SWI-Prolog/XPCE Semantic Web Library

Jan Wielemaker
SWI,
University of Amsterdam
The Netherlands
E-mail: jan@swi.psy.uva.nl

June 23, 2003

Abstract

This document describes a library for dealing with standards from the W3C standard for the *Semantic Web*. Like the standards themselves (RDF, RDFS and OWL) this infrastructure is modular. It consists of Prolog packages for reading, querying and storing semantic web documents as well as XPCE libraries that provide visualisation and editing. The Prolog libraries can be used without the XPCE GUI modules. The library can handle upto about 2 million *RDF triples* on current commonly used hardware (256MB memory, Pentium 1.5Ghz).

Contents

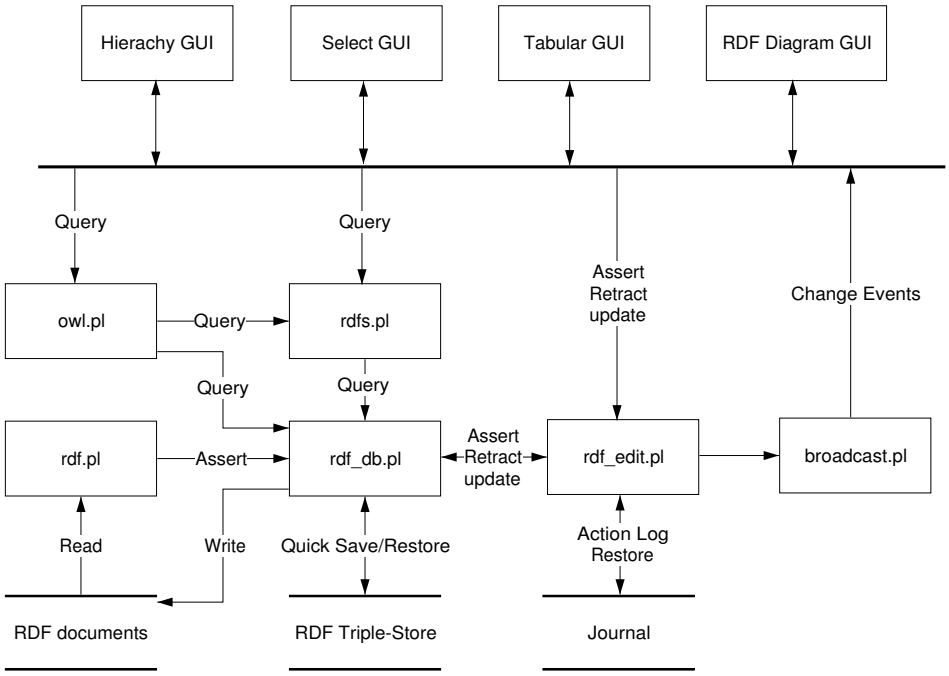

Figure 1: Modules for the Semantic Web library

1 Introduction

SWI-Prolog has started support for web-documents with the development of a small and fast SGML/XML parser, followed by an RDF parser (early 2000). With the `semweb` library we provide more high level support for manipulating semantic web documents. The semantic web is a likely point of orientation for knowledge representation in the future, making a library designed in its spirit promising.

2 Modules

Central to this library is the module `rdf_db.pl`, providing storage and basic querying for RDF triples. This triple store is filled using the RDF parser realised by `rdf.pl`. The storage module can quickly save and load (partial) databases. The modules `rdfs.pl` and `owl.pl` add querying in terms of the more powerful RDFS and OWL languages. Module `rdf_edit.pl` adds editing, undo, journaling and change-forwarding. Finally, a variety of XPCE modules visualise and edit the database. Figure figure 1 summarised the modular design.

3 Module rdf_db

The central module is called `rdf_db`. It provides storage and indexed querying of RDF triples. Triples are stored as a quintuple. The first three elements denote the RDF triple. *File* and *Line* provide information about the origin of the triple.

{*Subject Predicate Object File Line*}

The actual storage is provided by the *foreign language (C)* module `rdf_db.c`. Using a dedicated C-based implementation we can reduced memory usage and improve indexing capabilities.[1] Currently the following indexing is provided.

