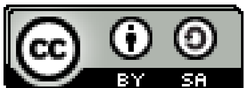
***SASSY***

# SOFTWARE ARCHITECTURE SYSTEM

User Guide for RDFGUI

# Publication History

| Date | Who | What Changes |
|---|---|---|
| 24 October 2019 | | Initial version. |
| 24/09/20 | | Major revision with lots of details. |
| 19/02/21 | | Forms interface. |



Copyright © 2009 - 2021 Brenton Ross
This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.
The software is released under the terms of the GNU General Public License version 3.

# Table of Contents

1 Introduction....5
1.1 Scope....5
1.2 Overview....5
1.3 Audience....5
2 A Short Introduction to RDF....6
2.1 RDF Graphs....6
2.2 The Central Importance of the URI....7
2.3 On the Limits of Universality....8
2.4 Statements and Triples....9
2.5 Literal Data Types....10
2.6 Data Structures....10
2.7 Schema....11
2.8 Reification....11
2.8.1 Queries and Rules....11
2.9 Submodels and Contexts....12
3 Program Components....13
3.1 Menu....13
3.2 Catalogue Tab....14
3.2.1 Opening a Model....14
3.2.2 Adding a Model....15
3.2.3 Submodels....18
3.3 Models Tab....19
3.3.1 Transactions....19
3.3.2 Contexts....20
3.4 Prefixes Tab....21
3.5 Statements Tab....22
3.5.1 Statements Table....22
3.5.2 Statements Dialog....25
3.6 Visual Tab....28
3.7 Schema Tab....29
3.7.1 Class Panel....29
3.7.2 Property Panel....32
3.7.3 Property Attributes Panel....32
3.7.4 Description Panel....34
3.8 Forms....35
4 Tutorials....38
4.1 Schema Examples....38
4.1.1 Classes....38
4.1.2 Object Properties....39
4.1.3 Data Properties....40
4.1.4 Domain and Range....41
4.1.5 Containers....42
4.2 Forms Examples....43

# 1 Introduction

This document is the guide to using the RDF editor program, RDFGUI.

## *1.1 Scope*

The document will cover all aspects of using the program to view and edit RDF data. It will also include some aspects of the RDF specification.

## *1.2 Overview*

The first section provides a brief introduction to RDF. This will allow the reader to understand its benefits and determine if it is suitable for their purposes.

The second section is divided into subsections that describe each major component of the program.

The third section is a set of tutorials that take you through the different ways that the program can be used.

## *1.3 Audience*

Anyone that wants to use RDF data.

# 2 A Short Introduction to RDF

The following is a short introduction to RDF from Cambridge Semantics. I have made a few changes to better align with my view of RDF.

RDF is the foundation of the Semantic Web and what provides its innate flexibility. All data in the Semantic Web is represented in RDF, including schema describing RDF data.

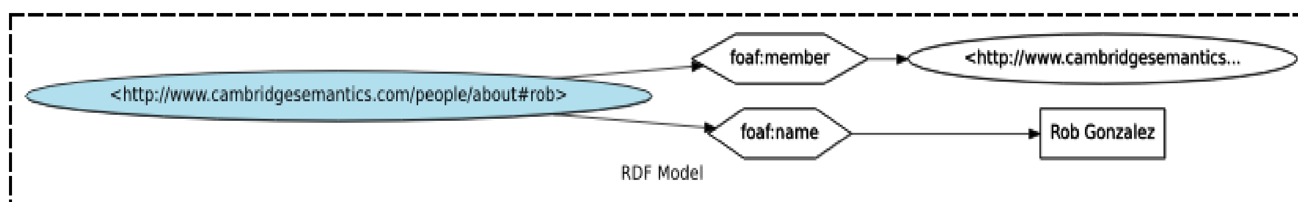
RDF is not like the tabular data model of relational databases. Nor is it like the trees of the XML world. Instead, RDF is a graph.

## 2.1 RDF Graphs

In particular, it’s a labelled, directed graph.

Therefore you can think of RDF as a bunch of nodes (the dots) connected to each other by edges (the lines) where both the nodes and edges have labels.

The term labelled, directed graph will mean a lot to the mathematicians in the audience, but for the rest of you I’ve included a simple example here.




This is a complete, valid, visual representation of a small RDF graph. Show it to any Semantic Web practitioner and it will be immediately obvious to her what it represents.

The nodes of the graph are the ovals and rectangles (ovals and rectangles are a convention that we’ll get to shortly). Technically the edges are labelled arrows that connect nodes to each other, but I find that using hexagons for the labels makes it easier to read. The labels are URIs (this is very important, and we’ll cover it in more detail in a bit).

There are three kinds of nodes in an RDF directed graph:

- Resource nodes. A resource is anything that can have things said about it. It’s easy to think of a resource as a thing vs. a value. In a visual representation, resources are represented by ovals.
- Literal nodes. The term literal is a fancy word for value. In the example above, the resource is http://www.cambridgesemantics.com/people/about#rob (once again, a URI) and the value of the foaf:name property is the string “Rob Gonzalez”. In a visual representation, literals are represented by rectangles.

- Blank nodes. A blank node is a resource without a URI. Blank nodes are an advanced RDF topic. I usually recommend avoiding them in general, especially if you're new to the space. They are listed here simply for completeness.

Edges can go from any resource to any other resource, or to any literal, with the only restriction being that edges can't go from a literal to anything at all.

This means that anything in RDF can be connected to anything else simply by drawing a line.

This idea is key. When we talk about Semantic Web technologies being fundamentally more flexible than other technologies (XML, relational databases, BI cubes, etc.), this is the reason behind it. In the abstract, you're just drawing lines between things. Moreover, creating a new thing is as easy as drawing an oval.

If you compare this mentally to the model you might know from working with a relational database, it's starkly different. Even for basic relationships, such as many-to-many relationships, the abstract model of a relational database gets complicated. You end up adding extra tables and columns (think foreign keys, join tables, etc.) just to work around the inherent rigidity in the system.

The ability to connect anything together, any time you want, is revolutionary. It's like hyperlinking on the Web, but for any data you have!

This linking between things is the fundamental capability of the Semantic Web, and is enabled by the URI.

## 2.2 The Central Importance of the URI

If you want to connect two things in a relational database you have to add foreign keys to tables (or, if you have a many-to-many relationship, create join tables), etc. If you want to link things between databases, you need an ETL job using something like Informatica. It's just not easily done.

If you consider the XML world, the same thing is true. Connecting things within an XML document is possible, if tedious. Connecting things between XML documents requires real work. Unless you're one of the very few who just loves XSLT, you're not doing that very often.

The fundamental value and differentiating capability of the Semantic Web is the ability to connect things.

The URI is what makes this possible.

URI stands for Universal Resource Identifier. The universal part of that is key. Instead of making ad hoc IDs for things within a single database (think primary keys), in the Semantic Web we create universal identities for things that are consistent across databases. This enables us to create linkages between all things (hold the skepticism for a second; we'll get to it!).

In RDF, resources and edges are URIs. Literals are not; they are simple values. Blank nodes are not (this is what the "blank" means in the name). Everything else is, including the edges.

If you look at our example above, there are several examples of URIs.

- http://www.cambridgesemantics.com/
- http://www.cambridgesemantics.com/people/about/rob
- foaf:member (this is shorthand for http://xmlns.com/foaf/0.1/member)
- foaf:name (again, shorthand for http://xmlns.com/foaf/0.1/name)

The first one is the URI for the company Cambridge Semantics. The second is a URI for Rob, the author of this article. The other two are URIs for the edges that connect the resources (we'll say more about URIs of edges in a minute).

You should notice that a couple of the URIs above are URLs. You can click on them. They are valid Web addresses. So what makes them a URI?

In short, all URLs are URIs, but not all URIs are URLs. It's a little confusing, for sure, but the vast majority of Semantic Web practitioners stick to using URLs for all of their URIs.

Back to the concept of universality of identity. If I have a database that contains information about myself, I would use the URI http://www.cambridgesemantics.com/people/about/rob to refer to any data relating to me. If you have another database that has other information about me, you would also use that same URI. That way, if we wanted to find all facts in both databases about me, we could query using the single universal URI.

