SWI-Prolog and the Web

JAN WIELEMAKER

Human-Computer Studies laboratory
University of Amsterdam
Matrix I
Kruislaan 419
1098 VA, Amsterdam
The Netherlands
(*e-mail:* wielemak@science.uva.nl)

ZHISHENG HUANG, LOURENS VAN DER MEIJ

Computer Science Department
Vrije University Amsterdam
De Boelelaan
1081 HV, Amsterdam
The Netherlands
(*e-mail:* huang,lourens@cs.vu.nl)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Prolog is an excellent tool for representing and manipulating data, written in formal languages as well as natural language. Its safe semantics and automatic memory management make it an ideal tool for programming robust web services.

We propose a server architecture where Prolog communicates to other components in the web-server using HTTP. By turning Prolog into an HTTP server and client we reach at a more flexible server organization as well as easier deployment and debugging compared to embedding Prolog in an existing server network.

This paper presents the enabling extensions to the Prolog language such as Unicode and multi-threading support as well as the enabling libraries for handling web documents and protocols. The described libraries support a wide range of web applications ranging from HTML and XML documents to Semantic Web RDF processing.

The benefits using Prolog for web-related tasks is illustrated using three case studies.

KEYWORDS: Prolog, HTTP, HTML, XML, RDF

1 Introduction

The Web is an exiting place offering new opportunities to AI techniques and Logic Programming. Information extraction from the Web, reasoning inside web servers and the Semantic Web are just a few examples.

There are two views on deploying Prolog for web-related tasks. In the most popular view, Prolog acts as an embedded component in a general web processing environment such as Tomcat or Apache. In this role it generally does reasoning tasks such as searching or configuration within constraints. Alternatively Prolog

itself can act as a stand-alone HTTP server as also proposed by ECLiPSe (Leth et al. 1996). In this view it is a component that can be part of any of the layers of the popular three-tier architecture for web-application, generally exchanging XML if used as part of the backend or middleware services or HTML when used in the presentation layer.

The latter view is in our vision more attractive. Using HTTP and XML, the service is cleanly isolated using standard protocols rather than proprietary ad-hoc embedded communication. Running as a stand-alone application, the attractive interactive development nature of Prolog can be maintained much more easily than embedded in a C, C++, Java or C# application. Automatic testing of the Prolog components can be done using any web-oriented test framework. **TBD: example** Talking HTTP, Prolog can easily play role in any part of the service architecture or even realise the entire service in one or more Prolog processes.

TBD: project names and references The libraries and Prolog extensions to provide this support have been realised in the context of various research projects over a long period. The SGML, HTML and XML parser was realised when document fragmentation and classification was our main topic. The HTTP libraries have been realised in the context of agent-based programming. The RDF read/write and storage libraries result from projects on ontology-based annotation and search.

This paper is organized as follows. Section 2 to Sect. 3.3 describe reading, writing and representation of web-related documents. Section 4 describes the architecture of the extensible HTTP client and server libraries. Section 5 describes core extensions to the Prolog language that facilitate use in web-services. Section 6 to Sect. 8 describe three case studies exploiting Prolog as Sematic Web query service, component in a DL reasoning service and smart portal for a Sematic Wed database.

2 Parsing and representing XML and HTML documents

The core of the Web is formed by document standards and exchange protocols. Protocols are described in Sect. 4. We consider two types of documents: tree-structured documents transferred as SGML or XML applications and graph-structured documents forming the heart of the Semantic Web. Graph structured documents are described in Sect. 3.1. Here we concentrate on markup languages.

HTML, an SGML application, is the most commonly used document format on the web. HTML represents documents as a tree using a fixed set of *elements* (tags), where the SGML DTD (Document Type Declaration) puts constraints on how elements can be nested. Each node in the hierarchy has a name (the element-name), a set of name-value pairs known as its attributes and a *content*, a sequence of sub-elements and text (data). Initially HTML elements were oriented on *semantics*, using tags like `author`. Later extensions concentrated more on *layout* to facilitate WYSIWYG editors, introducing elements such as `font`.

XML is a rationalisation of SGML using the same tree-model, but removing many seldomly used features as well as abbreviations that were introduced in SGML to make the markup easier to type and read by humans. Examples are the *short-tag* notation (`<b/bold-text/`), the *short-ref* notation where normal characters can

⟨*document*⟩	::=	list-of ⟨*content*⟩
⟨*content*⟩	::=	⟨*element*⟩
	|	⟨*pi*⟩
	|	⟨*cdata*⟩
	|	⟨*sdata*⟩
	|	⟨*ndata*⟩
⟨*element*⟩	::=	element(⟨*tag*⟩, list-of ⟨*attribute*⟩, list-of ⟨*content*⟩)
⟨*attribute*⟩	::=	⟨*name*⟩ = ⟨*value*⟩
⟨*pi*⟩	::=	pi(⟨*atom*⟩)
⟨*sdata*⟩	::=	sdata(⟨*atom*⟩)
⟨*ndata*⟩	::=	ndata(⟨*atom*⟩)
⟨*cdata*⟩	::=	⟨*atom*⟩
⟨*name*⟩	::=	⟨*atom*⟩
⟨*value*⟩	::=	⟨*svalue*⟩
	|	list-of ⟨*svalue*⟩
⟨*svalue*⟩	::=	⟨*atom*⟩
	|	⟨*number*⟩

Fig. 1. SGML/XML tree representation in Prolog. The notation list-of ⟨*x*⟩ describes a Prolog is of terms of type ⟨*x*⟩.

take the role of tags in specified context, so `name=value` is automatically translated into a tree and the *omitted-tag* feature that allows the user to omit tags that can be inferred automatically from the context. The omitted-tag feature allows HTML documents to omit closing a paragraph using `</p>`. Disambiguating these abbreviations require the document syntax (DTD) and relatively complicated software.

XML documents are used to represent text using custom application-oriented tags as well as a serialization protocol for arbitrary data exchange between computers. XHTML is HTML based on XML rather than SGML.

The first SGML parser for SWI-Prolog was created by Anjo Anjewierden based on the SP parser by James Clark. A stable Prolog term-representation for SGML/XML trees plays a similar role as the DOM (*Domain Object Model*) representation in use in the object-oriented world. The term-structure is described in Fig. 1. Some issues have been subject to debate.

- An SGML/XML document consists of a single element, so ⟨*document*⟩ could have been ⟨*element*⟩ instead of a content-list. We have choosen for the list notation for uniform handling of partial and full documents.
- Representation of text by a Prolog atom is biased by the use of SWI-Prolog which has no length-limit on atoms and atoms that can represent Unicode text (see Sect. 5.2). At the same time SWI-Prolog stacks are limited to 128MB. Using atoms only the structure of the tree is represented on the stack, while the bulk of the data is stored on the unlimited heap. Especially Prolog implementations with unlimited stacks and packed arrays for storing strings can consider using strings for data. Using lists of character codes is another possibility addopted by both PilloW and ECLiPSe. Two observations make lists less attractive: lists use two cells per character while practical experience shows text is frequently processed as a unit only. This is especially true

for XML documents representing serialized data-structures, where it is also common to see the same value appearing many times in the document.
- Attribute values of multi-values attributes (e.g. `NAMES`) are returned as a Prolog list. This implies the DTD must be available to get unambiguous results. With SGML this is true anyway, but not with XML.
- Optionally attribute values of type `NUMBER` or `NUMBERS` are mapped to Prolog numbers. In addition to the DTD issues mentioned above, this conversion also suffers from possible loss of information. Leading zeros and float notation used is lost after conversion. Prolog systems with bounded arithmetic may also not be able to represent all values. Still, automatic conversion is useful in many applications, especially involving serialized data-structures.
- Attribute values are represented as *Name*=*Value*. Using **Name**(*Value*) is an alternative. The *Name*=*Value* representation was choosen for its similarity to the SGML notation as avoiding the need for univ (`=..`) for processing argument-lists

Realisation The SWI-Prolog SGML/XML parser is a C-library that has been built from scratch to reach at a lightweight parser. Total source is 11,835 lines. The parser provides two interfaces. Most natural to Prolog is **load_structure/3** which parses a Prolog stream into a term as described above. Alternatively, **sgml_parse/2** provides an *event-based* parser doing call-backs on Prolog for the SGML events. The call-back mode can deal with unbounded documents in streaming mode. It can be mixed with the term-creation mode, a feature that can be used to process long files with a repetative record structure in limited memory. Section 3.1 describes how this is used to process RDF documents.