- Any of the 3 fields of the triple
- *Subject* + *Predicate* and *Predicate* + *Object*
- *Predicates* are indexed on the *highest property*. In other words, if predicates are related through `subPropertyOf` predicates indexing happens on the most abstract predicate. This makes calls to `rdf_has/4` very efficient.
- Literal *Objects* are indexed case-insensitive to make case-insensitive queries fully indexed. See `rdf/3`.

3.1 Query the RDF database

rdf(*?Subject, ?Predicate, ?Object***)**

Elementary query for triples. *Subject* and *Predicate* are atoms representing the fully qualified URL of the resource. *Object* is either an atom representing a resource or `literal`(*Text*) if the object is a literal value.[2] If a value of the form *NameSpaceID* : *NameSpaceID* is provided it is expanded to a ground atom using `expand_goal/2`. This implies you can use this construct in compiled code without paying a preformance penalty. See also section 3.4. For querying purposes, *Object* can be of the form `literal`(+*Query*, -*Value*), where *Query* is one of

exact(+*Text***)**

Perform exact, but case-insensitive match. This query is fully indexed.

substring(+*Text***)**

Match any literal that contains *Text* as a case-insensitive substring. The query is not indexed on *Object*.

word(+*Text***)**

Match any literal that contains *Text* delimited by a non alpha-numeric character, the start or end of the string. The query is not indexed on *Object*.

prefix(+*Text***)**

Match any literal that starts with *Text*. This call is intended for *completion*. The query is not indexed on *Object*.

rdf(*?Subject, ?Predicate, ?Object, ?Source***)**

As `rdf/3` but in addition return the source-location of the triple. The source is either a plain atom or a term of the format *Atom* : *Atom* where *Atom* is intended to be used as filename or URL and *Integer* for representing the line-number.

rdf_has(*?Subject, ?Predicate, ?Object, -TriplePred***)**

This query exploits the RDFS `subPropertyOf` relation. It returns any triple whose stored

[1] The orginal implementation was in Prolog. This version was implemented in 3 hours, where the C-based implementation costed a full week. The C-based implementation requires about half the memory and provides about twice the performance.

[2] The current implementation has no provisions for XML-Schema typed literals.

predicate equals *Predicate* or can reach this by following the recursive *subPropertyOf* relation. The actual stored predicate is returned in *TriplePred*. The example below gets all subclasses of an RDFS (or OWL) class, even if the relation used is not `rdfs:subClassOf`, but a user-defined sub-property thereof.[3]

```
subclasses(Class, SubClasses) :-
        findall(S, rdf_has(S, rdfs:subClassOf, Class), SubClasses).
```

rdf_has(*?Subject, ?Predicate, ?Object***)**
Same as `rdf_has`(*Subject, Predicate, Object*, _).

rdf_reachable(*?Subject, +Predicate, ?Object***)**
Is true if *Object* can be reached from *Subject* following the transitive predicate *Predicate* or a sub-property thereof. When used with either *Subject* or *Object* unbound, it first returns the origin, followed by the reachable nodes in breath-first search-order. It never generates the same node twice and is robust against cycles in the transitive relation. With all arguments instantiated it succeeds deterministically of the relation if a path can be found from *Subject* to *Object*. Searching starts at *Subject*, assuming the branching factor is normally lower. A call with both *Subject* and *Object* unbound raises an instantiation error. The following example generates all subclasses of `rdfs:Resource`:

```
?- rdf_reachable(X, rdfs:subClassOf, rdfs:'Resource').

X = 'http://www.w3.org/2000/01/rdf-schema#Resource' ;

X = 'http://www.w3.org/2000/01/rdf-schema#Class' ;

X = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#Property' ;

...
```

rdf_subject(*?Subject***)**
Enumerate resources appearing as a subject in a triple. The main reason for this predicate is to generate the known subjects *without duplicates* as one gets using `rdf`(*Subject*, _, _).

3.2 Modifying the database

As depicted in figure 1, there are two levels of modification. The `rdf_db` module simply modifies, where the `rdf_edit` library provides transactions and undo on top of this. Applications that wish to use the `rdf_edit` layer must *never* use the predicates from this section directly.

rdf_assert(*+Subject, +Predicate, +Object***)**
Assert a new triple into the database. This is equivalent to `rdf_assert/4` using *SourceRef* `user`. *Subject* and *Predicate* are resources. *Object* is either a resource or a term `literal`(*Value*). All arguments are subject to name-space expansion (see section 3.4).