## *2.3 On the Limits of Universality*

There is a major problem with the concept of universality presented above.

It's impossible to get everyone everywhere to agree on a single label for every specific thing that ever was, is, or will be.

If you read the introductions to Semantic Web technologies around the web, you'll see lots of people focus on the importance of the URI. After all, how can you connect things if you don't agree on their labels? The focus on URI definition is especially true for those creating RDF vocabularies.

What we mean by an RDF Vocabulary is essentially the set of URIs for the edges that make up RDF graphs. The edges are what relates the things in graph, and are what give it meaning. Using specific URIs is like speaking in a specific language—hence the term vocabulary. For example, in order for two Semantic Web applications to share data, they must agree on a common vocabulary.

So if two applications have to agree on vocabulary for all concepts, then it stands to reason that all vocabularies must be set ahead of time, right? Fortunately, the a priori existence of share vocabulary turns out to be helpful, but far from necessary. In our example, foaf:name is not the first URI ever created that represents the name concept, and it's OK that another URI for the name concept wasn't reused.

Fortunately, it is very easy to translate RDF written in one vocabulary to another vocabulary. The Semantic Web technologies were built under the assumption that different people in different applications written for different purposes at different times would create related concepts that overlap in any number of ways, and therefore there are provisions and methods to make it all work together with little effort. There is no such provision in the XML or relational database worlds.

Said another way, you do not have to agree on all URIs for all things up front. In fact, it's much easier not to do so. Reuse vocabulary when possible and convenient, and don't worry too much about that when it doesn't work out.

This same universal identity conundrum also happens for resources. For example, you could consider my Linked In profile URL to be a URI representing me. This is clearly distinct from the URI that Cambridge Semantics uses, but, again, the Semantic Web offers very simple ways to merge identical concepts so that they appear as one universally.

## *2.4 Statements and Triples*

Now that you get the basics, I have to introduce some community jargon that will help you understand material you read on the Web about Semantic Web technologies.

Rather than talk in the language of nodes and edges, Semantic Web practitioners refer to statements or triples, which are representations of graph edges.

A statement or triple (they are synonymous) refers to a 3-tuple (hence triple) of the form (subject, predicate, object). This linguistic, sentential form is why RDF schemas are often called vocabularies.

As mentioned, the subject is a URI, the predicate is a URI, and the object is either a URI or a literal value.

If we represent our graph example as a set of triples, they would be:

(csipeople:rob, foaf:name, “Rob Gonzalez”)

(csipeople:rob, foaf:member,http://www.cambridgesemantics.com/)

(Note that for brevity I'm using the namespace alias csipeople for the URI namespace http://www.cambridgesemantics.com/people/about/).

RDF graphs therefore are simply collections of triples. An RDF database is often called a triple store for this reason.

However, Semantic Web practitioners found it very difficult to deal with large amounts of triples for application development. There are lots of reasons that you would want to segment different subsets of triples from each other (simplified access control, simplified updating, trust, etc.), and vanilla RDF made segmentation tedious.

At first the community tried using reification to solve this data segmentation problem (reification is essentially triples about triples), but today everyone has converged on using named graphs. See section 2.9 Submodels and Contexts.

## 2.5 Literal Data Types

RDF uses the same data types as XML, sometimes called XSD, an initialism for XML Schema Definition.

XSD provides a set of 19 primitive data types (anyURI, base64Binary, boolean, date, dateTime, decimal, double, duration, float, hexBinary, gDay, gMonth, gMonthDay, gYear, gYearMonth, NOTATION, QName, string, and time). It allows new data types to be constructed from these primitives by three mechanisms:

- restriction (reducing the set of permitted values),
- list (allowing a sequence of values), and
- union (allowing a choice of values from several types).

Twenty-five derived types are defined within the specification itself, and further derived types can be defined by users in their own schemas.

## 2.6 Data Structures

RDF has some rudimentary support for data structures:

**List –** This is effectively a linked list structure made up of a “head” node, a “rest” node and a “null” node to signify the end of the list.

**Group or Bag –** This is an unordered collection.

**Sequence –** This is an ordered collection.

**Alternate –** This is usually included with the other containers but is more like a menu of possible values rather than something that is concrete.

## *2.7 Schema*

A schema is not strictly necessary but it can be useful for organising and structuring your data.

You can define classes and a hierarchy of subclasses. These can then be used to provide types for your subjects and objects.

Predicates can be divided into data properties which have literals as their objects and object properties that have resources for their objects. Object properties can also be organised into hierarchies.

Predicates can also have a domain and range defined. Predicates can have characteristics, such as functional, symmetric or transitive and relations such as sameAs and inverseOf. These can be used by inference engines to fill in additional relationships.

The schema can also specify the expected data structures.

## *2.8 Reification*

Reification is the RDF term for metadata. A node can have a type of “statement” and be linked to the nodes of a statement via “subject”, “predicate” and “object” predicates.

This statement node can then have links to various data items such as the author of the statement, when it was inserted into the database and the provenance of the data. It could even have a cryptographic signature. This metadata is sometimes called a “nanopublication”.

An additional use is to store a confidence level for the statement. RDF statements are supposed to be facts, but facts are often just theories waiting to be disproven.

### 2.8.1 Queries and Rules

An alternative use for a “statement” node is to record statements that are not necessarily true. The main data is supposed to only include true facts. This makes it difficult to store a query in the database as it could be misinterpreted as a fact. Using a statement node to store a query avoids this problem. It also allows us to include variables in our queries.

Following on from using the statement node for queries it can also be used to hold the statements that can be inferred from a query. An inference engine can then use these rules to create a model of inferred statements.

## 2.9 Submodels and Contexts

RDF has the concept of a “context”. It is a way of labelling statements so they can be grouped together. The problem is that no two proponents of RDF can agree on exactly how contexts should be used. [They also cause a significant increase in model size since our triples have now become quadruples.]

The Redland library, on which rdfxx is built, has good support for contexts but has not properly implemented submodels. I had used contexts to create a system which enables RDF models to be used as reference data for other RDF models. We can thus build more complex databases from smaller models.

In practice it appears that the Redland support for contexts isn't all that good. The performance drops off drastically when contexts are enabled. Hence my submodel implementation does not use contexts. This means that my system does not include the ability to remove submodels from an open model.

# 3 Program Components

## 3.1 Menu and Toolbar

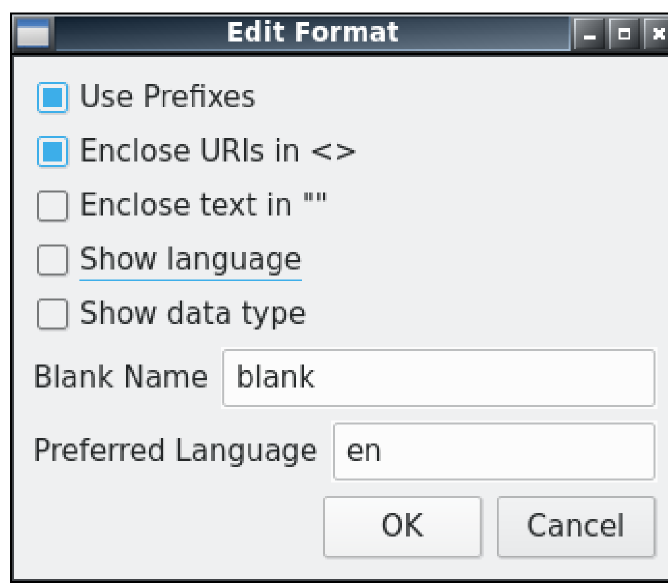
File | Exit – This leaves the program.

Edit | Prefix - This brings up a dialog for editing a prefix. (see below)

Edit | Statement - This brings up a dialog for editing a statement. (see below)

Edit | Edit – This brings up the Forms interface (see below).

View | Format – This brings up the format dialogue. This can be used to specify how data values are displayed.