Full documentation is available from `http://www.swi-prolog.org/packages/sgml2pl.html` The SWI-Prolog SGML parser has been adopted by XSB Prolog.

2.1 Generating documents from its Herbrand Term

If Prolog is used as a document processing or filtering tool, documents are read into a Herbrand Term as described in Sect. 2. The obvious continuation is to process this term into a new term and make it available again as SGML or XML document. This functionality is provided by the library `sgml_write.pl`. The library is complicated due to character encoding issues and different tradeoffs between strict maintenance of whitespace and layout for human readability.

2.2 Generating documents using DCG

The traditional method for creating web-documents is using print routines such as **write/1**, **writef/2** or **format/2**. Although simple and easy explain to novices, the approach has serious drawbacks from a software engineering point of view. In particular one has to be careful about HTML quoting rules, character encoding issues and ensure the code produces a valid HTML document. Automatic validation is vitually impossible using this approach.

```
        ...,
        mkthumbnail(URL, Caption, ThumbNail),
        output_html([ env(h1, [], ["Photo gallery"]),
                      ThumbNail
                    ]).

mkthumbnail(URL, Caption, Term) :-
        Term = [ env(table, [],
                     [ tr$[], td$[halign=center], img$[src=URL],
                       tr$[], td$[halign=center], Caption
                     ])
               ].
```

Fig. 2. Building PiLLoW terms

Alternatively we can produce a DOM term as describes in Sect. 2 and use the library described in Sect. 2.1 to create the HTML or XML document. Such documents are guaranteed to use proper nesting of elements, escape sequences and character encoding. The terms however are big, deeply nested and hard to read and write. Prolog allows them to be built from skeletons containing variables. This approach is taken by PiLLoW (Sect. 2.3) to control the complexity. In our opinion, the result is hard to read and write due to the unnatural order of statements as illustrated in Fig. 2 PilloW has partly overcome this shortcomming by defining a large number of 'utility terms' that are translated in a special way. **TBD: examples**

We introduced a DCG rules html//1. This rule translates proper trees into a list of high-level HTML/XML commands that are handed to **html_print/1** to realise proper quoting, character encoding and layout. The intermediate format is of no concern to the user and similar in structure to the 'flat' version of the PilloW representation, opening elements, inserting—quoted—text, and closing elements. Generated from the tree representation however, proper open and close of elements is guaranteed. Instead of passing sub-terms using variables, we allow for *Rule* embedded in the argument of html//1. It causes the grammar rule to call *Rule*. Figure 3 illustrates our approach. Note that any reusable part of the page generation can easily be translated into a DCG rule and the difference between direct translation of terms to HTML and rule-invocation is eminent.

In our current implemetation rules are called using meta-calling from html//1. Using **term_expansion/2** it is straightforward to move the rule invocation out of the term, using variable substitution similar to PilloW. It is now also possible to recursively expand the generated tree and validate it to the HTML DTD at compile-time and even insert omitted tags at compile-time to generate valid XHMTL from an incomplete specification. A complete overview of the argument to html//1 is given in Fig. 4.

2.3 Comparing to PiLLoW

The PiLLoW library (**?**) is another well established framework for web-programming based on Prolog. PiLLoW defines **html2terms/2**, converting be-

```
affiliation_table :-
        findall(Name-Aff, affiliation(Name, Aff), Pairs0),
        keysort(Pairs0, Pairs),
        reply_page(table([border(2),align(center)],
                         [ tr([th('Name'), th('Affiliation')])
                         | \affiliations(Pairs)
                         ])).

affiliations([]) -->
        [].
affiliations([H|T]) -->
        affiliation(H),
        affiliations(T).

affiliation(Name-Aff) -->
        html(tr(td(Name), td(Aff))).

% database
affiliation(wielemaker, uva).
affiliation(huang, vu).
affiliation('van der meij', vu).

% Page template
reply_page(Term) :-
        format('Content-type: text/html~n~n'),
        phrase(html(Term), Tokens),
        print_html(Tokens).
```

Fig. 3. Library html_write.pl in action

⟨*html*⟩	::=	list-of ⟨*content*⟩
	|	⟨*content*⟩
⟨*content*⟩	::=	⟨*atom*⟩
	|	&⟨*entity*⟩
	|	⟨*tag*⟩(list-of ⟨*attribute*⟩, ⟨*html*⟩)
	|	⟨*tag*⟩(⟨*html*⟩)
	|	\⟨*rule*⟩
⟨*attribute*⟩	::=	⟨*name*⟩(⟨*value*⟩)
⟨*tag*⟩	::=	⟨*atom*⟩
⟨*entity*⟩	::=	⟨*atom*⟩
⟨*value*⟩	::=	⟨*atom*⟩
	|	⟨*number*⟩
⟨*rule*⟩	::=	⟨*callable*⟩

Fig. 4. The html//1 argument specification

tween an HTML string and document represented as a Herbrand term. There are fundamental differences between PiLLoW and the primitives described here.

- PiLLoW uses a term that is passed to **html2terms/2**. Complex terms by composing them using partial terms passed through Prolog variables, where our approach inserts \ escape sequences calling DCG rules. As a result, PiL-

```
[env(table, [], [tr$[], td$[], "Hello"])]

[element(table, [],
         [ element(tbody, [],
                   [ element(tr, [],
                             [ element(td, [rowspan='1', colspan='1'],
                                       ['Hello'])])])])]
```

Fig. 5. Term representations for `<table><tr><td>Hello</table>` in PiLLoW (top) and our parser (bottom). Our parser completes the `tr` and `td` environments, inserts the omitted `tbody` element and inserts the defaults for the `rowspan` and `colspan` attributes

LoW defines a large number of 'convenience terms' that are handled special, while in our approach new high-level primitive can be defined and used naturally using DCG definitions and \ terms to call them. In a way, the approaches are complementary, as the output of our html//1 DCG produces a list of tokens similar in spirit to **html2terms/2** without environents. This format with all its special shorthands however is not exposed to the user.

- The PiLLoW parser does not create the SGML document tree. It does not insert omitted tags, default attributes, etc. As a result, HTML documents that differ only in omitted tags and whether or not default attributes are included in the source produce different terms. In our approach the term representation is equivalent, regardless of the input document. This is ilustrated in Fig. 5. Having a cannonical DOM representation greatly simplifies processing parsed HTML documents.

3 RDF documents

Where the datamodel of both HTML and XML is a tree-structure with attributes, the datamodel of the Semantic Web RDF language consists of {*Subject*, *Predicate*, *Object*} *triples*. Both *Subject* and *Predicate* are a *URI*.[1] *Object* is either a URI or a *Literal*. As the *Object* of one triple can be the *Subject* of another, a set of triples forms a graph, where each edge is labeled with a URI (the *Predicate*) and each vertex is either a URI or a literal. Literals have no out-going edges. Figure 6 illustrates this.

A number of languages are layered on of the RDF triple model. RDFS provides a frame-based representation. The OWL-dialects provide three increasingly complex Description Logic (DL) languages. **TBD: ref** SWRL is a proposal for a rule language.

The W3C standard for exchanging these triple models is an XML application known as RDF/XML.[2] This representation is widely accepted, but at the same

[1] URI: *Uniform Resource Identifier* is like a URL, but need to refer to an existing resource on the Web.
[2] http://www.w3.org/RDF/

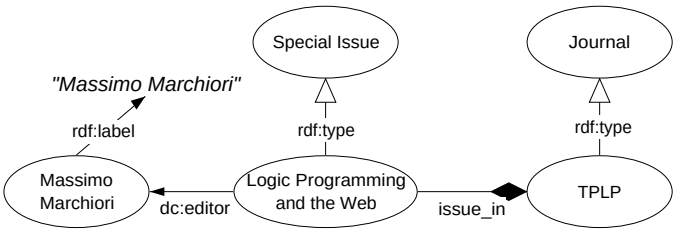

Fig. 6. Sample RDF graph. Ellipses are vertices representing URIs. Quoted text is a literal. Edges are labeled with URIs.