[3] This predicate realises semantics defined in RDF-Schema rather than RDF. It is part of the `rdf_db` module because the indexing of this module incorporates the `rdfs:subClassOf` predicate.

rdf_assert(+*Subject, +Predicate, +Object, +SourceRef***)**
As `rdf_assert/3`, adding *SourceRef* to specify the orgin of the triple. *SourceRef* is either an atom or a term of the format *Atom*:*Int* where *Atom* normally refers to a filename and *Int* to the line-number where the description starts.

rdf_retractall(?*Subject, ?Predicate, ?Object***)**
Removes all matching triples from the database. Previous Prolog implementations also provided a backtracking `rdf_retract/3`, but this proved to be rarely used and could always be replaced with `rdf_retractall/3`. As `rdf_retractall/4` using an unbound *SourceRef*.

rdf_retractall(?*Subject, ?Predicate, ?Object, ?SourceRef***)**
As `rdf_retractall/4`, also matching on the *SourceRef*. This is particulary useful to update all triples coming from a loaded file.

rdf_update(+*Subject, +Predicate, +Object, +Action***)**
Replaces one of the three fields on the matching triples depending on *Action*:

subject(*Resource***)**
Changes the first field of the triple.

predicate(*Resource***)**
Changes the second field of the triple.

object(*Object***)**
Changes the last field of the triple to the given resource or `literal`(*Value*).

3.3 Loading and saving to file

The `rdf_db` module can read and write RDF-XML for import and export as well as a binary format built for quick load and save described in section 3.3.2. Here are the predicates for portable RDF load and save.

rdf_load(+*In***)**
Load triples from *In*, which is either a stream opened for reading or an atom specifying a filename. This predicate calls `process_rdf/3` to read the source one description at a time, avoiding limits to the size of the input. If *In* is a file, `rdf_load/1` provides for caching the results for quick-load using `rdf_load_db/1` described below. Caching is activated by creating a directory `.cache` (or `_cache` on Windows) in the directory holding the `.rdf` files. Cached RDF files are loaded at approx. 25 times the speed of RDF-XML files.

rdf_save(+*File***)**
Save all known triples to the given *File*.

rdf_save(+*File, +FileRef***)**
Save all triples whose file-part of their *SourceRef* matches *FileRef* to the given *File*. Saving arbitrary selections is possible using predicates from section 3.3.1.

rdf_source(?*File***)**
Test or enumerate the files loaded using `rdf_load/1`.

rdf_make
Re-load all RDF sourcefiles (see `rdf_source/1`) that have changed since they were loaded the last time. This implies all triples that originate from the file are removed and the file is re-loaded. If the file is cached a new cache-file is written. Please note that the new triples are added at the end of the database, possibly changing the order of (conflicting) triples.

3.3.1 Partial save

Sometimes it is necessary to make more arbitrary selections of material to be saved or exchange RDF descriptions over an open network link. The predicates in this section provide for this.

rdf_save_header(+*Stream, ?FileRef*)
Save an RDF header, with the XML header, `DOCTYPE`, `ENTITY` and opening the `rdf:RDF` element with appropriate namespace declarations. It uses the primitives from section 3.4 to generate the required namespaces and desired short-name.

rdf_save_footer(+*Stream*)
Close the work opened with `rdf_save_header/2`.

rdf_save_subject(+*Stream, +Subject, +FileRef*)
Save everything known about *Subject* that matches *FileRef*. Using an variable for *FileRef* saves all triples with *Subject*.

3.3.2 Fast loading and saving

Loading and saving RDF format is relatively slow. For this reason we designed a binary format that is more compact, avoids the complications of the RDF parser and avoids repetitive lookup of (URL) identifiers. Especially the speed improvement of about 25 times is worth-while when loading large databases. These predicates are used for caching by `rdf_load/1` under certain conditions.

rdf_save_db(+*File*)
Save all known triples into *File*. The saved version includes the *SourceRef* information.

rdf_save_db(+*File, +FileRef*)
Save all triples with *SourceRef FileRef*, regardless of the line-number. For example, using `user` all information added using `rdf_assert/3` is stored in the database.

rdf_load_db(+*File*)
Load triples from *File*.