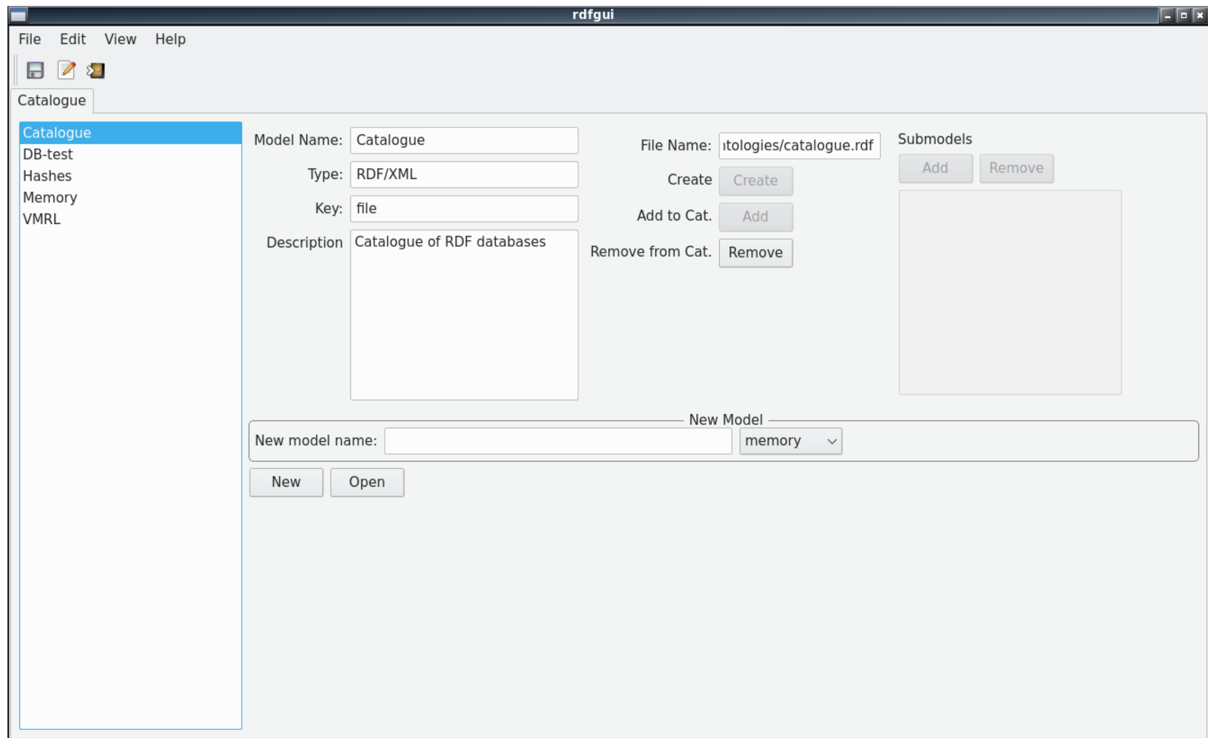

View | Visual + - This increases the span shown in the Visual tab.

View + Visual - - This decreases the span shown in the Visual tab.

Help | Info - This shows various attributes that are built into the Redland RDF library.

## 3.2 Catalogue Tab

This is used to manage a catalogue of RDF databases. Projects will often use more than one RDF model (aka database) and this tab allows you to specify all the ones you need for a project. Since rdfgui is a general purpose program its catalogue will probably end up with all of your models.




### 3.2.1 Opening a Model

On the left is a list of all the models. When one is selected its details are displayed in the panels to the right. The “Open” button can then be used to open the model. This will cause the other tabs to be displayed.

[The rdfgui program is a little unusual in that its user interface is completely redrawn whenever the model is modified. This may change in the future, but for now it is programmatically convenient.]

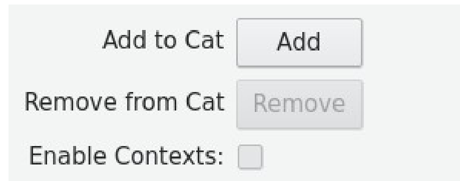
To the right of the list of models is a panel showing the data that is common to all models. These values are set when the model is entered into the catalogue.

Further to the right are the values that are specific to the type of storage used. A button is available to remove a model from the catalogue. Removal from the catalogue is non-destructive – the file or database remains.

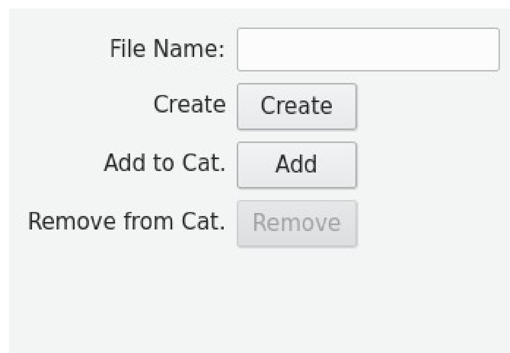
### 3.2.2 Adding a Model

Add a name for the model in the field “New Model Name”, select the type from the dropdown, and then press “New”.

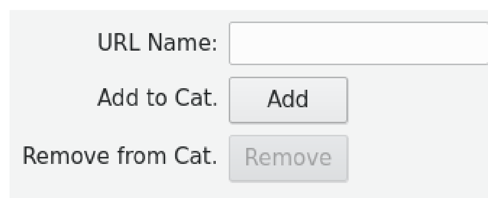
Fill in a description for the model and the other values as appropriate:



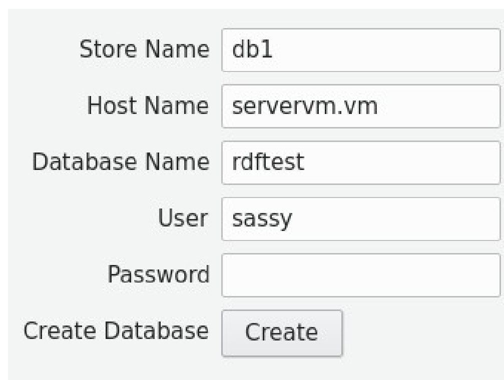

When the selected type is “memory” or “indexed” the above panel is displayed. These are both only held in memory. Use indexed for large models. You can optionally enable contexts for these models when opening them.




When the selected type is “file” the above panel is displayed. You can either enter the name of the file or leave it blank and press “Create” to create a new RDF/XML file, or “Add” to add an existing file using a file selection dialogue.




When the selected type is “uri” the above panel is displayed. You can paste in the URL of an RDF file from the internet. [Opening may be a little slow as the data is retrieved across the net.]




When the selected type is “postgresql” the above panel is displayed so you can create a triple store in a PostgreSQL database. The “Store Name” should be the same as the model name [this will be automatically set in future]. The “Host Name” is the location of the database server. [Should also have the port number.] The “Database Name” is the name of a database on the server which must be created beforehand. The User and Password are for the log in to the database. [Buttons for adding and removing an entry in the catalogue for an existing database will be added.]

### 3.2.3 Submodels

On the far right a panel shows the submodels that the selected model imports and buttons for adding and removing submodels.