⟨*subject*⟩	::=	⟨*URI*⟩
⟨*predicate*⟩	::=	⟨*URI*⟩
⟨*object*⟩	::=	⟨*URI*⟩
	|	literal(⟨*lit_value*⟩)
⟨*lit_value*⟩	::=	⟨*text*⟩
	|	lang(⟨*langid*⟩, ⟨*text*⟩)
	|	type(⟨*URI*⟩, ⟨*text*⟩)
⟨*URI*⟩	::=	⟨*atom*⟩
⟨*text*⟩	::=	⟨*atom*⟩
⟨*langid*⟩	::=	⟨*atom*⟩ (ISO639)

Fig. 7. RDF types in Prolog.

time critized for its poor readability and complex parser requirements. Two popular alternative representations are Turtle and N3. **TBD: Refs!**

As there are multiple XML tree representations for the same triple-set as well as multiple RDF serialization formats, RDF documents cannot be processed at the level of the DOM as described in Sect. 2. The triple and graph-representation are clearly the most natural representations for an RDF document in Prolog. First we must decide on the representation of URIs and literals. As a URI is a string and the only operation defined on URIs by SW languages is equivalence test, using a Prolog atom is the clear winner.[3] Literals are expressed as **literal**(*Value*). The full type description is in Fig. 7.

The typical SW use-scenario is to 'harvest' triples from multiple sources and collect them in a database before start reasoning with them. This requires for access as a Prolog predicate. Here we have two options. One is the obvious predicate **rdf**(*Subject, Predicate, Object*) using the argument types described above. The alternative is to map each RDF predicate on a Prolog predicate **Predicate**(*Subject, Object*). We have choosen for **rdf/3** because it makes queries with uninstantiated predicates much easier and a single predicate makes it easier to manage the predicate and module namespaces than an unbounded set of predicates with unknown names.

[3] Some people may consider introducing a representation based on the notion of XML namespaces, such as *NS:Local*. Decomposing a URI into its namespace and local name however is only relevant during I/O, while such a representation is more expensive, both considering memory usage and time. Also consider indexing if URIs are asserted into the Prolog database.

```
load_triples(File, Options) :-
        process_rdf(File, assert_triples, Options).

assert_triples([], _).
assert_triples([rdf(S,P,O)|T], Src) :-
        rdf_assert(S, P, O, Src),
        assert_triples(T, Src).
```

Fig. 8. Loading triples using **process_rdf/3**

3.1 Input and output of RDF documents

The RDF/XML parser is realised as a Prolog library on top of the XML parser described in Sect. 2. Similar to the XML parser it has two interfaces. The **load_rdf**(*+Src, -Triples, +Options*) parses a document and returns a Prolog list of **rdf**(*S,P,O*) triples. Note that despite harvesting to the database is the typical use-case scenario, the parser delivers a list of triples for maximal flexibility. The predicate **process_rdf**(*+Src, :Action, +Options*) exploits the mixed call-back/convert mode of the XML parser to process the RDF file one *description* (record) at a time, calling *Action* with list a triples extracted from the description. Figure 8 illustraces how this is used by the storage module to load unbounded files with limited stack usage. Source location as ⟨*file*⟩:⟨*line*⟩ is passed to the *Src* argument of **assert_triples/2**.

The parser from XML and RDF triples covers the full RDF specification, including Unicode handling, RDF datatypes and RDF language tags. The Prolog source is 1,788 lines. It processes approximately 12,000 triples per second on an AMD 1600+ based computer. Implementation details and evaluation of the parser are described in (Wielemaker et al. 2003).[4]

Writing RDF/XML documents is supprisingly complicated, especially considering human-readable output. In the first phase we resolve all namespaces used in the graph as we have to declare these in the document header. As we only have triples and URIs we have to process each URI in each triple to collect the set of namespaces. In the write phase we process the triples grouped by subject, creating one RDF description per subject. First we write all named subjects and in-line anonymous subjects referenced by the named subject. At the end we write all anonymous (un-named) subjects that have not been written as part of the description of a named subject.

We have two libraries for writing RDF/XML. One, **rdf_write_xml**(*+Stream, +Triples*), is the inverse of **load_rdf/2**, writing an XML document from a list of **rdf**(*S,P,O*) terms. The other, called **rdf_save/2** is part of the RDF storage module, writing a named graph or the entire database directly from the database. The first (rdf_write_xml/2) is frequently used to exchange computed graphs to external programs using network communication, while the second (rdf_save/2) is more used to save modified graphs back to file. The

[4] The parser described did not support RDF datatypes and RDF language tags.

Index pattern			Calls
-	-	-	58
+	-	-	253,554
-	+	-	62
+	+	-	23,292,353
-	-	+	633,733
-	+	+	7,807,846
+	+	+	26,969,003

Table 1. Call-statistics on a real-world system

resulting code duplication is unfortunate, but unavoidable. Creating a temporary named graph in the database requires potentially much memory, and harms concurrency, while graphs fetched from the database into a list may not fit in the Prolog stacks and is also considerably slower than a direct write.

3.2 Storage of RDF

Assuming the 'harvesting' use-case, we need to realise a predicate **rdf**(*?S,?P,?O*). Indexing the database is crucial for good performance. Table 1 illustrates the calling pattern from a real-world application with over 4 million triples. In addition, we know our data is described by Fig. 7. The RDF store was developed in the context of a project which formulated the following requirements. **TBD: name projects and web pages. MIA, HOPS and MultiMedian**

- Upto at least 10 million triples on 32-bit hardware.
- Fast graph traversal using any instantion pattern.
- Include rdfs:subPropertyOf in search with small overhead. **TBD: explain rdfs:subPropertyOf**
- Case-insensitive search on literals.
- Prefix search on literals for completion in the UI.
- Searching for words that appear in literals.
- Multi-threaded access based on read/write locks.
- Transaction management and persistent store.
- Maintain source information, so we can update, save or remove data based on its source.
- Fast load/save of current state.

Within SWI-Prolog, especially the indexing and space-requirements to scale to 10 million triples formed the bottleneck. Considering the C-interface that supports non-deterministic predicates implemented in C we decided to realise the low-level store in C. For the store we took the following design decisions.

- The RDF predicates are represented as unique entities and organised according to the rdfs:subPropertyOf relation in multiple hierarchies. The root of the

hierarchy is used to compute the hash for the triple. If there is no unique root due to a cycle an arbitrary predicate is assigned to be the root.

- Literals are kept in an AVL tree, sorted case-insensitive and case-preserving (e.g. AaBb...). The uppercase version is used to determine the hash-key of the triple. Space is saved by avoiding duplicates and lost by the AVL nodes. Practical experience on real-life data ranges between -5% to +10%.
- Resources are represented by Prolog atom-handles. The hash is computed from the handle-value. Note that avoiding the translation between Prolog atom and text both avoids duplication of data, but also a table-lookup. We consider this a crusial aspect.
- Each source is represented by a structure in a hash-table. for each source we maintain a triple-count and an MD5 sum computed from the set of triples that belong to the source. The MD5 sum is computed per-triple and added, so it can be maintained incrementally on both assert and retract and the sum is independent from the irrelevant order of the triples.
- Each triple is represented by the atom-handle for the subject, predicate-pointer, atom-handle or literal pointer for object, a pointer to the source, a line number, a general bit-flag field and 6 hash-next pointers covering all indexing patterns except for +,+,+. The un-indexed table is a simple linked list. The others are hash-tables that are automatically resized if they become too populated.

The store itself does not allow for writes while there are active reads in progress. If another thread is reading the write operation will stall until all threads have finished reading. If the thread itself has an open choicepoint a permission error exception is raised. To arrive at meaningful update semantics we introduced *transactions*. The thread starting a transaction obtains a write-lock, initially allowing readers to proceed. During the transaction all changes are recorded in a linked list of actions. Assert actions carry the new triple as data, delete actions hold a pointer to the triple-to-delete. If the transaction is commited the thread denies access for new readers and waits for all readers to vanish before updating the database and releasing the locks. Transactions are realised by **rdf_transaction**(*:Goal*). If *Goal* succeeds, its choicepoints are discarded and the transaction is committed. If *Goal* fails or raises an exception the transaction is discarded and **rdf_transaction/1** returns failure or exception. Transactions can be nested. Nesting a transaction places a transaction-mark in the list of actions of the current transaction. Committing implies removing this mark from the list. Discarding removes all action cells following the mark as well as the mark itself.

It is possible to monitor the database using **rdf_monitor**(*:Goal, +Events*). Whenever one of the monitored events happens *Goal* is called. Modifying actions inside a transaction are called during the commit. Modifications by the monitors are collected in a new transaction which is committed immediately after completing the preceeding commit. Monitor events are assert, retract, update, new_literal, old_literal, transaction begin/end and file-load. *Goal* is called in the modifying thread. As this thread is holding the database write lock all invocations of monitor calls are fully serialized.