3.4 Namespace Handling

Prolog code often contains references to constant resources in a known XML namespace. For example, `http://www.w3.org/2000/01/rdf-schema#Class` refers to the most general notion of a class. Readability and maintability concerns require for abstraction here. The dynamic and multifile predicate rdf_db:ns/2 maintains a mapping between short meaningful names and namespace locations very much like the XML `xmlns` construct. The initial mapping contains the namespaces required for the semantic web languages themselves:

```
ns(rdf,  'http://www.w3.org/1999/02/22-rdf-syntax-ns#').
ns(rdfs, 'http://www.w3.org/2000/01/rdf-schema#').
ns(owl,  'http://www.w3.org/2002/7/owl#').
ns(xsd,  'http://www.w3.org/2000/10/XMLSchema#').
ns(dc,   'http://purl.org/dc/elements/1.1/').
ns(eor,  'http://dublincore.org/2000/03/13/eor#').
```

All predicates for the semweb libraries use `goal_expansion/2` rules to make the SWI-Prolog compiler rewrite terms of the form *Id* : *Id* into the fully qualified URL. In addition, the following predicates are supplied:

rdf_equal(*Resource1, Resource2***)**
Defined as *Resource1, Resource2* = *Resource1, Resource2* As this predicate is subject to goal-expansion it can be used to obtain or test global URL values to readable values. The following goal unifies *X* with `http://www.w3.org/2000/01/rdf-schema#Class` without more runtime overhead than normal Prolog unification.

```
rdf_equal(rdfs:'Class', X)
```

rdf_register_ns(*+Alias, +URL***)**
Register *Alias* as a shorthand for *URL*. Note that the registration must be done before loading any files using them as namespace aliases are handled at compiletime through `goal_expansion/2`.

rdf_global_id(*?Alias:Local, ?Global***)**
Runtime translation between *Alias* and *Local* and a *Global* URL. Expansion is normally done at compiletime. This predicate is often used to turn a global URL into a more readable term.

rdf_global_term(*+Term0, -Term***)**
Expands all *Alias*:*Local* in *Term0* and return the result in *Term*. Use infrequently for runtime expansion of namespace identifiers.

rdf_split_url(*?Base, ?Local, ?URL***)**
Split a URL into a prefix and local part if used in mode -,-,+ or simply behave as `atom_concat/3` in other modes. The *URL* is split on the last # or / character.

3.5 Miscellaneous predicates

This section describes the remaining predicates of the `rdf_db` module.

rdf_node(*-Id***)**
Generate a unique reference. The returned atom is guaranteed not to occur in the current database in any field of any triple.

rdf_source_location(*+Subject, -SourceRef***)**
Return the source-location as *File*:*Line* of the first triple that is about *Subject*.

rdf_generation(*-Generation***)**
Returns the *Generation* of the database. Each modification to the database increments the generation. It can be used to check the validity of cached results deduced from the database.

rdf_statistics(*?Statistics***)**
Report statistics collected by the `rdf_db` module. Defined values for *Statistics* are:

lookup(*?Index, -Count***)**
Number of lookups using a pattern of instantiated fields. *Index* is a term `rdf`(*S,P,O*), where *S*, *P* and *O* are either `+` or `-`. For example `rdf`(+,+,-) returns the lookups with subject and predicate specified and object unbound.

properties(-*Count***)**
Number of unique values for the second field of the triple set.

sources(-*Count***)**
Number of files loaded through `rdf_load/1`.

subjects(-*Count***)**
Number of unique values for the first field of the triple set.

triples(-*Count***)**
Total number of triples in the database.

rdf_match_label(*+Method, +Search, +Atom***)**
True if *Search* matches *Atom* as defined by *Method*. All matching is performed case-insensitive. Defines methods are:

exact
Perform exact, but case-insensitive match.

substring
Search is a sub-string of *Text*.

word
Search appears as a whole-word in *Text*.

prefix
Text start with *Search*.

3.6 Issues with rdf_db

This RDF low-level module has been created after two year experimenting with a plain Prolog based module and a brief evaluation of a second generation pure Prolog implementation. The was to be able to handle upto aboud 2 million triples on standard (notebook) hardware and deal efficiently with `subPropertyOf` which was identified as a crucial feature of RDFS to realise fusion of different data-sets.

