RDF2Graph Documentation

Table of Content

Installation

Option 1: Using the Boxed version.

If using Windows please use this option. A virtual box with all the software requirements is distributed along with RDF2Graph, which can be downloaded at http://download.systemsbiology.nl/downloads/RDF2Graph.ova

Virtual box can be downloaded at Virtual box download (https://www.virtualbox.org/wiki/Downloads). Execute installer and load the RDF2Graph.ova file.

When the virtual machine is finished starting up, please login with the rdf2graph user and use 'rdf2graph' as password. Upon login please start terminal, which is available under activities. RDF2Graph is installed within the home directory under the rdf2graph directory.

Option 2: Full installation

Requirements

RDF2Graph has been created and tested on Ubuntu. RDF2Graph needs Java 1.8, Jena, Cytoscape 3.2, Maven2, GIT, NodeJS, python and html2text.

```
#install java 8
sudo add-apt-repository ppa:webupd8team/java
sudo apt-get update
sudo apt-get install oracle-java8-installer
sudo update-java-alternatives -s java-8-oracle

#maven2,git, nodejs and html2text
sudo apt-get install maven2 git nodejs html2text npm
```

Jena instalation: please download Jena download.
(https://jena.apache.org/download/index.cgi) Extract to directory of choice
Cytoscape installation: Download Cytoscape. (http://www.cytoscape.org/download.php)
Make downloaded file executable using
`chmod +x Cytoscape_3_2_0_unix.sh`
Execute installer and install to directory of choice.

Modifying the path: To include Jena and Cytoscape into the path add the following lines to `~/.bash.rc and` restart command console.

```
export JENAROOT=<your jena directory>
PATH=$JENAROOT/bin:$PATH
PATH=<your cytoscape path>:$PATH
```

Setup

To install RDF2Graph :

```
git clone https://github.com/jessevdam/RDF2Graph
cd RDF2Graph
./setup.sh
```

General usage

The main RDF2Graph Java jar can be used to execute the structure recovery. It extracts the structure from an RDF resource and stores the results in a local database (Jena TDB). The simplification process can be enabled with the `--executeSimplify` option.
After completion of the recovery step the results can be exported either to a network representation (Cytoscape) or to Shex OWL orOWL ontology files. This can be done by using one of the exporter tools. These can be found in the following folders: **owlExporter**, **shexExporter** and **cytoscapeExporter**. Each folder contains an executable bash script called *export.sh*.

The recovery process performs a set of queries that are computational demanding, so please do not execute the recovery directly on any public endpoint, since this might lead to the public endpoint to become overloaded
Instead we recommend to set up your own SPARQL endpoint and load the data of the public endpoint.

Please note that some versions of Virtuoso contain a bug, which make Virtuoso not return all results as it is should return. This problem could make RDF2Graph to either stop or generate incomplete results.

NOTE: The queries executed by RDF2Graph, contain limitation to limit the execution time, however the results might become incomplete.
NOTE!!: If the recovery fails or you want it to rerun it with different settings, please remove the output folder.
NOTE: If you get weird Java exceptions, please first try to remove the output folder.

Parameters RDF2Graph

RDF2Graph can be executed with:

```
java -jar RDF2Graph.jar <output directory> (<user>:<pass>@)?<URL
of the SPARQL endpoint>([<uri of graph>])? --multiThread <number
of threads> --all --eachThreadOnePort --collectPredicateStatistics
--collectClassStatistics --collectShapePropertyStatistics
--checkPredicateSourceClass --collectForwardMultiplicity
--collectReverseMultiplicity --executeSimplify
--treatSubClassOfAsIntanceOf --keepOWLClasses
```

Options

```
<output directory>: The directory in which the Jena TDB database
will be stored, which contains the results of the structure
recovery.

(<user>:<pass>)?@<uri of sparql endpoint>([<uri of graph>])?:
Sparql endpoint on which the recovery has to be executed.
```

- `<user>:<pass>@... User and password can be optionally given`
- `<URL fo the SPARQL endpoint> The full URL of the Sparql endpoint, for example` http://127.0.0.1:8080/main/sparql

- ...[<uri of graph>] Optional parameter to specify the default graph.

--multiThread <number of threads>: Specify the number of concurrent Threads/requests that will be executed on the Sparql endpoint.

--eachThreadOnePort: In case your Sparql endpoint does not support multi threading, use this option to tell RDF2Graph to use for each concurrent Thread a different instance. If the main instance runs on port 7000 then the other instances has to be running on port 7001, 7002, …,7000 + <number of threads>.

--all: Same as --collectPredicateStatistics
--collectClassStatistics --collectShapePropertyStatistics
--checkPredicateSourceClass --collectForwardMultiplicity
--collectReverseMultiplicity

--collectPredicateStatistics: Tell RDF2Graph to collect the predicate statistics.

--checkPredicateSourceClass: Tell RDF2Graph to check if all instances using a specific predicate have a type defined.

--collectClassStatistics: Tell RDF2Graph to collect the class statistics.

--collectShapePropertyStatistics: Tell RDF2Graph to collect the property statistics per class type.

--collectForwardMultiplicity: Tell RDF2Graph to collect the forward multiplicity of each type link.

--collectReverseMultiplicity: Tell RDF2Graph to collect the reverse multiplicity of each type link.

--treatSubClassOfAsIntanceOf: Tell RDF2Graph to consider the rdfs:subClassOf predicate to be also treated as a rdf:type predicate.

--executeSimplify: Perform the simplification step. This options can also be applied on a already created RDF2Graph database, in which only the simplification step will be executed.

--removeOWLClasses: Any recovered classes other then OWL:Class related to the OWL ontology itself will be removed from the results.

--useClassPropertyRecoveryPerClass: Optimization parameter, which can be used to make RDF2Graph search for the class properties per class recovered. Do not use it if not needed.