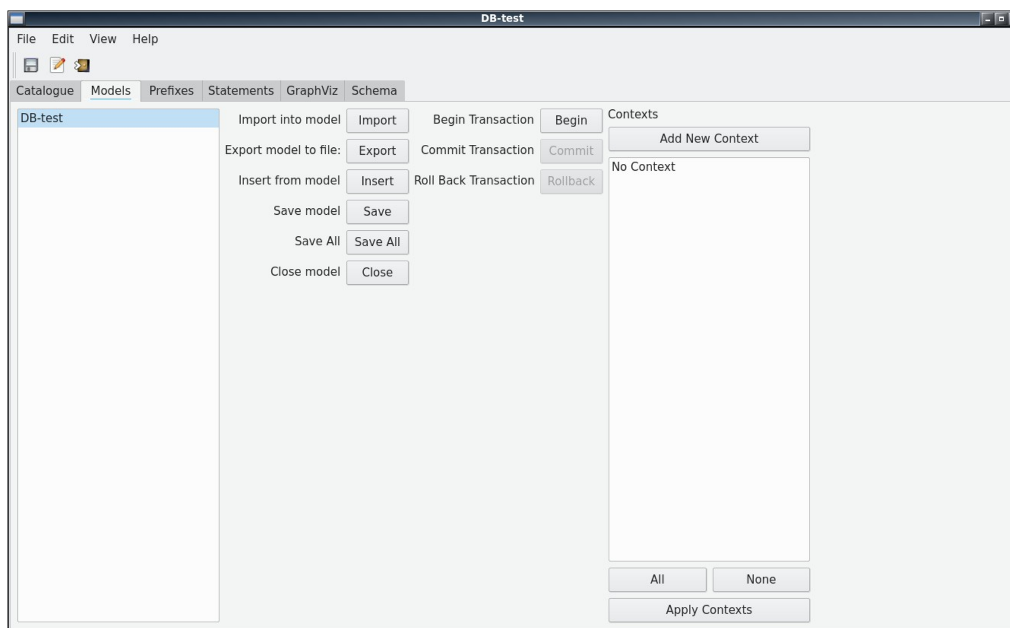
The underlying C library does not have real support for submodels, so I have created my own. When a model has submodels it is not opened directly but a separate model, using hashes, is created and the selected model is loaded into it, followed by a single instance of each of the submodels, recursively. All updates are applied to both the hashes model and the selected model, however some actions are not fully supported.

Adding or removing submodels has no effect other than to load their statements when the model is opened. The models and the underlying storage are not affected – it is entirely an artifact of the catalogue.

The Disable button allows you to open the model without the submodels whilst retaining the association.

## 3.3 Models Tab

The Models tab is used to switch between the open models. A list of the open models is displayed.




It provides buttons for managing the models:

**Import** – Opens a file selection dialogue so that another RDF/XML file can be merged into the current model.

**Export –** Saves the current model as an RDF/XML file.

**Insert –** Opens a model selection dialogue and then merges the selected model into the current model.

**Save** – Synchronises the model with its storage. (Not available for URLs or in-memory models)

**Save All –** Synchronises all open models.

**Clear** - Removes all the data from the model. (Note that submodels are not affected.)

**Close** – Closes the selected model.

### 3.3.1 Transactions

For models that use a database as storage buttons are displayed that allow you to use transactions. Normally changes to the model are only committed to the storage when you press Save. However if you press Begin (Transaction) the changes will be committed when you press Commit. Pressing Rollback will back out the changes.

This is mostly for testing the transaction mechanism which is of more use in applications.

### 3.3.2 Contexts

The models tab also provides the means for adding or removing contexts for a model. This is only made available for those storage modes that support contexts, which is memory, indexed and postgresql.

My basic rule for contexts is “Don’t bother, use a submodel.”

The problem with the contexts list is that it is refreshed whenever the model is updated and subsequently only shows the contexts that are actually in the model. Hence you can add a context to the list but it will quickly disappear.

The trick is to add the required contexts, select them, and then press “Apply Contexts”. This stores them in the application so that they are available for the Statement Dialog, for example.

## 3.4 Prefixes Tab

The URI is the fundamental mechanism for specifying the identity of a resource in RDF. While it is fine for machines to use they are a bit of a pain for us humans. I find that two similar URIs are very easy to confuse with each other. They are also pain to have to type.

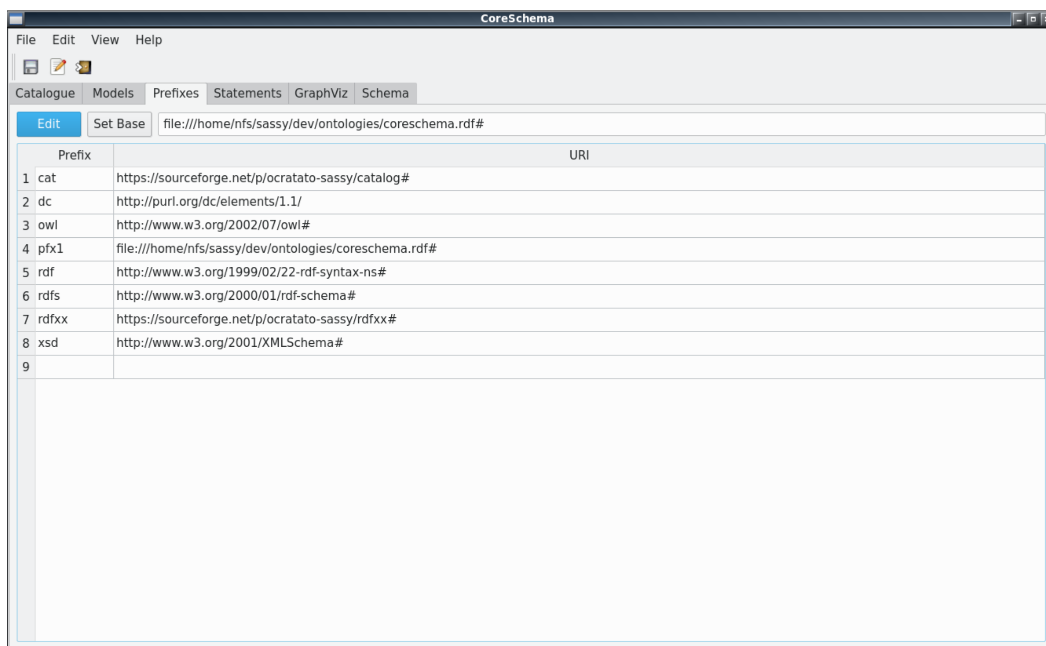
Hence RDF borrows the XML namespace concept to provide a shorthand mechanism for URIs called a “prefix”. They are generally a short mnemonic string ending in a colon (:). The part after the colon usually corresponds to the part after the hash (#), or after the last slash(/), in a URI.

The librdfxx library predefines a set of prefixes. There are the well known ones: rdf, rdfs, xsd, dc, and owl. It also defines rdfxx for some of the things we need for this library. The catalog library will also introduce cat:. In addition any model that is loaded can introduce additional prefixes either using the namespace mechanism in an RDF/XML file, or as saved into the RDF by librdfxx. URIs that don’t have a prefix are assigned one such as pfx1, pfx2.

The rdfxx library also saves the prefixes in the model. This is necessary for models that are stored in databases since they do not have the namespace clause of XML.

RDF (as does XML) includes the concept of a “base” namespace which is the namespace that applies when none is specified. Some parts of RDFGUI rely on the base being set, so it is good practice to always set it whenever you change models.

Selecting an entry in the list of prefixes enables the buttons for editing a prefix and for setting one as the base for the current model.



## *3.5 Statements Tab*

Statements are the building blocks of an RDF database. This tab provides the ability to view, search and edit the statements of a model.

Currently it displays the entire set of statements for a model. This is OK for small models but will be problematic for very large models. Various means will be provided later to filter the set that is displayed down to a more manageable set.

### 3.5.1 Statements Table

The table takes the entire tab space and shows three columns – the subject, predicate and object. This is loaded as soon as the model is opened.

[If contexts are supported a fourth column is displayed to show the context of each statement.]

The columns can be sorted by clicking the title, and a second [and subsequent] click will reverse from ascending to descending [or vice versa].

When a cell is clicked it is highlighted, and [this is the interesting bit] so are all the other instances of the cell contents. Sorting will bring them all together for the column selected and the selections will be made visible.

This makes it possible to follow a chain of statements. First select a node in the Subject column and then sort the list. Then select a cell in the Object column that has the predicate you are interested in. Finally sort the Subject column to see how your selected object is used as a subject. Repeat to follow the chain.

The reverse also works to follow a chain in the reverse direction.

The subject of a selected statement becomes the highlighted node in the Visual Tab display [see below].