The following issues are identified and not solved in suitable manner.

Logical update as provided by Prolog means that active queries are not affected by subsequent modification of the database. The current C-based implementation adheres the *immediate* update model, mainly because the current foreign language interface does not provide the required information to realise logical updates in C.

Property hierarchy The system currently cannot deal with properties that have multiple parents if not all parents ultimately have the save root. I.e. there must be a single root property for each property hierarchy. Although the design accomodates for this case, it has not yet been implemented.

`subPropertyOf` **of** `subPropertyOf` is not supported.

4 Module rdfs

The `rdfs` library adds interpretation of the triple store in terms of concepts from RDF-Schema (RDFS).

4.1 Hierarchy and class-individual relations

The predicates in this section explore the `rdfs:subPropertyOf`, `rdfs:subClassOf` and `rdf:type` relations. Note that the most fundamental of these, `rdfs:subPropertyOf`, is also used by `rdf_has/[3,4]`.

rdfs_subproperty_of(*?SubProperty, ?Property***)**
: True if *SubProperty* is equal to *Property* or *Property* can be reached from *SubProperty* following the `rdfs:subPropertyOf` relation. It can be used to test as well as generate sub-properties or super-properties. Note that the commonly used semantics of this predicate is wired into `rdf_has/[3,4]`.[4,5]

rdfs_subclass_of(*?SubClass, ?Class***)**
: True if *SubClass* is equal to *Class* or *Class* can be reached from *SubClass* following the `rdfs:subClassOf` relation. It can be used to test as well as generate sub-classes or super-classes.[6].

rdfs_class_property(*+Class, ?Property***)**
: True if the domain of *Property* includes *Class*. Used to generate all properties that apply to a class.

rdfs_individual_of(*?Resource, ?Class***)**
: True if *Resource* is an indivisual of *Class*. This implies *Resource* has an `rdf:type` property that refers to *Class* or a sub-class thereof. Can be used to test, generate classes *Resource* belongs to or generate individuals described by *Class*.

4.2 Collections and Containers

The RDF construct `rdf:parseType=Collection` constructs a list using the `rdf:first` and `rdf:next` relations.

rdfs_member(*?Resource, +Set***)**
: Test or generate the members of *Set*. *Set* is either an individual of `rdf:List` or `rdf:Container`.

rdfs_list_to_prolog_list(*+Set, -List***)**
: Convert *Set*, which must be an individual of `rdf:List` into a Prolog list of objects.

[4] BUG: The current implementation cannot deal with cycles

[5] BUG: The current implementation cannot deal with predicates that are an `rdfs:subPropertyOf` of `rdfs:subPropertyOf`, such as `owl:samePropertyAs`.

[6] BUG: The current implementation cannot deal with cycles

4.3 Labels and textual search

Textual search is partly handled by the predicates from the `rdf_db` module and its underlying C-library. For example, literal objects are hashed case-insensitive to speed up the commonly used case-insensitive search.

rdfs_label(*?Resource, ?Label***)**
Extract the label from *Resource* or generate all resources with the given *Label*. The label is either associated using a sub-property of `rdfs:label` or it is extracted from the URL using `rdf_split_url/3`.

rdfs_ns_label(*?Resource, ?Label***)**
Similar to `rdfs_label/2`, but prefixes the result using the declared namespace alias (see section 3.4) to facilitate user-friendly labels in applications using multiple namespaces that may lead to confusion.

rdfs_find(*+String, +Description, +Properties, +Method, -Subject***)**
Find (on backtracking) *Subject*s that satisfy a search specification for textual attributes. *String* is the string searched for. *Description* is an OWL description (see section **??**) specifying candidate resources. *Properties* is a list of properties to search for literal objects where *rdfs:label* is replaced by a call to `rdfs_label/2` and finally, *Method* defines the textual matching algorithm. All textual mapping is performed case-insensitive. The matching-methods are described with `rdf_match_label/3`.

5 Module rdf_edit

The module `rdf_edit.pl` is a layer than encasulates the modification predicates from section 3.2 for use from a (graphical) editor of the triple store. It adds the following features:

- *Transaction management*
  Modifications are grouped into *transactions* to safeguard the system from failing operations as well as provide meaningfull chunks for undo and journalling.
- *Undo*
  Undo and redo-transactions using a single mechanism to support user-friendly editing.
- *Journalling*
  Record all actions to support analysis, versioning, crash-recovery and an alternative to saving.