Cytoscape Exporter

The cytoscape exporter can be executed with:

```
cytoscapeExporter/export.sh <input directory> <keep> <output>

<input directory>: The directory in which the Jena TDB database is
stored, which will contain the results of the structure recovery.

<keep>: options to specify whether concept classes (classes for
which no instances are found) should be included in the overview.
Either specify 'true' or 'false'.

<output>: Specify output mode, which can be one of the following
options
  • view, opens Cytoscape and directly shows the network.
  • <filename>.cys, creates cytoscape session file and stores it
    in the specified filename.
  • <filename>.xgmml, creates one xgmml file(<filename>.xgmml)
    that contains the network view and one xgmml
    file(<filename>_error.xgmml) that contains the error report.
```

OWL Exporter

The OWL Exporter can be executed with:

```
owlExporter/export.sh echo <input directory> <include reverse
cardinality> <output>

<input directory>: The directory in which the Jena TDB database is
stored, which contains the results of the structure recovery.

<include reverse cardinality>: Tell the owl exporter to include
the reverse cardinalities in the owl file. Either specify 'true'
or 'false'.

<output>: Specify the output owl file. The used output format is
N3. Please use Protoge to convert to other formats.
```

SHEX Exporter

The SHEX exporter can be executed with:

```
shexExporter/export.sh echo <input directory> <output>

<input directory>: The directory in which the Jena TDB database is
stored, which contains the results of the structure recovery.

<output>: Specify the output SHEX file.
```

Example usage:

Here we give a detailed example on how to recover the structure from an example rdf file generated by SAPP. This example is included within the virtual box as ~/Examples/run.sh. The associated example RDF file from SAPP can be found at doc/EcoliBL21_SAPP.ttl and in the virtual box at ~/Examples/EcoliBL21_SAPP.ttl.

First you need to load the the RDF file into a triple store. Here we use Jena Fuseki. If your not using the virtual box setup then first download Fuseki from Fuseki Download (http://jena.apache.org/download/index.cgi). Then extract to directory of choice. Then start the Fuseki server with the example file and wait until its loaded. This can be done with the following commands.

```
cd <your fuseki directory>
./fuseki-server --file <example.ttl> /SAPP
```

Then use RDF2Graph to recover the structure with the following comments.

```
cd ~/rdf2graph
java -jar RDF2Graph.jar <result directory>
http://127.0.0.1:3030/SAPP/query --all --executeSimplify
```

Then use the cytoscape exporter to view the results

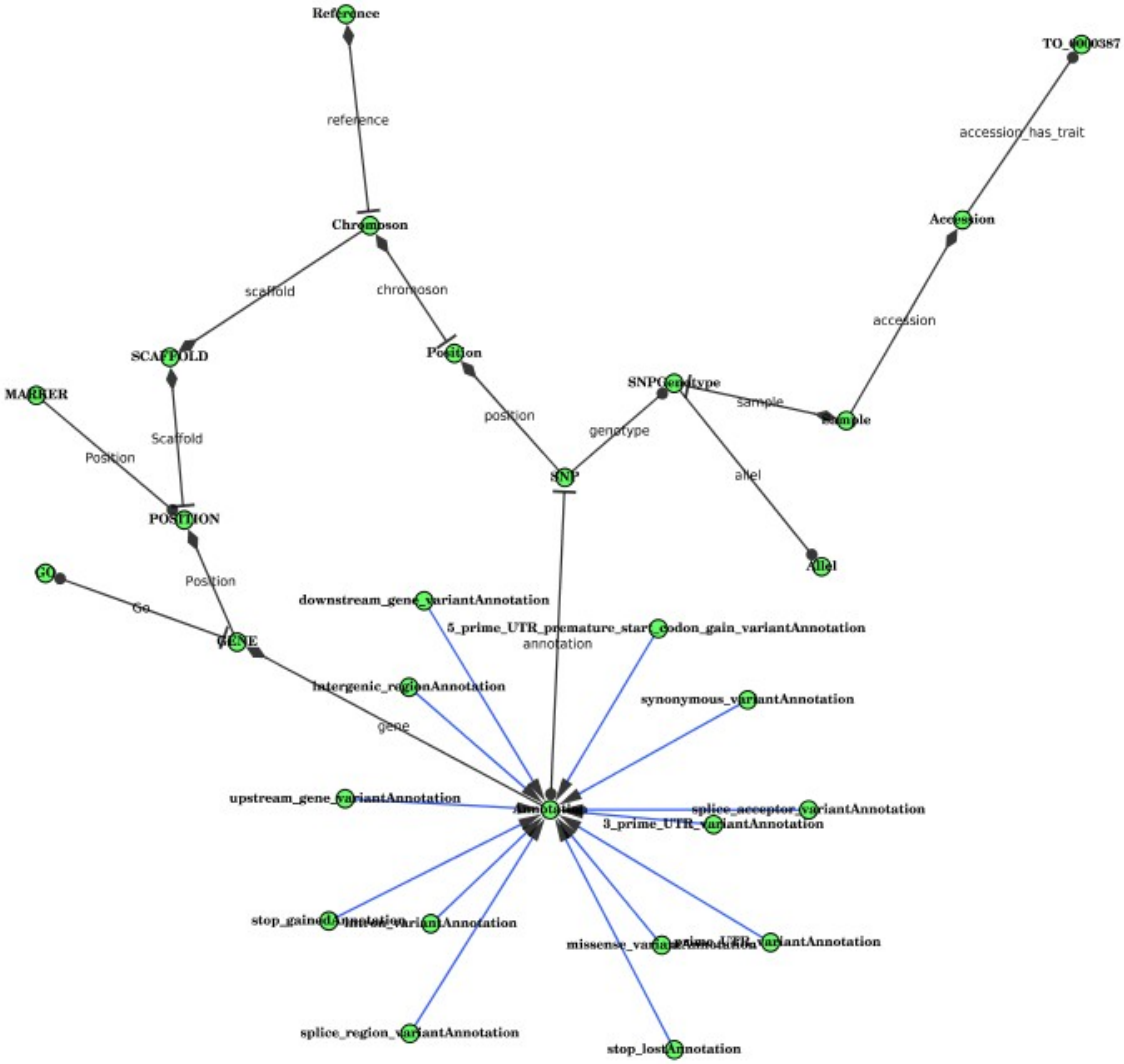
```
./cytoscapeExporter/export.sh <result directory> true view
```

Additional notes about the exporters

Cytoscape Exporter: Visualization off the RDF structure in Cytoscape

The following image give an example view of the network representation.