CoreSchema
File Edit View Help
Catalogue Models Prefixes Statements GraphViz Schema

| | Subject | Predicate | Object |
|---|---|---|---|
| 1 | rdfs:Resource | rdf:type | rdfs:Class |
| 12 | owl:propertyDisjointWith | rdf:type | owl:ObjectProperty |
| 10 | owl:inverseOf | rdf:type | owl:ObjectProperty |
| 11 | owl:equivalentProperty | rdf:type | owl:ObjectProperty |
| 4 | owl:TransitiveProperty | rdf:type | rdfs:Class |
| 17 | owl:TransitiveProperty | rdfs:subClassOf | owl:ObjectProperty |
| 8 | owl:SymmetricProperty | rdf:type | rdfs:Class |
| 21 | owl:SymmetricProperty | rdfs:subClassOf | owl:ObjectProperty |
| 7 | owl:ReflexiveProperty | rdf:type | rdfs:Class |
| 20 | owl:ReflexiveProperty | rdfs:subClassOf | owl:ObjectProperty |
| 2 | owl:ObjectProperty | rdf:type | rdfs:Class |
| 13 | owl:ObjectProperty | rdfs:subClassOf | rdfs:Resource |
| 9 | owl:IrreflexiveProperty | rdf:type | rdfs:Class |
| 22 | owl:IrreflexiveProperty | rdfs:subClassOf | owl:ObjectProperty |
| 5 | owl:InverseFunctionalProperty | rdf:type | rdfs:Class |
| 18 | owl:InverseFunctionalProperty | rdfs:subClassOf | owl:ObjectProperty |
| 3 | owl:FunctionalProperty | rdf:type | rdfs:Class |
| 14 | owl:FunctionalProperty | rdfs:subClassOf | owl:ObjectProperty |
| 15 | owl:FunctionalProperty | rdfs:label | Functional Property |
| 16 | owl:FunctionalProperty | dc:description | A subject can only have one object connected by th... |
| 6 | owl:AsymmetricProperty | rdf:type | rdfs:Class |

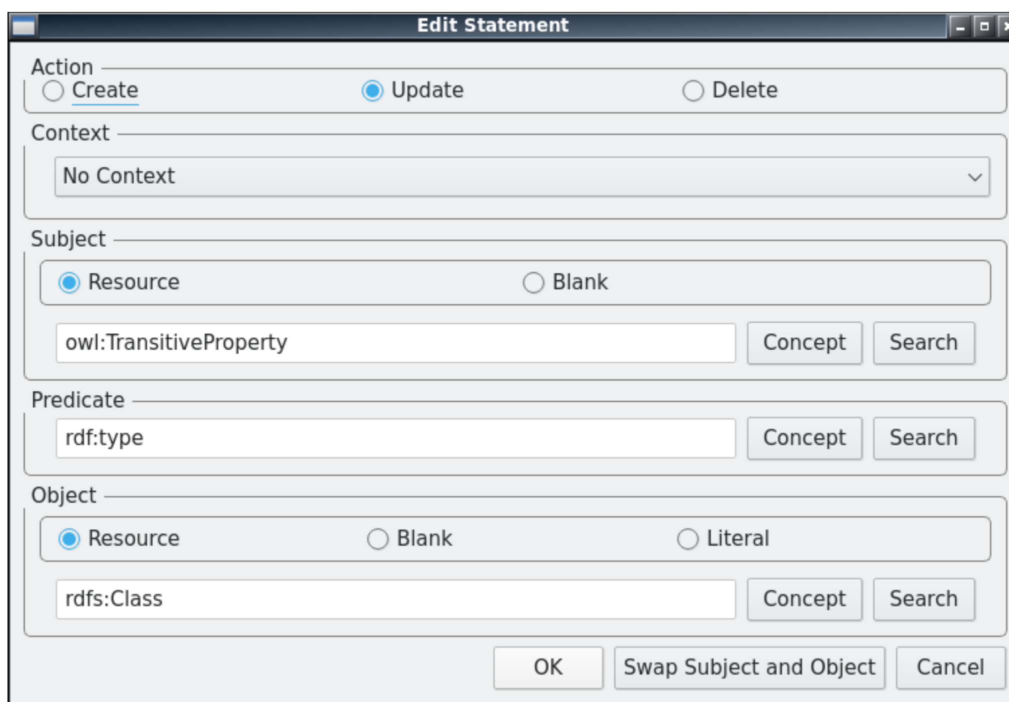
### 3.5.2 Statements Dialog

Double clicking a statement, or selecting Edit | Statement from the menu brings up the Statement Dialogue.

At the top you can select from Create, Update or Delete. Delete removes the statement, Update changes the selected statement, Create makes a new statement.

Next is a drop down for contexts. “No Context “ is the default. The drop-down will have the contexts you set using “Apply Contexts” in the Models Tab.

There are then three panels for the subject, predicate and object.




The Subject and Object panels have check boxes to indicate what type of node. [Predicates are always resource nodes.]

Each panel has a Concept button for the predefined concepts, such as rdf:type.

Each panel has a Search button which brings up a panel where you can enter a SPARQL query. You can then select a result and it gets pasted into the field when you press OK.

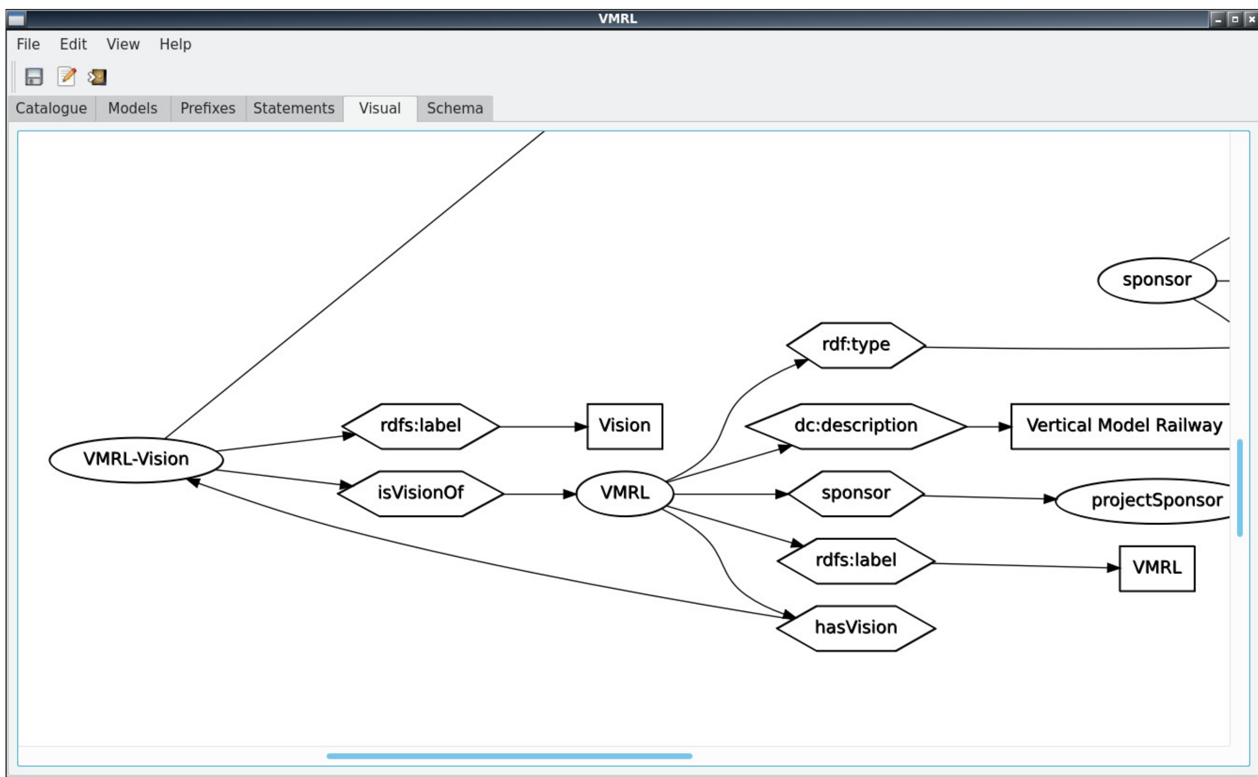
A button at the bottom enables you to swap the Subject and Object, provided the Object is not a Literal since a Subject cannot be a Literal.

If Literal is selected for the Object a larger field is shown so you can enter any amount of text. Drop-down menus for the language and data type are also shown.

## 3.6 Visual Tab

The Visual tab provides a visual view of the model.