5.1 Transaction management

Transactions group low-level modification actions together.

rdfe_transaction(*:Goal***)**
Run *Goal*, recording all modifications to the triple store made through section 5.2. Execution is performed as in `once/1`. If *Goal* succeeds the changes are committed. If *Goal* fails or throws an exception the changes are reverted.

Transactions may be nested. A failing nested transaction only reverts the actions performed inside the nested transaction. If the outer transaction succeeds it is committed normally. Contrary, if the outer transaction fails, comitted nested transactions are reverted as well.

A successful outer transaction ('level-0') may be undone using `rdfe_undo/0`.

5.2 Encapsulated predicates

The following predicates encapsulate predicates from the `rdf_db` module that modify the triple store. These predicates can only be called when inside a *transaction*. See `rdfe_transaction/1`.

rdfe_assert(+*Subject, +Predicate, +Object***)**
Encapsulates `rdf_assert/3`.

rdfe_retractall(*?Subject, ?Predicate, ?Object***)**
Encapsulates `rdf_retractall/3`.

rdfe_update(+*Subject, +Predicate, +Object, +Action***)**
Encapsulates `rdf_update/4`.

rdf_load(+*In***)**
Encapsulates `rdf_load/1`.

5.3 High-level modification predicates

This section describes a (yet very incomplete) set of more high-level operations one would like to be able to perform. Eventually this set may include operations based on RDFS and OWL.

rdfe_delete(+*Resource***)**
Delete all traces of *resource*. This implies all triples where *Resource* appears as *subject*, *predicate* or *object*. This predicate starts a transation.

5.4 Undo

Undo aims at user-level undo operations from a (graphical) editor.

rdfe_undo
Revert the last outermost ('level 0') transaction (see `rdfe_transaction/1`). Successive calls go further back in history. Fails if there is no more undo information.

rdfe_redo
Revert the last `rdfe_undo/0`. Successive calls revert more `rdfe_undo/0` operations. Fails if there is no more redo information.

rdfe_can_undo
Test if there is another transaction that can be reverted. Used for activating menus in a graphical environment.

rdfe_can_redo
Test if there is another undo that can be reverted. Used for activating menus in a graphical environment.

5.5 Journalling

Optionally, every action through this module is immediately send to a *journal-file*. The journal provides a full log of all actions with a time-stamp that may be used for inspection of behaviour, version management, crash-recovery or an alternative to regular save operations.

rdfe_open_journal(+*File*, +*Mode***)**
: Open a existing or new journal. If *Mode* equala `append` and *File* exists, the journal is first replayed. See `rdfe_replay_journal/1`. If *Mode* is `write` the journal is truncated if it exists.

rdfe_close_journal
: Close the currently open journal.

rdfe_current_journal(-*Path***)**
: Test whether there is a journal and to which file the actions are journalled.

rdfe_replay_journal(+*File***)**
: Read a jorunal, replaying all actions in it. To do so, the system reads the journal a transaction at a time. If the transaction is closed with a *commit* it executes the actions inside the journal. If it is closed with a *rollback* or not closed at all due to a crash the actions inside the journal are discarded. Using this predicate only makes sense to inspect the state at the end of a journal without modifying the journal. Normally a journal is replayed using the `append` mode of `rdfe_open_journal/2`.

5.6 Broadcasting change events

To realise a modular graphical interface for editing the triple store, the system must use some sort of *event* mechanism. This is implemented by the XPCE library `broadcast` which is described in the XPCE User Guide. In this section we describe the terms brodcasted by the library.

rdf_transaction(+*Id***)**
: A 'level-0' transaction has been committed. The system passes the identifier of the transaction in *Id*. In the current implementation there is no way to find out what happened inside the transaction. This is likely to change in time.

  If a transaction is reverted due to failure or exception *no* event is broadcasted. The initiating GUI element is supposed to handle this possibility itself and other components are not affected as the triple store is not changed.

rdf_undo(+*Type*, +*Id***)**
: This event is broadcasted after an `rdfe_undo/0` or `rdfe_redo/0`. *Type* is one of `undo` or `redo` and *Id* identifies the transaction as above.

Index