- Each node in the network represents one Class.
- Black edges represent an object project between two classes.
  - A diamond represents forward multiplicity of 1..1 or 0..1
  - A round blob represents forward multiplicity of 0..N or 1..N
  - A T-bar represents reverse multiplicity of 1..N or x..N
- Blue edges represent an rdfs:subClassOf relationship between two classes.



Upon selection of a class node the following details will be shown in the node browser

- name, short name of the node as depicted in the network
- count, number of instances of the class
- child instance count, number of instances including the number of instances of all the child classes
- full IRI, the full IRI of the class
- other, please use the hide function to hide these columns

shared name	name	hide	count	child instance count	/ full iri
http://pbr.wur.nl/VCF/SNP	SNP		6428		<http://pbr.wur.nl/VCF/SNP>

Node Table | Edge Table | Network Table

Figure 1: Node information: table displaying the node information

The 'select adjacent edges' (alt+e) function will show all edges including those linking to the hidden data type. All links to a data type such as for example xsd:integer and xsd:string are

hidden. Note that also inward edges are also selected so please sort the 'source' column, this will make is easy to see which edges are the outgoing edges.

Upon edge selection the following information will be shown in the edge browser.

- source, the name of the source class
- predicate name, the name of the predicate of the 'type link'
- destination type, the name of the target class or data type
- forward multiplicity, the forward multiplicity of the 'type link'
  - 1..1, each source instance has exactly one reference to the target (rs:Exactly-one)
  - 1..N, each source instance has one or more references to the target (rs:One-or-may)
  - 0..1, only some source instances have at most one reference to the target (rs:Zero-or-one)
  - 0..N, some source instances have one or even more than one reference to the target (rs:Zero-or-many)
- reverse multiplicity
  - 1..1, each target instance has exactly one reverse reference to the source (rs:Exactly-one)
  - 1..N, each target instance has one or more reverse reference to the source (rs:One-or-may)
  - 0..1, only some target instances have at most one reverse reference to the source (rs:Zero-or-one)
  - 0..N, some target instances have one or even more than one reverse reference to the source (rs:Zero-or-many)
  - X..1, target instances have no more than one reverse reference to the source(RDF2Graph:x_1)
  - X..N, target instances can have more than one reverse references to the source(RDF2Graph:x_n)
- is_simple: True if target is a data type and False if target is a instance of a class
- full predicate, the full predicate of the *type link*
- other, please use the hide function to hide these columns

sh...	sh...	name	inte...	source	predicate na...	destination t...	forwar...	revers...	refere...	is_sim...	full predicate
htt...	htt...	http://...	http:...	SNP	annotation	Annotation	0..N	1..N	14376	false	<http://pbr.wur.nl/VCF/annotation>
htt...	htt...	http://...	http:...	SNP	position	Position	1..1	1..1	6428	false	<http://pbr.wur.nl/VCF/position>
htt...	htt...	http://...	http:...	SNP	genotype	SNPGenotype	1..N	1..1	19284	false	<http://pbr.wur.nl/VCF/genotype>
htt...	htt...	http://...	http:...	SNP	allelySNP	string	1..N	X..N	7626	true	<http://pbr.wur.nl/VCF/allelySNP>
htt...	htt...	http://...	http:...	SNP	refSNP	string	1..1	X..N	6428	true	<http://pbr.wur.nl/VCF/refSNP>

Figure 2: 'type link' table: Table displaying the 'type link' information.

The OWL exporter

The owl exporter automatically creates the following axioms/items

- Object properties
  - domain definitions
  - range definitions
- Data type properties
  - domain definitions
  - range definitions

- class subClassof restrictions
  - 'all values from'
  - cardinality (min, max or exact)
  - inverse cardinality(min, max or exact)

The parent to child subClassOf definitions are included if present in the original RDF resource, but are not automatically generated by RDF2Graph.
If a predicate links to an external reference a data type property is included which has the 'xsd:anyURI' type.
NOTE: If a property is found to be both an object property and a domain property it is exported as such. This is against the OWL specifications and so reasoners might not accept it or crash due to this. If this condition occurs it is an error in the resource upon which the structure recovery is executed.
NOTE: In case a base class has for a given predicate only integer values while a subclass has for the same predicate only strings the reasoner will mark the base class to be equal to none.
NOTE: Links defined as invalid are not exported in the OWL exporter.

The SHEX exporter

The SHEX standard is still in development, therefore it might happened that some of the recovered elements lack a complete or correct definition.

Overview of internal ontology

The following section gives a overview of the ontology used in the internal database of the structure recovery tool.
The structure/ontology is based on the concept of class properties and *type links*. So a class can have multiple class properties and each class property can have multiple type links. A *type link* is defined as a link joining a subject class type to an object class
or value data type, via a predicate. A *unique type link* is defined as an unique tuple: `{subject type, predicate, object (data) type}`. Therefore, each unique type link connects to only one Type. For example, if class *person* has as a predicate *age*, which is either an integer or string, then the class *person* has one class property that has two *unique type links* one with integer as type and one with string as type: `{:person, :age, xsd:integer}` and `{:person, :age, xsd:string}`

The ontology can be found in *doc\RDF2Graph.owl*
The uml diagram is shown in Figure 3 and can also be found in doc\RDF2Graph.dia.
The file doc\RDF2Graph.cys contains the cytoscape sesion that was used to generate the network representation from Figure 4.