If there are more than 100 statements it changes to only show a view of the model centred on the selected node. Selecting a node will cause the graph to be redrawn around the selection. Since there are very few ways to influence how GraphViz generates the diagram this may be a bit confusing as the model changes shape and location.

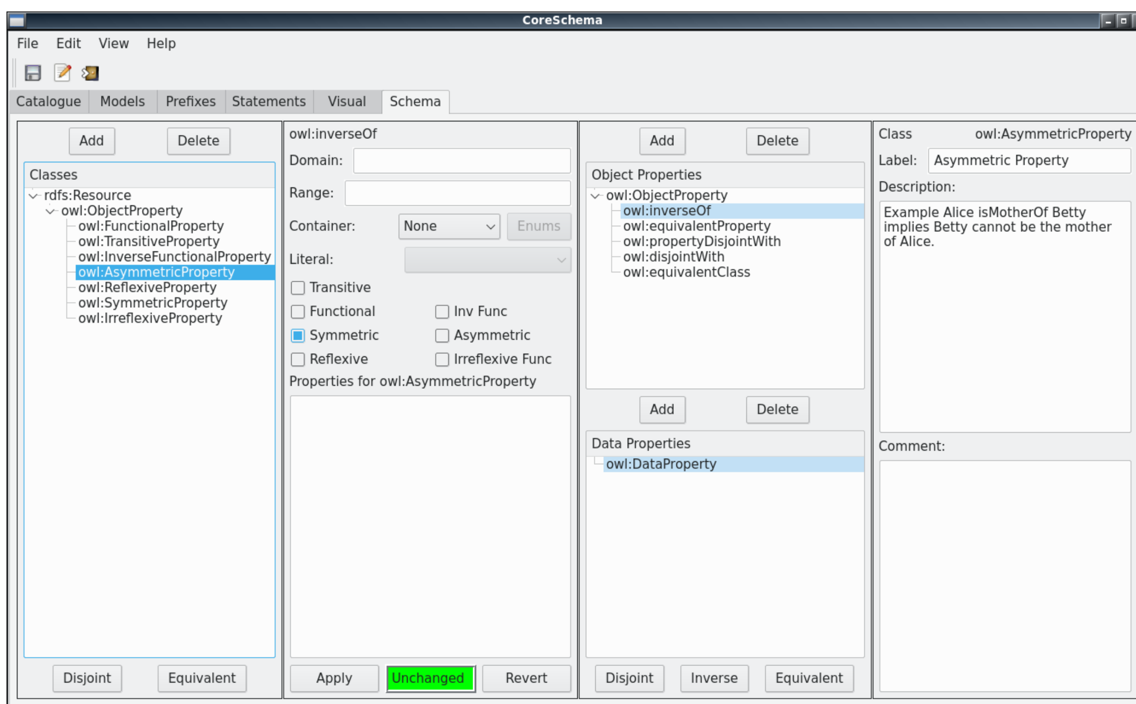



You can drag the model, or use the mouse wheel to zoom.

## 3.7 Schema Tab

The Schema Tab is used to enter a schema. This is not strictly necessary for RDF but the Forms part of RDFGUI uses it to automatically generate the data entry forms. A schema can help you better organise your understanding of the data.

Note that changes made on this tab are not applied to the current model until Apply is pressed. Until then you can remove all the changes with the Revert button. When Apply is pressed the model is updated which causes the model to be redrawn on all of the tabs, including this one.




### 3.7.1 Class Panel

The left (first) panel is for defining classes or types of things.

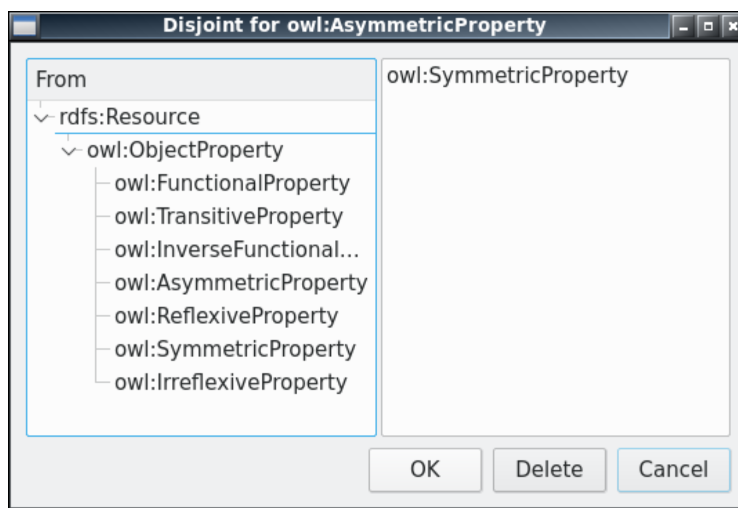
These can be organised into a hierarchy where rdfs:Resource is the most general possible thing.

Pressing the Add button will create a dummy entry under Resource. Type in your name for the new class and press Enter when completed. You can then position the new class by dragging it within the tree while holding the Shift key. [Without the shift key it will make a copy. Accidental copies can be removed by selecting it and pressing the Delete button. Sadly, this will delete all the instances, not just the one you selected.]

While dragging you will see either an outline around an existing entry or a line between two entries. If you drop while there is an outline then the new class

will be positioned as a child, otherwise it is positioned as a sibling.

Buttons at the bottom will bring up a dialogue box for specifying the classes that are disjoint to the selected class, or classes that are equivalent. If two classes are disjoint then it would be an error for an item to have both for its types [however this is not enforced]. You would specify two classes as being equivalent if you have imported two different schemas [vocabularies] that have given different names to the same concept. Again, it would be an error for an item to have both for its types [also not enforced].




On the left is the class tree. On the right is the list of classes which are disjoint for “AsymmetricProperty”. You can add to the list by dragging from the tree to the list. Items on the list can be removed by selecting them and pressing Delete.

The “Equivalent” button works the same.

### 3.7.2 Property Panel

The third panel is for defining properties, which are the predicates in the statements. It may be subdivided into “object properties” at the top and “data properties” at the bottom. Object properties are predicates that have resources as their associated object, while data properties have a literal for their objects.

If you open an existing model that was made by some other system it probably won’t have separate object and data properties. In this case the properties will all be shown in a single tree.

The properties can be organised into hierarchies.

Entry and removal is similar to that used for classes.

At the bottom are button for Disjoint and Equivalent that work identically to the ones for classes. In addition is a button for “Inverse” where you specify the property that is the inverse of the selected object property.

### 3.7.3 Property Attributes Panel

The second panel allows you to enter some attributes for the properties.

[This panel works best with separate object and data property lists.]

**Domain**: This allows you to specify the domain that the property applies to. The domain corresponds to the subject in a statement. You should specify the most general class that applies. You can drag a class from the left panel into this field.

**Range**: This allows you to specify the range that the property applies to. The range corresponds to the objects that are resources in a statement. It is disabled for data properties. You can drag a class from the left panel into this field.

**Container**: There is a drop-down menu where you can select a type of container:

> **Group**: Also known as Bag. This is for an unordered set, despite the fact that it represented as a numbered list in RDF.
>
> **Sequence**: This is for an ordered list. This is the RDF version of a vector or array.
>
> **List**: This is the RDF version of a linked list. It divides the list into a head and rest recursively with a null to signify the end of the list.
>
> **Alternate**: This is a more like a menu of allowable values. When selected the you can press the Enums button and define the alternatives.

**Literal**: A drop-down menu allows you to select the data type for the selected data property.

Under the Literal are seven check boxes where you can specify characteristics for the selected property. Conflicting characteristics can be selected if you want, but it is, obviously, a bad idea.

[RDFGUI does not enforce much consistency in the models. This enables you to view models that may be inconsistent. It also allows you to create inconsistent models so that you can create test data for other programs.]

The lower part of the panel is a list of properties for the currently selected class [as specified by the domain of the properties].

### 3.7.4 Description Panel

The right hand panel displays the currently selected object and its type.

There are fields for entering a label, description and comment.