Although the 12,000 triples per second of the RDF/XML parser ranks it among the fastest parsers, loading 10 million triples takes nearly 15 minutes. For this reason we developed a binary format. The format is described in (Wielemaker et al. 2003) and loads over 20 times faster than RDF/XML, while using about the same space. The format is independent from byte-order and word-length (32/64 bit machines). **TBD: Verify numbers; these are old**

Persistency is achieved through the library `rdf_persistency.pl`, which monitors the database to maintain a set of files in a directory. Each source known to the database is represented by two files, a file representing the initial state using the quick-load binary format and a file containing Prolog terms representing changes, called the *journal*. Full reliability can be achieved by flushing the Prolog streams after each change and enabling synchronous write on the file. Our current implementation flushes the Prolog streams, but does not use synchronous write to the file.

3.3 Reasoning with RDF documents

We have identified two approaches for reasoning on top of the plain RDF predicate for more high-level languages such as RDFS or OWL. One approach is taken by the SeRQL query system described in Sect. 6. It is based on the observation that these languages provide rules to deduce new triples from the set of known triples. The API for high level languages is now simply the **rdf/3** predicate, where **rdf**(*S,P,O*) is true for any triple in the deductive closure of the original triple set under the given language. The deductive closure can be realised using full forwards reasoning, deducing new triples until this is no longer possible or by a combination of backward reasoning and forward reasoning. An alternative approach is to consider RDFS or OWL at the conceptual level and introduce a set of predicates that are inspired on this level. This approach is taken by our library `rdfs.pl`, defining predicates such as **rdfs_individual_of**(*?Resource, ?Class*), **rdfs_subclass_of**(*?Sub, ?Super*). Figure 9 explains the difference in the approaches.

4 Supporting HTTP

HTTP, or HyperText Transfer Protocol, is the key W3C standard protocol for exchanging web-documents. All browsers and web-servers such as Apache **TBD: ref** implement it. The initial version of the protocol was very simple. The client request consists of a single line of the format ⟨*action*⟩ ⟨*path*⟩, the server replies with the requested document and closes the connection. Version 1.1 of the protocol is more complicated, providing additional name-value pairs in the request as well as the reply, features to request status such as modification time, transfer partial documents, etcetera.

When considering HTTP support in Prolog, we must consider both the client- and server-side. In both cases our choice is between doing it in Prolog or re-use an existing application or library by providing and interface for it. We compare

```
% triples
mary rdf:type woman .
women rdf:type rdfs:Class .
women rdf:subClassOf human .
human rdf:type rdfs:Class .

% entailment interface

?- rdf(mary, rdf:type, X).
X = woman ;
X = human ;
No

% RDFS interface
?- rdfs_individual_of(mary, Class).
X = woman ;
X = human ;
No
```

Fig. 9. Different interface styles for RDFS

our work to PiLLoW (Cabeza and Hermenegildo 2003) and the ECLiPSe HTTP services (Leth et al. 1996).

Given a basic TCP/IP socket library, writing an HTTP client is trivial (our basic client is 258 lines of code). Both PiLLoW and ECLiPSe include a client written in Prolog. More issues complicate the choice for a pure Prolog based server.

- The server is more complex, which implies there is more to gain by re-using external code. Our core server library counts 1,784 lines.
- A single computer can only host one server at the default TCP port 80. Using alternate ports often conflicts with firewall or proxy settings. This can be solved using a proxy server, such as the Apache *mod_proxy*, where the public Apache server redirects part of the website to the Prolog server.
- Servers by definition introduce security riscs. Administrators are reluctant to see non-proven software in the role of a public server. Using a proxy as above also reduces this risc, especially if the proxy blocks malformed requests.

Dispite the observations, like the ECLiPSe developers, we consider a pure Prolog based server worthwhile. As argued in Sect. 5.1, many Prolog web-applications profit from using state stored in the server or large resources such as WordNet **TBD: ref** cause long startup times. In such cases the use of CGI (Common Gateway Interface) is not appropriate as this starts a new copy of the application for each request. PiLLoW resolves this issue using *Active Modules*, where a simple CGI application talks using a private protocol to a continuously running Prolog server. Using a Prolog HTTP server and the Apache mod_proxy approach has the same benifits, but uses a standard protocol and is much more flexible as we can also deploy the Prolog server directly.

Another approach is embedding Prolog in another server like the Java based Tomcat server **TBD: ref**. Although feasible, embedding non-Java based Prolog

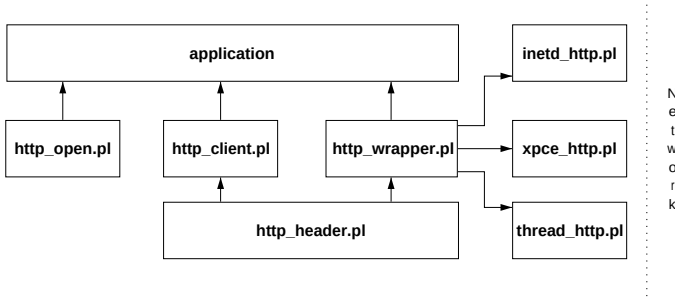

Fig. 10. Module dependencies of the HTTP library

systems in Java is complicated. Embedding through *jni* introduces platform and Java version dependent problems. Connecting Prolog and Java concurrency models and garbage collection is difficult and the resulting system is much harder the manage by the user than a pure Prolog based application.

In the following sections we describe our HTTP client and server libraries. An overall overview of the modules and their dependencies is given in Fig. 10.

4.1 HTTP client libraries

We support two clients. The first is a very lightweight client that is only capable of supporting the HTTP GET method. Its interface is **http_open**(*+URL, -Stream, +Options*). *Options* allows for setting a timeout or proxy as well as getting information from the reply-header such as the size of the document. The **http_open/3** predicate internally handles HTTP 3XX (redirect) replies. Other non-ok replies are mapped to a Prolog exception. After reading the document the user must close the returned stream-handle using the standard Prolog **close/1** predicate. This predicate makes accessing an HTTP resource as simple as accessing a local file. The second library called `http_client.pl` is more complicated. Providing support for HTTP POST and a plugin interface that allows for installing handlers for documents of specified MIME-types it shares `http_header.pl` with the server libraries for creating and analyzing HTTP headers. Currently provided plugins include `http_mime_plugin.pl` to handle multipart MIME messages and `http_sgml_plugin.pl` for automatically parsing HTML, XML and SGML documents. Figure 11 shows the code for fetching a URL and parsing the returned HTML document it into a Prolog term as described in Sect. 2.

Both the PiLLoW and ECLiPSe approach return the documents content as a string. Our interface is stream-based (http_open/3) or allows for plugin-based processing of the stream (http_get/3 and **http_post/4**). This interface avoids potentially large intermediate datastructures and allows for processing unbounded documents.

```
?- use_module(library('http/http_client')).
?- use_module(library('http/http_sgml_plugin')).

?- http_get('http://www.swi-prolog.org/', DOM, []).
DOM = [element(html, [version='-//W3C//DTD HTML 4.0 Transitional//EN'],
                     [element(head, [],
                              [element(title, [],
                                       ['SWI-Prolog\'s Home']),
       ...
```

Fig. 11. Fetching an HTML document

```
:- use_module(library('http/thread_httpd')).

start_server(Port) :-
        http_server(reply, [port(Port)]).

reply(Request) :-
        format('Content-type: text/plain~n~n'),
        writeln(Request).
```

Fig. 12. A simple HTTP server

4.2 The HTTP server library

The server library shares a large part of its implementation with **http_post/4** from the client library in creating and parsing the HTTP headers. Both to simplify re-use of application code and to make it possible to use the server without committing to a large infrastructure we adopted the reply-stragety of the CGI protocol, where the handler writes a page consisting of an HTTP header followed by the document content. The only obligatory field in the header is the Content-type. Other fields such as the Content-length are filled in by the server, which also takes care of relaying the data to the client. Figure 12 provides a simple example that returns the request-data to the client. By importing `thread_http.pl` we implicitely selected the multi-threaded server model. Other models provided are `inetd_http`, causing the (Unix) inet deamon to start a server for each request and `xpce_http` which uses I/O multiplexing based on select() to talk to multiple clients without using Prolog threads. The logic of handling a single HTTP request given a predicate realising the handler, an input and output stream is implemented by `http_wrapper`.