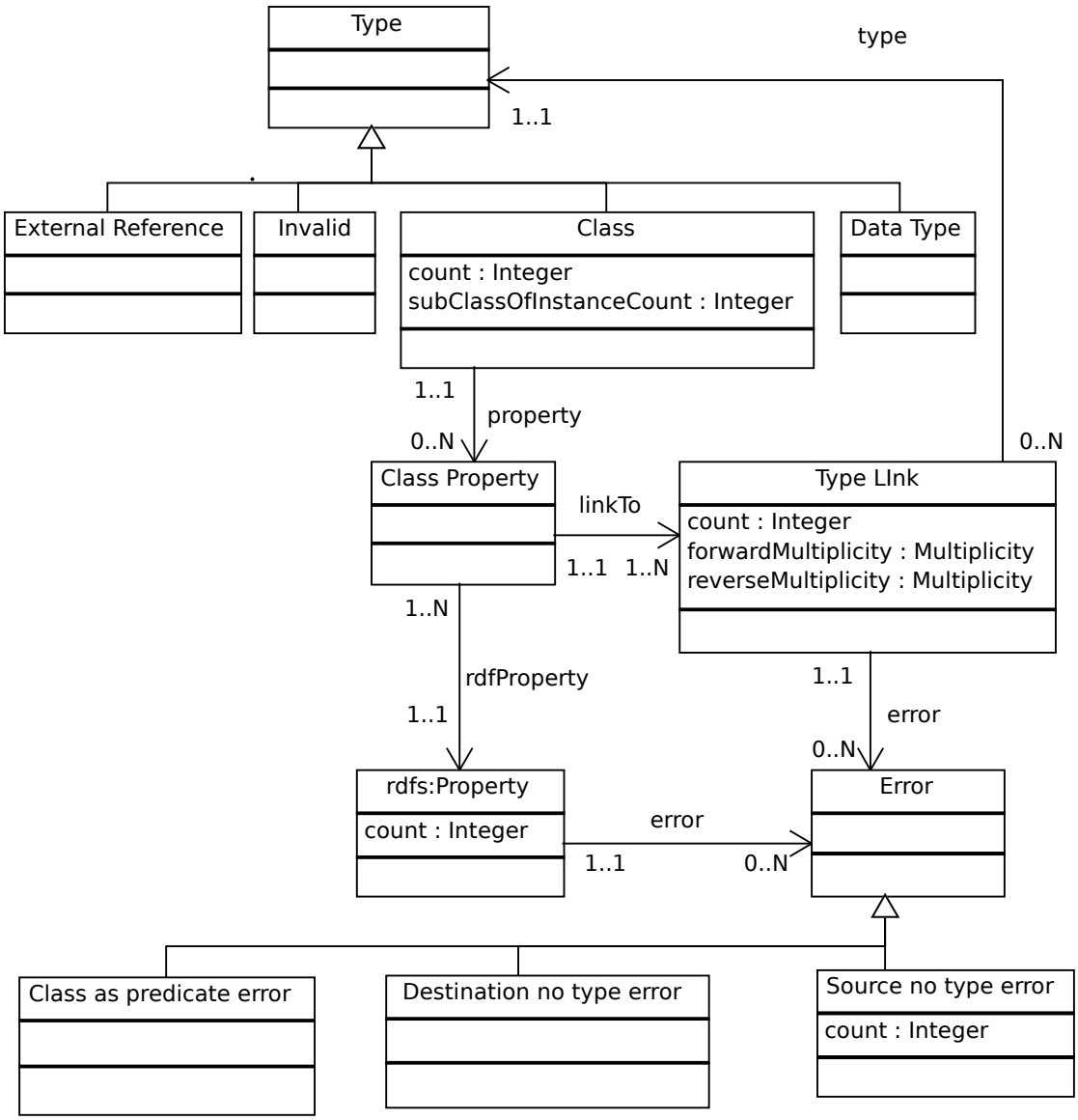


Figure 3: UML diagram: simplified uml diagram of the classes used in the RDF2Graph ontology

