Otherwise if the name is “HlpTxt”
    a. Adds a rdfs:comment to *Parent*
4. Otherwise
    a. Statements are added that denote: *Attribute* rdf:type cv:a_attribute
    b. Also that *Attribute* cv:used_in *Parent*

The above descriptions do not specify all of the statements that are added and focuses on the important ones. Statements that add labels, comments or reference rdf/rdfs concepts are mostly omitted. Also it extends the names used for the URI with additional prefixes to separate into four categories:

- “m_” for URIs that deal with model types
- “o_” for URIs that deal with modelling objects
- “r_” for URIs that deal with relations
- “a_” for URIs that deal with attributes

This approach is chosen since there can be cases where for example a class, a model type and an attribute have the same name. Therefore to still be able to separate them their names are extended with those prefixes for the URIs.

Also the description of the metamodel should be extended with the following statements if applicable:

- Indicate that an attribute denotes the URI of an object
    - This is achieved by stating that the attribute is a rdfs:subPropertyOf owl:sameAs. The application provides an “equivalence” button in the **Tree view** to state this.
- Indicate that a relation without attributes denotes additional types of an object
    - This is achieved by stating that the relation is a rdfs:subPropertyOf rdf:type. The application provides a “membership” button in the **Tree view** to state this.
- Indicate that a table contains triples (“Direct transformation into triples” from the different table cases from D3.1.2)
    - This is achieved by stating that the corresponding table attribute (rdf:type cv:a_table_attribute) is also of the type cv:direct_transformation (rdf:type cv:direct_transformation)
    - Additionally state for one of the tables attributes that it is a sub-property of (rdfs:subPropertyOf) rdf:subject, for a different column that it is a sub-proeprty of rdf:predicate and for another column that it is a sub-proeprty of rdf:object. Note that the current implementation of the Model transformation substitutes the parent object of the table for each of the three columns that is missing (e.g. if no column is specified to represent rdf:subject, then the object the table is in will be used as the subject of the statement). Also the Model transformation currently expects to find simple attribute values in those columns (not e.g. Interrefs) .

4.1.2 Model descriptions

The XML structure expected for the transformation is specified by the “adoxml31.dtd”. However, the current implementation does not consider “Attribute profiles” in any way. The procedure for creating the RDF description of models relies on having access to the metamodel description in RDF (simply

called *metamodel* here) and on the use of labels in the *metamodel* to bridge with the names of "name" attributes of elements from the XML. The procedure consist of the following steps:

1. Names of attributes that denote the URI are retrieved from *metamodel* and stored in a set *Sameas*.
2. For ever MODEL element
    a. Create a graph for the model (*model graph*)
    b. Add type, name and version statements about *model graph* in :graphmetadata
    c. Statements about the library-type of the model graph are added in :graphmetadata (necessary for transformation back into XML)
    d. Process all MODELATTRIBUTE elements of the model (if available)
    e. Process all RECORD elements of the model (if available)
    f. For every INSTANCE element of the model (*instance*)
        i. If the *instance* type is not part of the *metamodel* then skip it
        ii. Create a resource representing the *instance*
        iii. Add name and type statements about *instance* in *model graph*
        iv. Process all ATTRIBUTE elements of *instance*
        v. Process all RECORD elements of *instance*
        vi. Postpone all INTERREF elements of *instance*
    g. For every CONNECTOR element of the model (*relation*)
        i. If the *relation* is not part of the *metamodel* then skip it
        ii. Get the *source* and the *target* for the *relation* in model graph
        iii. If *source*, *target* or *relation* type is missing then skip the relation
        iv. If the *relation* type is a cv:r_modelling_relation_na
            1. Add a simple statement in the form of "*source relation*-type *target*" in *model graph*
        v. Else if the relation type is a cv:r_Modelling_relation_a
            1. Create a new resource representing the *relation*
            2. Add type, from and to statements about *relation* in *model graph*
            3. Process all ATTRIBUTE elements of relation
        vi. Else if the relation name equals "Is inside"
            1. Add a simple statement "*target* cv:contains *source*" in *model graph*
3. For every postponed *interref*
    a. If the *interref* (based on name) is not part of the *metamodel* then skip it
    b. Find the *target* of the *interref* and the graph where the target is located (*target's graph*)
    c. If the *interref* denotes type (is rdfs:subPropertyOf rdf:type)
        i. Add the *target* as a new rdf:type to the *source* of the *interref*
        ii. Add a statement denoting in which graph the *target* is found (cv:described_in, only when the target is not a graph)
    d. Otherwise
        i. Add a statement in the form of "*source interref target*"
        ii. Add a statement denoting in which graph the *target* is found (cv:described_in, only when the *target* is not a graph)
        iii. Add the interref statement also in *target's graph*
        iv. Add a statement in *target's graph* denoting in which graph the source is found (cv:described_in)