Replies other than "200 OK" are generated using an exception. For example to indicate the user has no access to a page we must use the following code fragment. Other recognised replies are defined by the predicate **http_reply**(*+Reply, +Stream, +HeaderExtra*). Other exceptions raised by the handler cause a "500 Server error" reply.

```
        ...,
        throw(http_reply(forbidden(URL))).
```

```
reply(Request) :-
        http_parameters(Request,
                        [ title(Title, [optional(true)]),
                          name(Name,   [length >= 2]),
                          age(Age,     [integer])
                        ]),
        ...
```

Fig. 13. Fetching HTTP form data

4.2.1 Form parameters

The library `http_parameters.pl` defines **http_parameters**(*+Request, ?Parameters*) to fetch and type-check parameters transparently for both request GET and POST data. Figure 13 illustrates the functionality. Parameter values are returned as atoms. If large documents are transferred using a POST request this may be undesirable and the user may wish to revert to **http_read_data**(*+Request, -Data, +Options*) which is also used by **http_get/3** to process arguments using plugins.

4.2.2 Session management

The library `http_session.pl` provides session over the stateless HTTP protocol. It does so by adding a cookie using a randomly generated code if no valid session id is found in the current request. The interface to the user consists of a predicate to set options (timeout, cookie-name and path) and a set of wrappers around **assert/1** and **retract/1**, the most important of which are **http_session_assert**(*+Data*), **http_session_retract**(*?Data*) and **http_session_data**(*?Data*). In the current version the data associated with sessions that have timed out is simply discarded. Session-data does not survive the server.

Note that a session generally consists of a number of HTTP requests and replies. Each *request* is scheduled over the available worker threads and requests belonging to the same session are therefore normally not handled by the same thread. This implies no session state can be stored in global variables or in the control-structure of a thread. If such style of programming is wanted the user must create a thread that represents the session and setup communication from the HTTP-worker thread to the session thread. Figure 14 illustrates the idea.

4.2.3 Evaluation

Table 2 shows performance testing using a trivial query using **http_get/3** and server using the multi-threaded server model. Tests were executed on a dual AMD 1600+ running SWI-Prolog 5.6.10 on SuSE Linux 10.0.

5 Enabling extensions to the Prolog language

SWI-Prolog has been developed in the context of projects and many leading projects in the past 5 years caused the development to focus on managing web documents

```
reply(Request) :-                               % HTTP worker
        (   http_session_data(thread(Thread))
        ->  true
        ;   thread_create(session_loop([]), Thread, [detached(true)]),
            http_session_assert(thread(Thread))
        ),
        current_output(CGIOut),
        thread_self(Me),
        thread_send_message(Thread, handle(Request, Me, CGIOut)),
        thread_get_message(Status).
        (   Status == true
        ->  true
        ;   Status == exception(Term)
        ->  throw(Term)
        ).

session_loop(State) :-                          % Session thread
        thread_get_message(handle(Request, Sender, CGIOut)),
        catch(next_state(Request, State, NewState, CGIOut), E, true),
        (   var(E)
        ->  thread_send_message(Sender, true),
            session_loop(NewState)
        ;   thread_send_message(Sender, exception(E))
        ).
```

Fig. 14. Managing a session in a thread. The **reply/1** predicate is part of the HTTP worker pool, while **session_loop/1** is executed in the thread handling the session.

Connection	Elapsed	Server CPU	Client CPU
Close	20.84	11.70	7.48
Keep-Alive	16.23	8.69	6.73

Table 2. HTTP performance executing a trivial query 10,000 times. Times are in seconds.

and protocols. In the previous sections we have described our web-enabling libraries. In this section we describe extensions to the ISO-Prolog standard (Deransart et al. 1996) we consider crusial for scalable and comfortable deployment of Prolog as an agent in a web-centered world.

5.1 Multi-threading

Concurrency is a necessary property for real-world web-applications for reasons we describe below.

- Network delays may cause communication of a single transaction to take very long. Such clients should not block access for other clients. This can be

achieved using multiplexed I/O, for example based on the POSIX select() API, multiple processes handling requests in a pool or multiple threads in one or more processes handling requests in a pool.
- CPU intensive services must be able to deploy multiple CPUs. This can be achieved using multiple instances of the service and some form of load-balancing or a single server running on multi-processor hardware or a combination of the two.

As indicated, none of the requirements above require multi-threading support in Prolog. Nevertheless, we added multi-threading (Wielemaker 2003) because it resolves the problems menitioned above for medium-scale applications while greatly simplifying deployment and debugging in a platform independent way. A multi-threaded server also allows maintaining state for a specific session or even shared between multiple sessions simply in the Prolog database. This is particulary interesting for accessing the RDF database described in Sect. 3.2.

5.2 Unicode support

Unicode[5] is a character encoding system that assignes unique code-points to all characters of almost all scripts known in the world. In Unicode 4.0, the code-points range from 1 to 0x10FFFF. Unicode allows for handling documents from different scripts and documents using multiple scripts easily and transparently, a feature that is very important in a general web-processing application. Even traditional HTML applications use it to insert special characters through entities such as the copyright (©) sign, greek and mathematical symbols, etc. As illustrated in the famous *Semantic Web layer cake* in Fig. 15, Unicode is at the heart of the sematic web.

For HTML we could represent text using Prolog strings. As lists of Prolog integers a string is capable of representing the full Unicode set. As we have claimed in Sect. 2 however, using Prolog strings is not the most obvious choice. Please also note that both XML attribute names and values can contain arbitrary Unicode characters, requiring the unnatural use of strings for these as well. If we consider RDF, URIs can have arbitrary Unicode characters **TBD: ref!** and we want to represent URIs as atoms to exploit compact storage as well as fast equivalence testing. Without Unicode support in atoms we could use a cannonical version of the standard encoding of Unicode URIs in ASCII, which first encodes the Unicode string using UTF-8 and uses the %XX notation for all characters except for 'standard' ASCII characters. **TBD: Improve description, give example using Greek (Chinese?) to show what it looks like**.

All these work arounds **TBD: better word** can be avoided by introducing Unicode in the Prolog kernel. Allowing atoms to contain arbitrary long sequences of arbitrary Unicode characters we can represent text in Web markup languages uniformly and without loss.

[5] http://www.unicode.org/

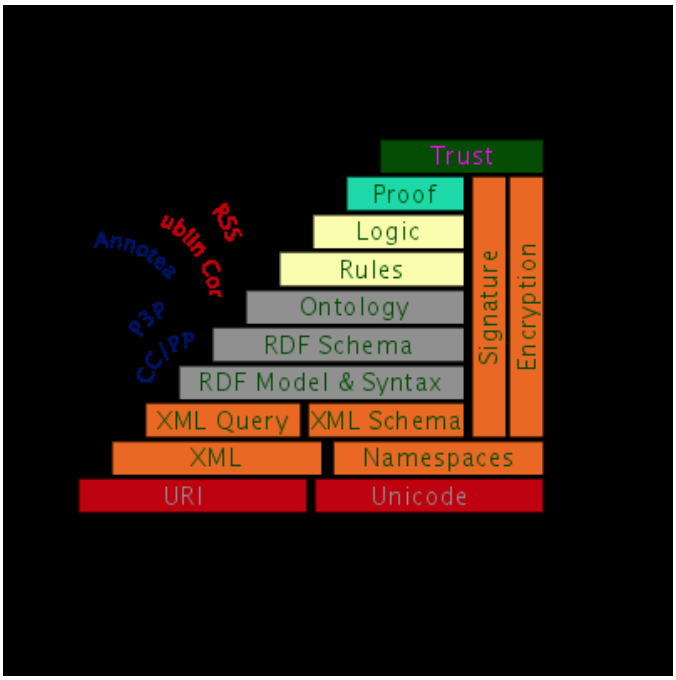

Fig. 15. The Semantic Web layer cake by Tim Burners Lee

SWI-Prolog internally uses a dual representation, one for 8-bit atoms using ISO-8859-1 encoding (Latin-1, a sub-set of Unicode) and one using the C *wchar_t* representation. The dual representation is completely hidden from the Prolog user. It is exposed to users of the foreign language interface, providing wide-character versions for the text-exchange functions. One of the motivations for using the dual representation over UTF-8 is that SWI-Prolog users commonly represent arbitrary binary data in atoms and using UTF-8 internally would break this.

5.3 Atom handling

We have already concluded that Unicode and unlimited length of atoms simplify web document processing. Continuously running servers however also must avoid memory leaks and therefore processing dynamic data using atoms requires atom garbage collection.

6 Case study — A Semantic Web Query Language

In this case-study we describe the SWI-Prolog SeRQL implementation.[6] SeRQL is an RDF query language developed as part of the Sesame project[7] (Broekstra et al.