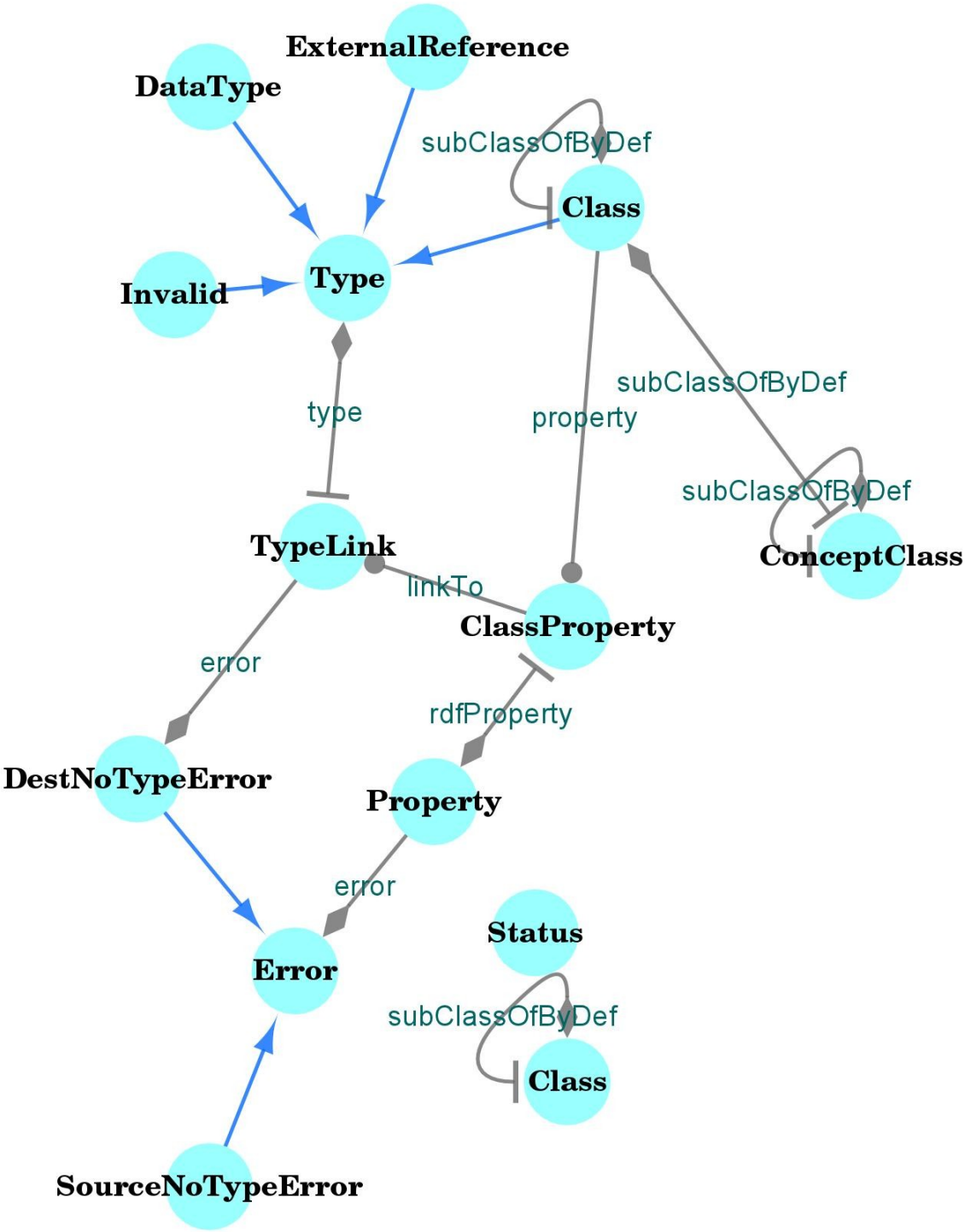


Figure 4: RDF2Graph result from the RDF2Graph internal datastructure

Now we proceed to a description of the key elements of this ontology.

Class

Properties

- *count*: the number of instances of the class.
- *subClassOfIntanceCount*: the number of instances of all the sub classes of the class.
- *property*: Each class has zero or more class properties.

NOTE: If limitations are included in the recovery step (to limit running time) *count* can be capped to, for example 100000. In this case the *subClassOfIntanceCount* will be a summation of possibly capped instance *counts*. For example it could be equal to 100000(capped) + 100000(capped) + 24234 + 10 = 224244.

Class property

Properties:

- *linkTo*: all *type links* of the class property.
- *rdfProperty*: the predicate of the class property.

Type Link

Properties:

- *count*: the number of triples that belong to this type link.
- *forwardMultiplicity*: the forward multiplicity of this link, which can be:
  - 1..1, each source instance has exactly one reference to the target (rs:Exactly-one)
  - 1..N, each source instance has one or more references to the target (rs:One-or-may)
  - 0..1, only some source instances have at most one reference to the target (rs:Zero-or-one)
  - 0..N, some source instances have one or even more than one reference to the target (rs:Zero-or-many)
- *reverseMultiplicity*: the reverse/inverse multiplicity of this link, which can be:
  - 1..1, each target instance has exactly one reverse reference to the source (rs:Exactly-one)
  - 1..N, each target instance has one or more reverse reference to the source (rs:One-or-may)
  - 0..1, only some target instances have at most one reverse reference to the source (rs:Zero-or-one)
  - 0..N, some target instances have one or even more than one reverse reference to the source (rs:Zero-or-many)
  - X..1, target instances have no more than one reverse reference to the source(RDF2Graph:x_1)
  - X..N, target instances can have more than one reverse references to the source(RDF2Graph:x_n)
- type: object type of the *type link*
-

NOTE: The current implementation has no support for forward or reverseMultiplicity on class properties. For example if a class *person* has always an age defined with a number this will have (1..1) forward multiplicity, but if this can be either an integer or string then the forward multiplicity of each *type link* will become 0..1.

Property:

Properties:

- *count*: total number of triples that contain this predicated

Type:

A type can be one of the following 4 types

- *externalref*: An object that has no defined type nor any properties. This means it is either a reference to an external database or a faulty link.
- *Invalid*: An object that has no defined type, but does have some properties. The type definition is missing.
- *Simple*: An object that is a data type, such as for example an integer or a string.
- *Class*: see above.

Error:

Properties:

- *rdfProperty*: property to which the errors applies

RDF2Graph also detect and reports some errors, which can be one of the following types.

- *Class as predicate error*: Reported when a class type is also used as predicate.
- ***Destination no type error*:** Reported when a link links to an object that has no type defined. This also presented as an invalid type.
- ***Source no type error:*** Reported when the source of a link has no type defined. An additional property is included that reports the number of links found with this condition for the specific RDF property.

NOTE: When these errors are occur it can be that RDF2Graph does not recovery the complete structure of the resource.

Detailed description structure recovery process

This section describes in detail the capabilities and functioning of the structure recovery tool.

Figure 5 provides an overview of the process to recover the structure and extract some useful statistics from a user specified database. During Steps 1 to 6 the information from the targeted RDF resource is collected and stored into a local RDF database. The optional step 7 simplifies the structure so that a neat graphical overview can be generated.

Each of the steps 1-6 involves a set of SPARQL queriesexecuted on the remote endpoint. Queries that contain some limitations to optimize the running time of RDF2Graph are marked with a *. However, these limitations might result in incomplete results.

The queries are provided in the **queries** folder of the RDF2Graph distribution. Within in this folder there are 2 sets of queries.

- **remote** folder queries for the default settings.
- remoteSubClassOf folder is for the `-treatSubClassOfAsIntanceOf` setting

The current queries are optimized for both the Jena and Stardog endpoints. They can be modified to tune and adapt RDF2Graph to your SPARQL endpoint and needs.