**Label**: The classes and properties generally have names the are in camel-case. These are not entirely readable, so the label field can be used to give them much more readable names.

**Description**: A single word can often be a bit ambiguous in its meaning, even if its a phrase in camel-case. This field allows you to enter as much text as necessary to describe the object. This description should be in the context of the model's data.

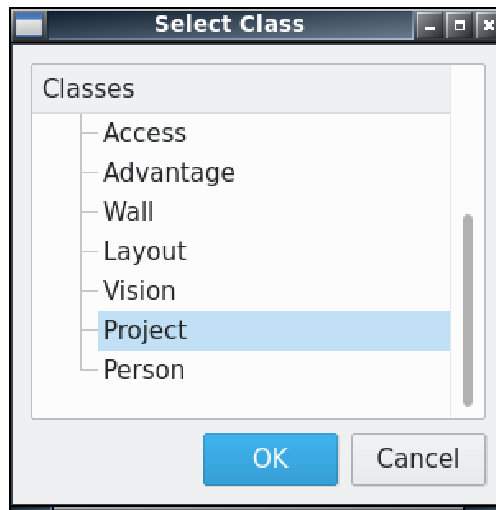
**Comment**: This allows you to add comments about the data that are not part of the model. For example it might contain an argument for doing things differently.

## 3.8 Forms

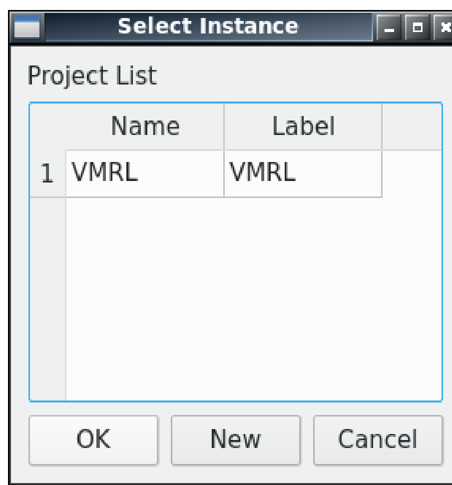
Selecting Edit | Edit… from the main menu or the Edit icon on the tool bar starts the forms interface.

This interface consists of a series of dialogue boxes that allow you to enter statements that conform to the schema.

This starts by displaying the list of classes that have been defined:




Selecting a class and pressing OK brings up a dialogue containing a list of the existing individuals that have the class as their type:




From here you can either select New to create a new object, or select an entry from the list and press OK to bring up a dialogue for the selected individual:

rdfgui

Project

Generate Name VMRL Label VMRL

Description

Vertical Model Railway Layout

sponsor hasVision

OK Cancel Apply

The dialogues for an individual all have Name, Label and Description fields. There is a Generate button that will append a sequence number to the string entered in the name field. This is useful if the individual does not have a well defined identity and you are not using blank nodes.

The example shown has buttons for the object properties defined for this class. Note that the “hasVision” button is using the name of the object property, however if the property had a label defined the label would be used instead.

Pressing the “sponsor” button, for example, will bring up a class dialogue for Person as this is the range defined for the sponsor object property. It will be populated from individuals that are sponsors of the project. Selecting from that list brings up another dialogue:

rdfgui

Person

Generate Name projectSponsor Label Project Sponsor

Description

name Brenton Phone Number 02 55545761 email rossb@example.com

title the grand boss

OK Cancel Apply

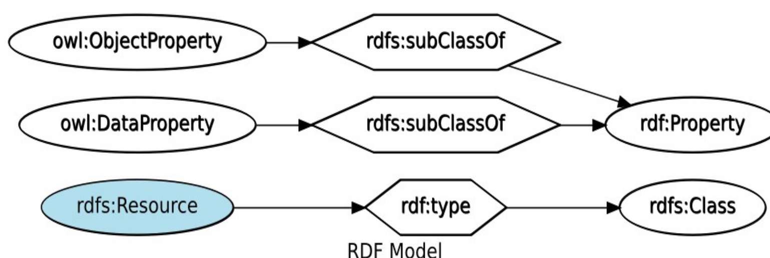
# 4 Tutorials

This will evolve into proper tutorials over time. Initially it will contain some examples of how actions in the user interface modify the model.

## 4.1 Schema Examples

### 4.1.1 Classes

In an empty model pressing “Apply” will create the following statements:

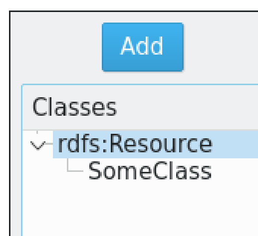



The blue colour just means it was the selected node.

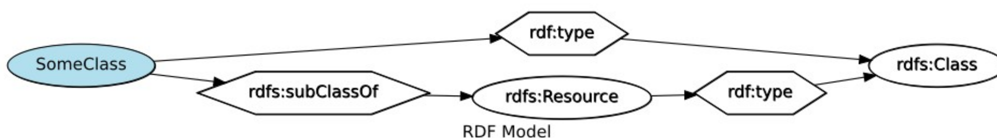
It states that Resource has a type of Class. Resource is the name for the root of the class hierarchy, which means it is the most general possible thing.

There are also two statements linking Property classes that will be covered below.

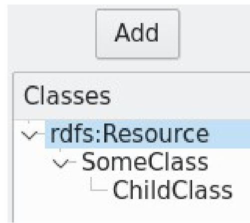
If we add a class:



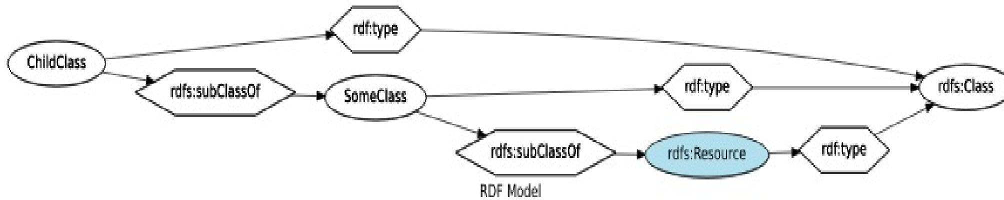

We get the following model:




Two statements have been added: One that says SomeClass has a type of Class, and one that says it is a subclass of Resource.

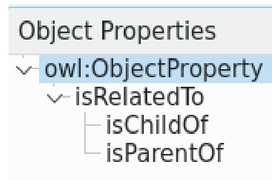



Adding further classes will create these two statements saying the new class has a type of class and saying which class it is a subclass of.

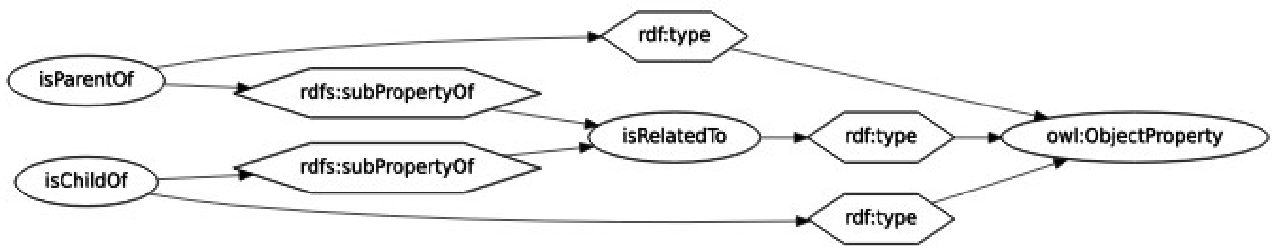



### 4.1.2 Object Properties

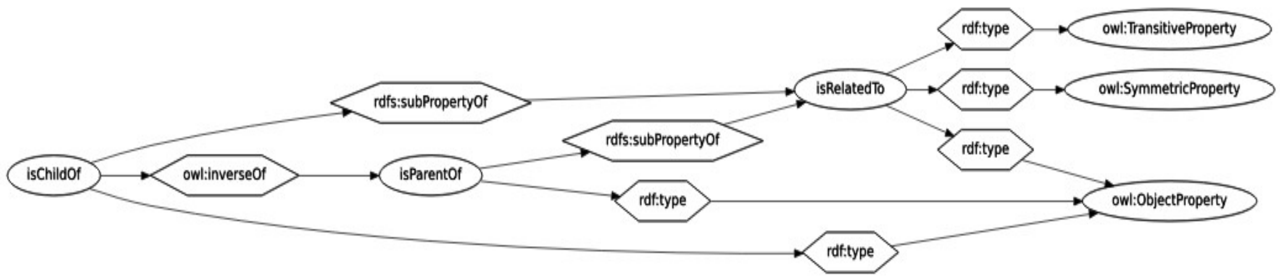
You can have a hierarchy of properties.:



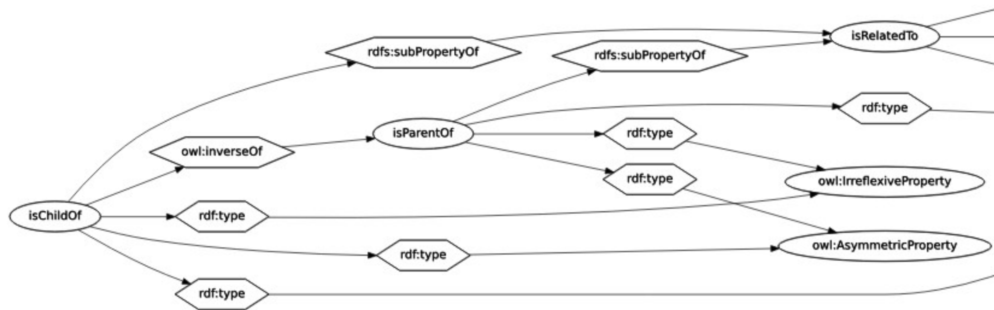

which results in this model:




We can embellish this a little by indicating that “isRelatedTo” is transitive and symmetric and that “isParentOf” is the inverse of “isChildOf”:

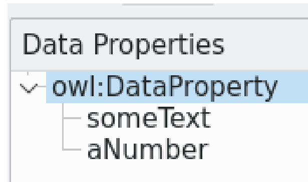



Since two people cannot be their own parents or children we can further add asymmetric and irreflexive characteristics.

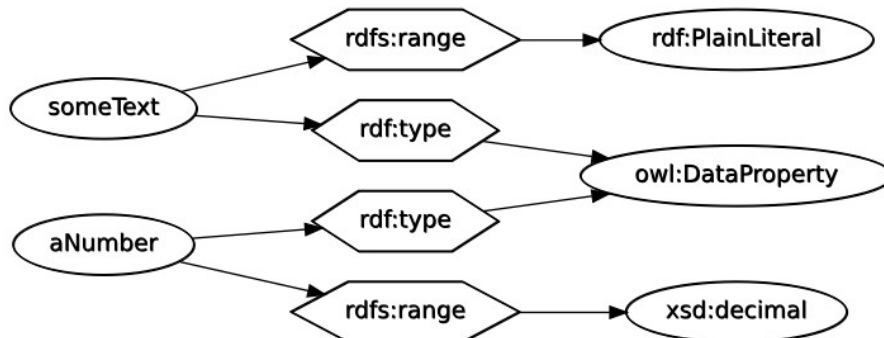



## 4.1.3 Data Properties

For properties that have a literal value as an object they are made to be of type dataProperty:



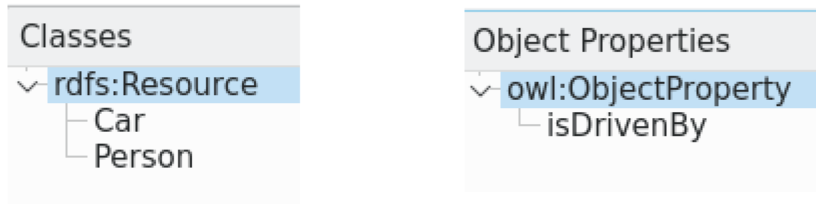

To specify the data type for these properties we specify the range that the object is allowed to have. The model will look similar to this:



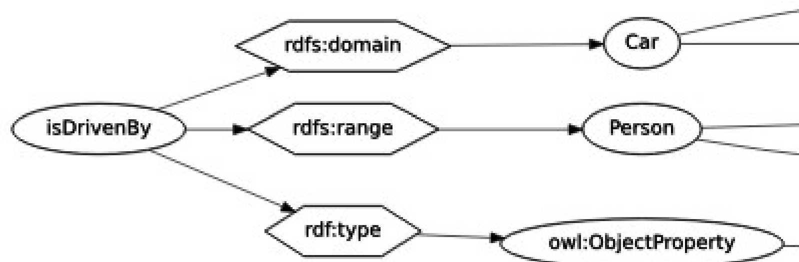
### 4.1.4 Domain and Range

For a property you can specify the domain (subject) and range (object) to which it applies. This ties the property to these classes and needs to be done with a bit of care as you may place unnecessary constraints on where the property can be used. However, this only applies where these constraints are enforced, which is not the case in RDFGUI.

For example with the following classes and properties:




We can specify that "isDrivenBy" has a domain of Car and a range of Person. The resulting model will look like:



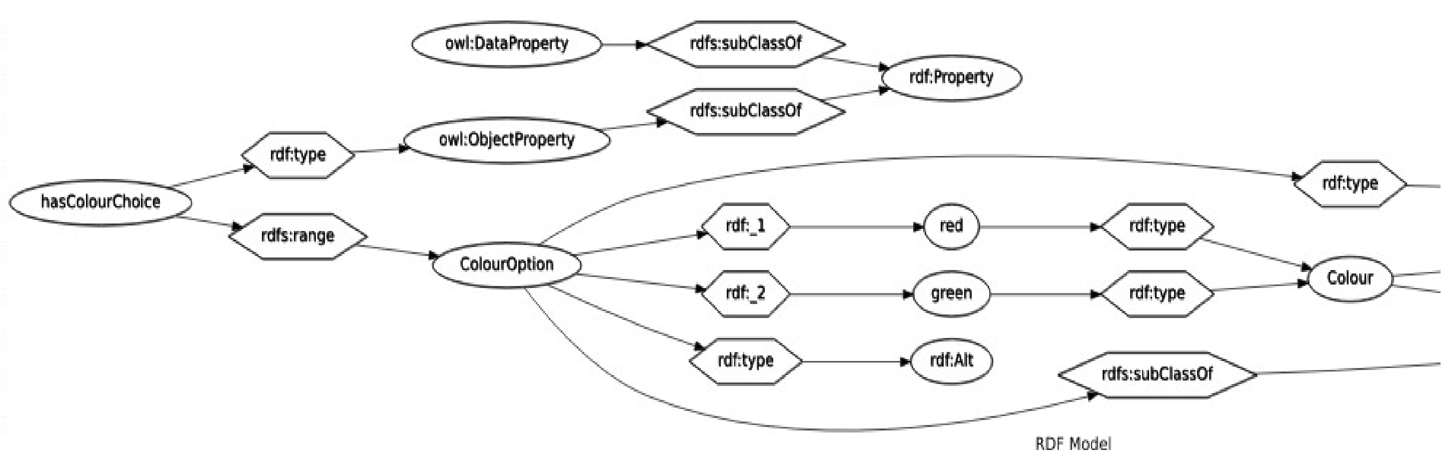

The problem now is that if we introduce the Locomotive class we could not use isDrivenBy even though it needs a driver. A solution is to introduce a super class "Drivable" that has Car and Locomotive as subclasses and change the domain of isDrivenBy accordingly. Remember that RDF uses multiple inheritance so Car can still be a subclass of MotorVehicle, for example.

### 4.1.5 Containers

Most of the container examples are in the next subsection of Forms examples. However, “Alternate” is handled by the Schema Tab since it is more like a menu of available options than a container.

To set up a container first set the range of the property to the type of object that is the container. Then change the drop-down from “None” to the type you require: Group [aka Bag], Sequence, Alternate or List.

If you select Alternate then the Enums button is enabled. Pressing this will bring up a dialogue where you can enter the alternatives [these are resource nodes].



## 4.2 Forms Examples