[6] http://www.swi-prolog.org/packages/SeRQL
[7] http://www.openrdf.org

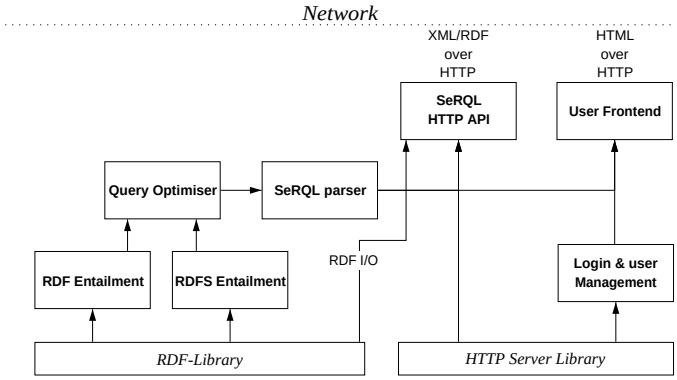

Fig. 16. Module dependencies of the SeRQL system. Arrows denote 'imports from' relations.

```
:- module(rdf_entailment, [rdf/3]).

rdf(S, P, O) :-
        rdf_db:rdf(S, P, O).
rdf(S, rdf:type, rdf:'Property') :-
        rdf_db:rdf(_, S, _),
        \+ rdf_db:rdf(S, rdf:type, rdf:'Property').
rdf(S, rdf:type, rdfs:'Resource') :-
        rdf_db:rdf_subject(S),
        \+ rdf_db:rdf(S, rdf:type, rdfs:'Resource').

:- multifile serql:entailment/2.
serql:entailment(rdf, rdf_entailment).
```

Fig. 17. RDF entailment module

2002). SeRQL uses HTTP as its access protocol. Sesame consists of an implementation of the server as a Java servled and a Java client-library. By implementing a compatible framework we made our Prolog based RDF storage and reasoning engine available to Java clients. The Prolog SeRQL implementation uses all of the described SWI-Prolog infrastructure and building it has contributed significantly. Figure 16 lists the main components of the server.

The *entailment* modules are plugins that implement the entailment approach to RDF reasoning described in Sect. 3.3. They implement **rdf/3** as a pure predicate, adding implicit triples to the raw triples loaded from RDF/XML documents. Figure 17 shows the somewhat simplified entailment module for RDF. The multifile rule registers the module as entailment module for the SeRQL system. New modules can be loaded dynamically into the platform, providing support for other SW languages or application-specific server-side reasoning. Prolog's dynamic loading and re-loading allows for updating such reasoning modules on the life server.

The SeRQL parser is a DCG-based parser translating SeRQL source into a compound goal calling **rdf/3** and predicates from the SeRQL runtime library which

```
...
rdf(Paper, author, Author),
rdf(Author, name, Name),
rdf(Author, affiliation, Affil),
...
```

Fig. 18. Split rdf conjunctions. After executing the first **rdf/3** query *Author* is bound and the two subsequent queries become independent. This is also true for other orderings, so we only need to evaluate 3 alternatives instead of 3! (6).

provides comparison and functions built into the SeRQL language. The resulting control-structure is passed to the query optimiser (Wielemaker 2005) which uses statistics maintained by the RDF database to reorder the pure **rdf/3** calls for best performance. The optimiser uses a generate-and-evaluate approach to find the optimal order. Considering the frequently long conjunctions of **rdf/3** calls, the conjunction is split into independent parts. Figure 18 illustrates this in a very simple example. During abstract execution, information on instantiation and types implied by the runtime library predicates is attached to the variables using dynamic attributed variables. (Demoen 2002).

HTTP access consists of two parts. The human-centered portal consists of HTML pages with forms to administer the server as well as view statistics, load and unload documents and run SeRQL queries presenting the result as an HTML table. Dynamic pages are generated using the `html_write.pl` library described in Sect. 2.2. Static pages are served from HTML files by the Prolog server. Machines use HTTP POST requests to provide query data and get a reply in XML or RDF/XML.

The system knows about various RDF input and output formats. To reach modularity the kernel exchanges RDF graphs as lists if terms **rdf**(*S,P,O*) and result-tables as lists of terms using the functor `row` and arity equal to the number of columns in the table. The system calls a multifile predicate using the format identifier and data to realise the requested format. The HTML output format use `html_write.pl`. The RDF/XML format uses **rdf_write_xml/2** described in Sect. 3.1. Both **rdf_write_xml/2** and the other XML output format use straight calls **format/3** to write the document, were quoting values is realised by quoting primitives provided by the SGML/XML parser described in Sect. 2. Using direct writing instead of techniques described in Sect. 2.2 avoids potentionally large intermediate datastructures and is not very complicated given the very simple structure of the documents.

6.1 Evaluation

The SeRQL server and the SWI-Prolog library development is too closely integrated to use it as an evaluation of the functionality provided by the web enabling libraries. We compared our server to Sesame, written in Java. The source code of the Prolog based server is 6,700 lines, compared to 86,000 for Sesame. As both systems have very different coverage in functionality and can re-use libraries at different levels it is hard to judge these figures. Both answer trivial queries in approximately 5ms

on a dual AMD 1600+ PC running Linux 2.6. On complex queries the two systems perform very different. Sesame's forward reasoning make it handle some RDFS queries much faster. Sesame does not contain a query optimizer which cause order-dependent and sometimes very long response times on large conjunctions.

The power of LP where programs can be handled as data is exploited by parsing the SeRQL query into a program, optimizing the program by manipulating it as data, after which we can simply call it to answer the query. The non-deterministic nature of **rdf/3** allow for a trivial translation of the query to a non-deterministic program that produces the answer on backtracking.

The server only depends on the Prolog file and the standard Prolog libraries. It runs unmodified on all systems supporting SWI-Prolog and has been tested on Windows, Linux and MacOS X.

All infrastructure described is used in the server. We use **format/3**, exploiting XML quoting primitives provided by the Prolog XML library to print highly repetitive XML files such as the SeRQL result-table. Alternatively we could have created the corresponding DOM term and call **xml_write/2** described in Sect. 2.1. **TBD: Try and compare**

7 Case study — XDIG

7.1 Introduction

Ontology management and reasoning about ontologies have become important issues for Semantic Web applications. Description Logics (DL) underpins the Web Ontology Language OWL-DL. DL-based reasoners, like Racer and FACT++, have become popular tools for ontology reasoning in the Web. The DIG description logic interface (Bechhofer et al. 2003), DIG interface for short, which is defined by the Description Logic Implementation Group (DIG)[8], provides a convenient high-level interface for DL reasoners. Many DL reasoners support the DIG interface and therefore more easily allow for the construction of highly portable and reusable components or extensions.

Most DL reasoners only support standard DL reasoning. The features are well formalized in the basic framework of Description Logics(Baader et al. 2003). However, a lot of Semantic Web applications may require non-standard DL reasoning services, like reasoning with inconnsistent ontologies, diagnosis and repair of inconsistent ontologies, reasoning with multi-version ontologies, and consistent ontology evolution and changes. Those non-standard DL reasoning have been proved to be very useful for practial Semantic Web applications(Huang et al. 2005; Huang and Stuckenschmidt 2005b; Huang and Stuckenschmidt 2005a; Schlobachm and Huang 2005).

In this section, we describe a DIG DL interface extension that, from a client application point of view, defines both DL reasoner services and special purpose services as provided by an intermediate extended description logic server; as a

[8] http://dl.kr.org/dig/

regular DIG client, an intermediate server can call an external DL reasoner which supports the DIG interface.

This extended DIG description logic interface, called XDIG, has been implemented as a package for Prolog. We will describe the extended description logic interface for Prolog in detail.

The XDIG has been used to develop non-standard DL reasoning services in the SEKT project[9]. The following open-source systems are powered by XDIG.

- **PION**: PION is a reasoning system that deals with inconsistent ontologies[10]. PION supports TELL requests both in DIG and OWL, and ASK requests in DIG(Huang and Visser 2004; Huang et al. 2005).
- **MORE**: MORE is a Multi-version Ontology REasoner which is based on a temporal logic approach [11]. MORE supports variant queries, including temporal reasoning queries, ontology comparison queries, and version retrieval queries(Huang and Stuckenschmidt 2005b).
- **DION**: DION (a Debugger of Inconsistent ONtologies) is a system for diagnosis and repair of inconsistent ontologies[12](Schlobachm and Huang 2005). DION supports multiple ontology languages: DIG data format and OWL.