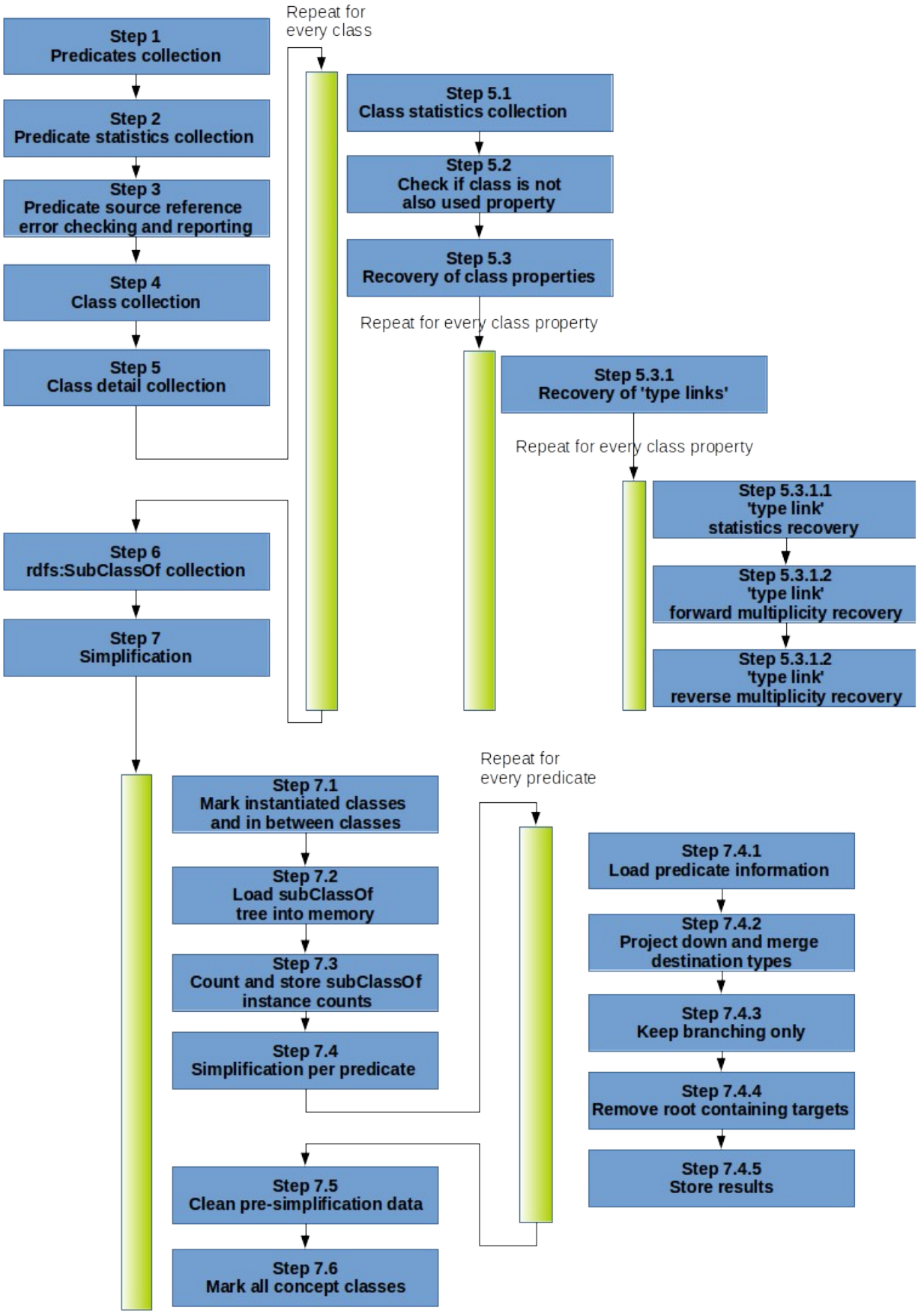


Figure 5: Overview of structrue recovery process RDF2Graph

Step 1: Predicates collection

All unique predicates in the database are recovered. Some predicates related to the RDF namespace are ignored. These are rdf:type, rdf:first, rdf:rest, rdfs:domain, rdfs:range, rdf:object, rdf:predicate, rdf:subject and rdf:value.
For each found predicate a property object (rdfs:Property) is generated.

Query: *getAllPredicates.txt*

Step 2: Predicate statistics collection

For each predicate the number of occurrences of the predicate are counted and the resulting count is stored (RDF2Graph:count) under the property object.

Query: **countPredicate.txt**

Step 3: Predicate source reference error checking and reporting

Each predicate is checked. During this check a search is performed to identify any subject referencing the predicate without a class type definition. If found, a “source has not type definition” error (RDF2Graph:SourceNoTypeError) is created and linked (RDF2Graph:error) to the predicate object. Furthermore the erroneous subjects are counted and the resulting number is stored (RDF2Graph:count) under the error object.

When such an error exists information in the overview will be missing. The tool does not try to 'infer' the missing type and skips any references for these subjects.

Query*: **testSourceTypesForPredicate.txt**
By default this query is limited so that only 10^7 triples are checked.

Step 4: Class collection

All classes with 1 or more defined instances are recovered. Classes without any instances are not considered. These could arise for example from an ontology like the Gene Ontology (GO). For each found class found class (RDF2Graph:Class) object is generated.

Query: **getAllClasses.txt**

Step 5: Class detail collection

The following steps are performed for each of the classes found in the previous step.

Step 5.1 Class statistics collection

The number of instances of the class are counted and the resulting count is stored (RDF2Graph:count) under the class object.

Query: **countNumberOfClassInstances.txt**

step 5.2 Check if class is also used as a property

If the class is also used as a predicate a class as predicate error object (RDF2Graph:ClassAsPredicateError) is generated and linked to the predicate object. Such error could produce an incompletely recovered structure.