The procedure for processing attributes (ATTRIBUTE elements) is (*Parent* is used to denote the element the attribute belongs to):

1. If the *attribute* (based on name) is not part of the *metamodel* or it has no value then skip it
2. If the *attribute* value is a URI then
    a. Create a URI resource for the value as *object*
3. Otherwise
    a. Create a literal resource for the value as *object*
4. Add a statement to *model graph* in the form of "*Parent attribute object*"

The procedure for processing tables (RECORD elements) is (*Parent* is used to denote the element the attribute belongs to):

1. If the *table* name has a resource of type cv:direct_transformation then for every ROW as *row*
    a. For every ATTRIBUTE of row as *column*
        i. If it denotes the *subject*, then set *subject* to a URI based on its value
        ii. Otherwise if it denotes *predicate*, then set *predicate* to a URI based on its value
        iii. Otherwise if it denotes *object*
            1. Try to set *object* to a URI based on its value
            2. If the previous fails, then set *object* to a Literal based on its value
    b. All of *subject*, *predicate* and *object* that have not been set are set to *Parent*
    c. Add a statement to *model graph* in the form of "*subject predicate object*"
2. Else If the *table* name is part of the *metamodel* then[9]
    a. Create an anonymous rdf:List (refereed as *list*)
    b. Add a statement to *model graph* in the form of "*Parent table*-name *list*"
    c. For every ROW as *row*
        i. Create a resource representing the *row*
        ii. If more rows are present
            1. Add the resource to the *list* as rdf:first
            2. Create a new anonymous rdf:List (referred to as *second list*)
            3. Add *second list* to the list as rdf:last
            4. Set *list* to *second list*
        iii. Otherwise (this is the last row)
            1. Add the resource to the list as rdf:last
        iv. Process all ATTRIBUTE elements of *row*
        v. Postpone all INTERREF elements of *row*

Note that in all the cases a resource is created to represent a model, an instance or a relation the application first searches for an attribute denoting the URI using the *Sameas* set. If there is no such attribute available then a URI is created based on the objects type, id and name (and possibly others depending on the case).

[9] The actual code is more complicated due to the way lists in rdf are structured (only consisting of rdf:first and rdf:last), but achieves the same thing.

4.2 Storage of queries

Queries in the application consist of at least a name and the query itself. Additionally there are “Local queries” which extend queries by also adding a type denoting if they should be used to add or remove triples for manipulation. Because of this a query can have up to three members/attributes:

- The query name
- The query value
- The query type

However, to simply the storage and also allow the queries to be used outside of the RDF Export they are stored in simple text files. To prevent the loss of data certain “hacks” are used:

- The name of the file (minus the file extension) denotes the name of the query.
- The content of the file contains (for the most part) the query value.
- The query type is indicated by a comment in the first line:
    - Add-type queries don’t contain the comment
    - Remove-type queries contain the comment “#remove” as the first line of the query

While the name and content of the query are easily handled, the application takes care to hide the query type from the user as best as possible. Therefore the first line comment is not shown for “Local queries” in the **Local query** panel and is instead indicated by the colour of the query name and the radio buttons when a query is selected.