In this section we will describe how XDIG is used to develop PION for reasoning with inconsistent ontologies, as an example of XDIG for the development of ontology management and reasoning system by using Prolog.

7.2 Extended DIG Description Logic Interface

7.2.1 The DIG DL Interface

The DIG interface is defined as a simple API for a general description logic system(Bechhofer et al. 2003). It uses a similar mechanism as SOAP (Simple Object Access Protocol), which has XML-based messaging protocols on top of HTTP.

Clients of a DL reasoner communicate through the use of HTTP POST requests. The body of the request is an XML encoded message which corresponds to the DL concept language. The DIG concept language is a description logic that includes the standard boolean concept operators, universal and existential restrictions, and other issues. A TELL request is used to assert DL statements in the knowledge base of the DL reasoner. An ASK request is used to perform knowledge base queries. In addition, management requests are used to maintain the knowledge base of DL reasoners or to obtain particular information of the system, like a reasoner identification. See (Bechhofer et al. 2003) for more DIG description logic interface details.

[9] http://www.sekt-project.com
[10] http://wasp.cs.vu.nl/sekt/pion
[11] http://wasp.cs.vu.nl/sekt/more
[12] http://wasp.cs.vu.nl/sekt/dion

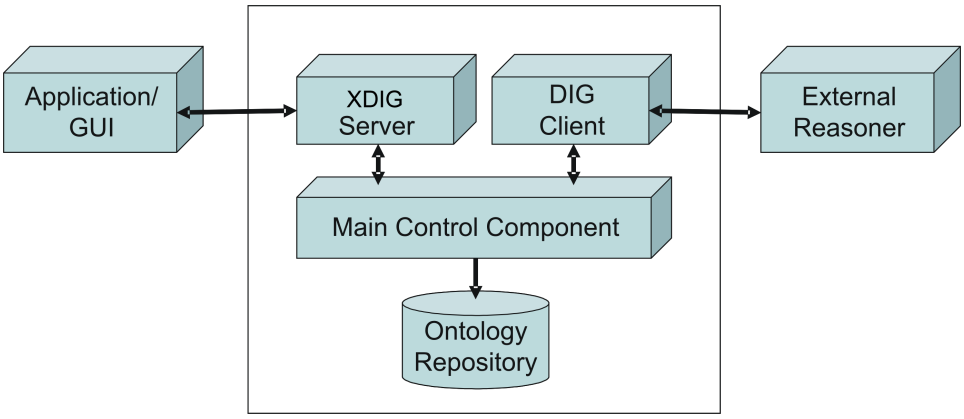

Fig. 19. Architecture of XDIG.

7.2.2 Architecture of XDIG

The XDIG libraries may be used to build DL reasoners that have additional reasoning capabilities. It is not necessary for extended Prolog-based DL reasoning systems to incorporate their own DL reasoning component. Several well-known DL reasoners exist (e.g. Racer(Haarslev and Möller 2001)) and an extended DL reasoner will access an existing external reasoner via its DIG interface.

In general, extended DL reasoners should be able to serve as a regular DL reasoner via their corresponding DIG description logic interface. Moreover, they will provide particular supplementary reasoning facilities. An intermediate extended DIG server can make systems independent of particular application specific characteristics, which significantly improves the reusability and applicability of software components; a highly decoupled infrastructure is usually beneficial for the construction of domain-specific services.

The general architecture of XDIG is shown in Figure 19. It consists of the following components:

- **XDIG Server**: The XDIG server deals with requests from ontology applications. It supports the extended DIG interface, i.e. it not only supports standard DIG/DL requests, like 'tell' and 'ask', but also additional processing features, like the identification of the reasoner and the change of system settings.
- **DIG Client**: XDIG is designed to rely on an external DL reasoner. It has a regular DIG interface client layer and calls the external DL reasoner in order to access the standard DL reasoning capabilities.
- **Main Control Component**: In order to provide its own control processing, XDIG has the main control component, which implements a general processing framework, like query analysis, query pre-processing, and a reasoning strategy, by interacting with local ontology repository.
- **Ontology Repository**: Ontology Repository serves as an internal knowledge base, which is used to store multiple ontologies locally. These ontology

statements are used for further processing when the reasoner receives an ASK request. The main control component usually selects some parts of ontologies to post them to an external DL reasoner in order to obtain the corresponding answers. This internal KB is also used to store system settings and other additional information about the system.

7.2.3 XDIG Prolog Libraries

The XDIG prolog package consists of the following libraries: dig_client, dig_server, dig_process, dig_db, and dig_client_setting.

- The library dig_client provides the mechanism to call an external DL reasoner. It defines the predicate: $dig\_post(+Data, -Reply, +Options)$ posts the data to the external DIG server with *Options* that are permitted by the HTTP POST request. The reply *Reply* from a DIG server has the form anwer(Header, Elements) where *Header* is the header of the response and *Elements* an XML element list which corresponds to the body of the DIG server response.
- The library dig_server provides a mechanism to build a HTTP server which supports the XDIG interface. It defines the following predicates: $dig\_server(+Request)$ processes a client's *Request*. It serves as the main entry point for the server, which is launched by an http_server process. XDIG server developers have to define their own predicate $my\_dig\_server\_processing(+Data, -Answer, +Options)$ to handle the corresponding *Data* (i.e. the body of a request without a header) and *Answer*. A XDIG server can be launched from a Prolog program by means of Prolog's HTTP server library as it has been discussed in Section 4.2:

```
:- http_server(dig_server,[port(8001)]).
```

- The library dig_process provides the main predicates to process XDIG messages. It defines several data conversion predicates, like these: $xml\_elements(+RawText, -XMLElements, -Header)$ translates a raw text reply from the server into a body with a list of elements and a text header. $elements\_xmltext(+XMLElements, -XMLText$ translates a list of XMLElements to an XML-encoded text. $dig\_requestdata\_analysis(+RequestData, -Data, -Type)$ gets the *Data* body from *RequestData* with type *Type*, where *Type* can be one of : $asks$, $tells$, $getIdentifier$, etc.
- The library dig_db provides facilities to maintain the ontology repository. It defines the following predicates: $dig\_assert\_data(+ID, +Data)$ asserts a data statement into the knowledge base with identifier ID. $dig\_db\_element(+ID, ?Element)$ checks or gets an *Element* from the knowledge base ID.
- The Library *dig_client_setting* is used to manage the settings of the corresponding external DIG DL server.

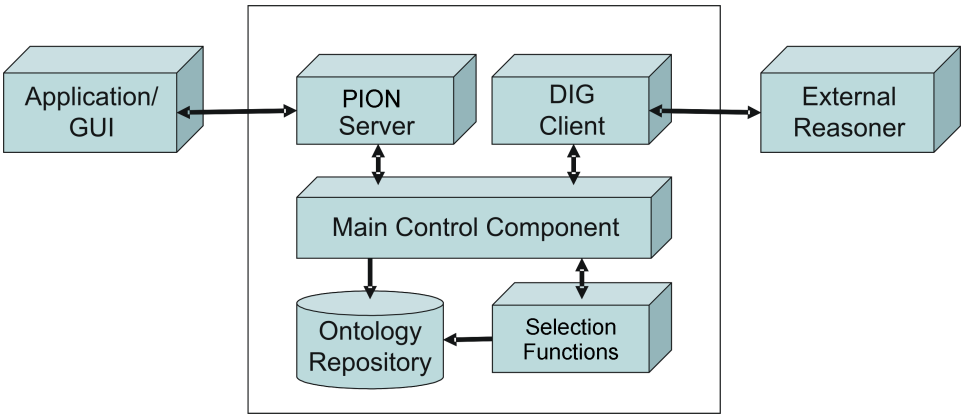

Fig. 20. Architecture of PION.

7.3 PION: a System of Reasoning with Inconsistent Ontologies Powered by XDIG

The classical entailment in logics is *explosive*: any formula is a logical consequence of a contradiction. Therefore, conclusions drawn from an inconsistent knowledge base by classical inference may be completely meaningless. In (Huang et al. 2005), a general framework for reasoning with inconsistent ontologies is propoed. The main idea of PION is as follows: given a selection function, which can be based on the syntactic or semantic relevance as used in computational linguistics, we can always select some consistent sub-theory from an inconsistent ontology. Then we apply standard reasoning on the selected sub-theory to find meaningful answers. If it cannot give a satisfying answer, the selection function will loosen the relevance degree to obtain a consistent sub-theory for further reasoning. A PION serves as a DL reasoner via its own XDIG interface. It is designed to be a simple API for a general reasoner with inconsistent ontologies. It supports DIG requests from other ontology applications or other ontology and metadata management systems. Therefore, the implementation of PION will be independent of those particular applications and systems.