Query: **checkClassIsNotPredicate.txt**

Step 5.3: Class property collection

All unique predicates referenced by the instances of the class are recovered. For each found predicate a class property object (RDF2Graph:ClassProperty) is generated and linked (RDF2Graph:rdfProperty) to the predicate object. Additionally, the class object is linked (RDF2Graph:property) to the class property object. Then the following steps are performed for each class property object.

This query can be either executed per class or executed as once depending on the `--useClassPredFindAtOnce` option.

Query*: getClassShapeProperties.txt (`--useClassPredFindAtOnce == false`)
Query: getClassAllShapeProperties.txt (`--useClassPredFindAtOnce == true`)

Step 5.3.1 'type link' *collection*

All unique value types referenced by the class property are recovered. There exist four groups of value typesi) A subject, which is an instance of any class type; (ii2) a literal value of any literal data type; (iii) a reference to an external resource, which is defined as reference to an IRI without any properties (RDF2Graph:externalref); (iv) an *'invalid'* subject, which has no class type defined, but has some properties (RDF2Graph:invalid). Each class type and/or literal data type is considered as a unique reference.
For each unique value type a *type link* object (RDF2Graph:TypeLink) is generated and linked(RDF2Graph:type) to the value type. The class property object is then linked (RDF2Graph:linkTo) to the *type link* object. The possible values are (1) a unique class type, (2) a unique literal data type, (3) a external reference (RDF2Graph:externalref) or (4) invalid reference (RDF2Graph:invalid). In the last case, when it is an invalid reference, an *destination has no type definition* error object (RDF2Graph:DestNoTypeError) is created and linked (RDF2Graph:error) to the predicate object.

Query*: **getClassPropertyDetails.txt**

If a invalid or external reference is found an additional check is performed.

Query: **checkClassPropertyExternalRef.txt**

Afterwards for each *type link* the following queries are executed.

Step 5.3.1.1. type link *statistics collection*

The occurrences of the *type link* are counted and the final number is stored (RDF2Graph:count) under the *type link* object.

Query*: **getShapePropertyCount_<suffix>.txt**
The suffix depends on 'type link' found.

- exref: used when the object is a external reference.
- ref: used when the object is an instance which has a class type definitions.
- simple: used when the object is a data type.

Step 5.3.1.2 type link *forward multiplicity collection*

The minimum and maximum multiplicities of the *type link* are separately recovered and encoded into one of the following multiplicity types:

- each source instance as exactly one reference to the target (rs:Exactly-one)
- each source instance one or more references to the target (rs:One-or-may)
- some source instances have at most one reference to the target (rs:Zero-or-one)
- some source instances have one or even more than one reference to the target (rs:Zero-or-many). The resulting encoded multiplicity is stored (RDF2Graph:forwardMultiplicity) under the *type link* object.

Query*: *getShapePropertyForwardMaxMultiplicity_<suffix>.txt* and
getShapePropertyForwardMinMultiplicity_<suffix>.txt

Step 5.3.1.3 type link *reverse multiplicity collection*

The same method as for the forward multiplicity is used interchanging the source and the target. However, the minimum multiplicity can only be determined for a unique class type and the maximum multiplicity can only be determined for a unique class type, external reference and the literal data type string. When an undetermined minimum and maximum multiplicities are encountered it is encoded into one of the following multiplicity types:

- target instances have no more than one reverse reference to the source(RDF2Graph:x_1),
- target instances can have more than one reverse references to the source(RDF2Graph:x_n)
- no reverse multiplicity can be determined (RDF2Graph:none).

The resulting encoded multiplicity is stored (RDF2Graph:reverseMultiplicity) under the *type link* object.

Query*: **getShapePropertyReverseMaxMultiplicity_<suffix>.txt** and
getShapePropertyReverseMinMultiplicity_ref.txt

Step 6 rdfs:SubClassOf collection

All rdfs:SubClassOf relationships are recovered and stored.

Query: **getAllSubClassOf.txt**

Step 7 (Optional) Simplification

Some RDF databases contain many subclasses, each of these subclasses can have instances that contain the same predicate as the parent class. If no simplification step is performed then for each subclass containing the same predicate a link will be generated. If many subclasses exists the network representation becomes an unreadable hairball.
To overcome this issue an optional simplification step might be performed that finds both the common source class and the common target class for a set of *type links* that share the same predicate. In the following figure, three examples of the type of simplifications performed in this step are shown.

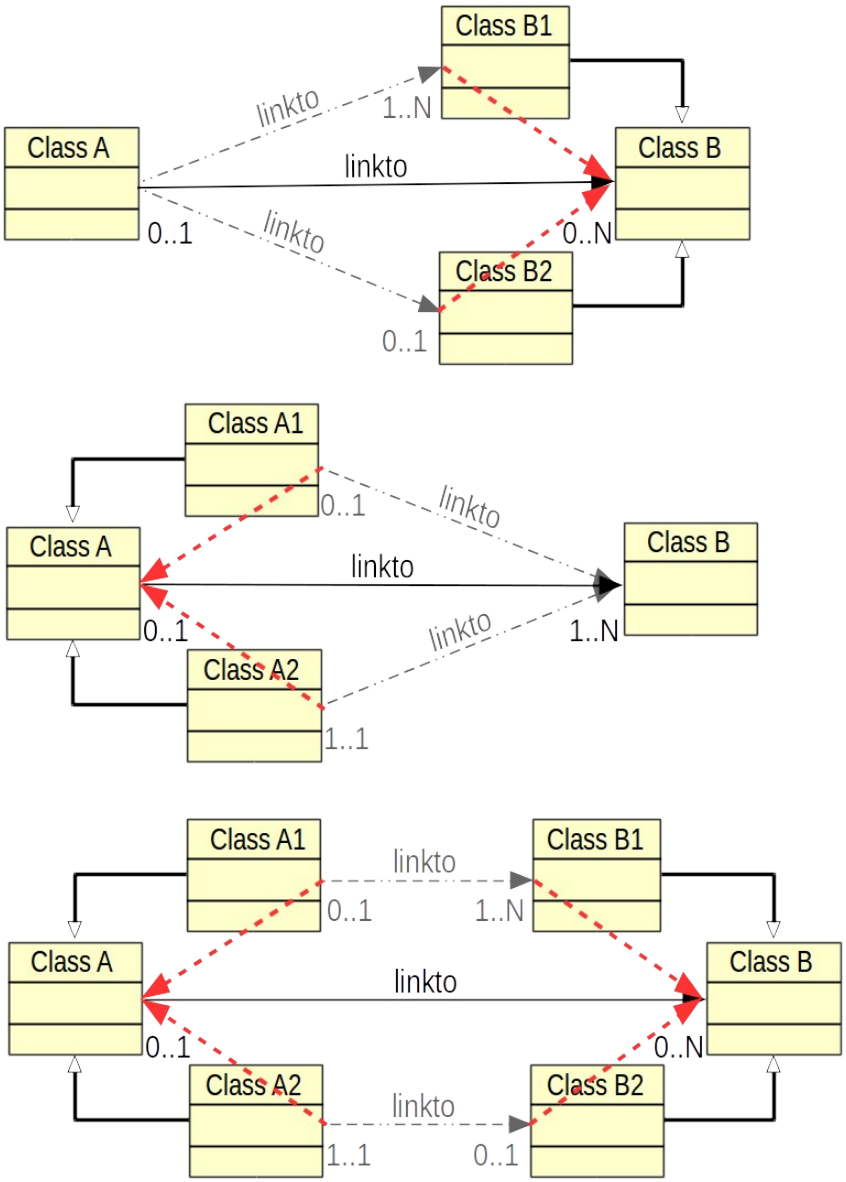


Figure 6: A. The red connections indicate the resulting connections after a simplification is performed in which an common ancestor for the source is found. B. The red connections indicate the resulting connection after a simplification is performed in which for both targets a common ancestor is found. C. The red connections indicate the resulting connections after a simplification is performed in which common ancestors are found for both the sources and the targets.

This simplification proceeds as follows.

Step 7.1 Mark instantiated classes

All classes that have either some instances or a child class that has some instances are marked as having been instantiated. A class is marked by making it an instance of RDF2Graph:Class.

Step 7.2 Load subClassOf tree into memory

All marked classes and associated subclassOf links are loaded into a memory based directed graph. This memory loaded class structure can then be used to load additional information and find the common parent of two classes.

Step 7.3 Count and store subClassOf instance counts

For each class the instances of all its subclasses are counted and the resulting count is stored (RDF2Graph:subClassOfIntanceCount) under the class object.

Step 7.4 Simplification per predicate

The simplification process is executed per retrieved predicate. For each predicate following steps are performed**.**

Step 7.4.1 Load predicate information

The information for all *'type links* objects belonging to the currently processed predicate are loaded into the tree. For each *type link* object a reference object is added to the source class in the tree that points towards the target class in the tree. The associated forward multiplicity, reverse multiplicity and count are loaded into the reference object.

The algorithm in the following steps will merge these reference objects. Upon merging two reference objects, the counts are added to each other, the forward multiplicities are merged and the reverse multiplicities are merged. After merging two multiplicities the new minimum per instance reference count will be equal to the lowest per instance reference count of the merged multiplicities and the new maximum per instance reference count will be equal to highest per instance reference count of the merged multiplicities.

Step 7.4.2 Project down and merge destination types

This step ensures that all source references contained within the child classes get copied into their parents classes after which the references gets merged if the target classes of the references share a common parent.

This step is based on a recursion depth first process over the sub class of tree in which the following steps are performed per class object. All references in the child classes are copied and added to the references of the parent class. Afterwards the references in the parent class are merged with each other if the referenced target classes have a common parent class. The resulting merged reference will point to the first found common parent of the target classes. If the common parent is owl:Thing no merging is performed.

Step 7.4.3 Keep branching only

During the previous step no shared references are also copied to the parent classes. This step ensures that this unwanted behavior is avoided and any reference not shared by multiple children gets removed again.

This step is based on a recursion breadth first process over the sub class of tree in which the following is performed for each class in the tree.

- For each reference (varC) to a type in the currently processed class (varA)
  - Count the number of direct subclasses (varB) of the currently processed class (varA) that fullfills the following condition:

        - The direct subclass(varB) has at least one reference targeting a class that is either equal or a subclass of the class referenced by the currently processed reference (varC).
    - If the count is equal to one then remove the currently processed reference(varC) from the currently processed class(varA).

Step 7.4.4 Remove root containing targets

Step 7.4.2 does not remove the source references that were merged in one of the parent classes. This step cleans this merged references.

This steps is based on a recursion depth first process over the sub class of tree in which all the references that target the currently process class are removed if they fulfill the following condition: Any reference for which the target class is already referenced by the currently processed class or parent class of the currently processed class.

Step 7.4.5 Store results

New class property and *type links* are created for the newly calculated references. This includes the merged reference counts, forward and reverse multiplicities. Each newly created class property objects is marked such that they can be distinguished from the old ones.
At last the sub class of tree gets cleaned from the memory so that the next predicate can be processed.

Step 7.5 Clean pre-simplification data

All the class property and associated *type links* inserted before the simplification step are removed and the marks from the new class property objects are removed.

Step 7.6 Mark all concept classes

All the classes that have any instances or that lack a sub class that has some instances are marked as RDF2Graph:ConceptClass.

Step 8 (Optional) Clean OWL classes (optional)

Optionally any information about the OWL ontology related classes is removed. The owl:Class class is ignored.

Modifying RDF2Graph

In the section explaining the different steps to recover the structure, a detailed list of the files containing the queries has been included. These can be customized to adapt RDF2Graph to different SPARQL endpoint and needs.

Known issues

The simplification process might miscalculate the minimum cardinality. This happens for classes that have multiple subclasses and among these subclasses some have predicates with a minimum cardinality of 1 while other subclasses do not have the predicate at all. In these cases the minimum cardinality is 0, however 1 is kept.