The PION architecture is an extended XDIG system with an additional component for selection functions, as shown in Figure 20. The selection function component as an enhanced component to XDIG defines the selection functions that can be used in the reasoning process.

PION is implemented by means of the XDIG libraries. The predicate '*my_dig_server_ processing*' serves as the standard entry point of the server and is defined for processing the 'tell' and 'ask' messages, as shown in the following. For a TELL request, PION stores the data into its own local ontology repositories. PION supports both the DIG data format and OWL. For the told data in OWL, PION will call its owl support predicate *owl2dig* to convert the data in OWL into one with DIG, i.e., the XDIG internal data format. For the ASK request, PION

posts the stored data in the repositories, then performs the queries on the external DL reasoner.

```
%deal with the tell request
my_dig_server_processing(RequestData, Answer, [connection(close)]):-
     dig_requestdata_analysis(RequestData, Data, Type),
     Type=tells,
     !,
     dig_post(RequestData, Answer, [connection(close)]),
     pion_setting(kb, KBID),
     dig_assert_data(KBID, Data).


%deal with owl tell request
my_dig_server_processing(RequestData, Answer, [connection(close)]):-
    dig_requestdata_analysis(RequestData, _Data, Type),
    Type = 'rdf:RDF',
    !,
    owl2dig(RequestData, elements(L)),
    dig_post(elements(L), Answer, [connection(close)]),
    dig_get_element(L, E, _),
    E = element(tells, _, L1),
    pion_setting(kb, KBID),
    dig_assert_data(KBID, L1).

%deal with the ask request
my_dig_server_processing(RequestData, Answer, _Options):-
    dig_requestdata_analysis(RequestData, Data, Type),
    Type=asks,
    !,
    dig_post(RequestData, Answer1, [connection(close)]),
    pion_answer_preprocessing(Answer1, Data, Answer).
```

If answers from the external reasoner do not specify particular error conditions, which would mean that an ontology is consistent, PION uses the answers from the external reasoner as its own answer to the applications. If the current ontology is inconsistent, PION will start the inconsistency processing by launching a search strategy, which uses one of selection functions in the system to find a relevant and consistent subset of the current ontology and apply the selected subset to the standard DL reasoner, as shown in the following:

```
pion_answer_preprocessing(Answer1, Queries, Answer):-
    Answer1 = answer(_Header, AnswerBody),
    AnswerBody = [element(responses, _, Elements)|_],
```

```
    pion_setting(kb, KBID),
    dig_consistent_check(kb(KBID), Status),
    Status=false,
    !,
    pion_setting(strategy, Strategy),
    pion_inconsistency_processing(KBID, Queries, Elements, Strategy, Answer).

pion_answer_preprocessing(Answer1, _Queries, Answer):-
    Answer = Answer1.
```

The PION testbed[13] allows for query tests with a comparison of query answers with PION and query answers without PION. Using a standard DL reasoner without PION, queries on an inconsistent ontology will often result in many errors, whereas queries on an inconsistent ontology with PION may result in intuitive answers. The tests show that Prolog, more concretely XDIG, is a convenient tool for non-standard DL reasoning services for the Semantic Web(Huang et al. 2005).

8 Case study — Facetted browser on Semantic Web database

8.1 Introduction

I will be adding unedited text for now. Absolutely not for reviewing yet. Am using cvs for backup of files.

8.2 Introduction

Prolog has been in use in the section AI of the Computer Department at the Vrije Universiteit for years. In the Multiagent group Prolog has proven to be an effective programming language, especially for the implementation of rule based systems, where some higher level reasoning language is defined. SWI-Prolog in particular has proven very valuable here as a prolog environment because it made porting our applications over multiple platforms (Windows, Unix, Linux) easy, it provides an integrated GUI xpce, the SWI-Prolog has proven to be relatively stable over our time of experience of 15 years and support was always great. **TBD: Could very well be not too relevant introduction**.

For this case study we describe a project in the context of the "Semantic Web": It is a pilot project for the STITCH-project[14] whose main aim is studying and finding solutions for the problem of integrating vocabularies, classification systems etc. in the Cultural Heritage domain.

In this CASE study we will not go into all details of the procedures and results of our Pilot study, they will be described elsewhere, but focus on parts of the project where SWI-Prolog played an illustrative role.

[13] http://wasp.cs.vu.nl/sekt/pion
[14] the STITCH project is a project within CATCH, funded by NWO **TBD: more details**

The pilot consisted of the integration of two collections – the Medieval Illuminations of the National Library of the Netherlands(Koninklijke Bibliotheek, abr. KB) and the Masterpieces collection from the Rijksmuseum (National Museum) – and developing a user interface for browsing the merged collections. One stringent requirement was the use of "standard semantic web techniques" during all stages, so as to be able to evaluate the added value SW techniques could bring. We expected to benifit from the advance made in the field of "Ontology mapping".

The problem could be split into three main tasks:

- Gathering of data, e.g. collections and thesauri and transforming them into SW notions, e.g. RDF (thesauri:SKOS..)
- Establishing mappings between the vocabularies
- Building a prototype to access (search and browse) the integrated collections: a web server.

For the Pilot project we did not develop ontology/structured vocabulary mappers ourselves. In all other phases SWI-Prolog played an important role as implementation language. All phases, all procedures/programs interacted by means of RDF files. This resulted in extensive use of SWI-Prologs rdf_db library.

8.3 Various Programs for transformation of data and construction of thesauri

8.4 Multifaceted search

TBD: Find structure for telling our story, is there a generalizable rationale for using SWI-Prolog

TBD: Early on I should split this up into: 1)Access to the database, querying 2)Generation of html code 3)Http web server

TBD: Hope to be able to evaluate SWI SeRQL - Sesame SeRQL: I did very easily succeed in switching from sesame to SWI: I do have the impression the performance is not as good: probably because we use subClassOf a lot where SWI's rdfs uses dynamical backtracking whereas Sesame does "forward instantiation". Might be an idea to provide the sesame xml based forward rule system as one "entailment" variant. But probably not relevant for this article

Multi faceted search is a search and browse paradigm where a collection is accessed by refining multiple (preferably) structured aspects of properties of elements of the collection. For the user interface and user interaction we have been influenced by the approach of Flamenco**TBD: Reference**.

An important part of the approach consists of providing all choices for refinement to be made with a number, telling how much collection elements will remain after applying the refinement defined by the choice. Important is that all choices leading to 0 elements being found to not be presented to the user**TBD: make the sentence**.

8.4.1 Interaction with the RDF-store

For the pilot project we decided on delegating as much of the algorithm to Semantic Web standard reasoning. We chose SeRQL. We started out using the sesame_client library that is part of SWI-Prolog SeRQL library using it to access an external Sesame web-server. We recently also started evaluating SWI-Prologs own SeRQL implementation.

The role of Prolog in the interaction between the User Interface and the RDF storage consisted mainly of building up complex SeRQL queries from URL query arguments, passing them on to the SeRQL-engine, gathering the result rows and filtering the output, mostly to provide a count of elements found. When using SeRQL for Multi Faceted search, the lack of SQL like constructs such as count(..) and group by is a shame**TBD: Rephrase!**.

Although post-processing of all our complex SeRQL queries needed similar manipulations we did not develop a generic procedure for this post processing.

TBD: Probably too vague, without concrete suggestions for it to me kept in the text: For simplicities sake we decided to limit ourselves to only reading from the rdf store, not asserting triples defining the current user query and formulating a SeRQL query referring to those dynamically added RDF-triples. This did imply that the SeRQL queries defining the refinement query had to be composed from the user query not statically expressible in SeRQL syntax, the more complex SeRQL queries had to be built up dynamically from the user query. We may investigate whether this may lead to suggestions for higher level additions to SeRQL.

Typical, more complex interaction:

```
matching_objects_annotated(SiteId, QueryFC, Objects) :-
        query_select_all_short_fields_string(RecordMode,QAll),
add_queries_get_rows(SiteId, QAll, QueryFC, Rows),
(st_debugging
        ->    rec_short_fields_unique(Rows, SiteId, RecordMode, [], Objects1)
        ;     Objects1 = Rows
        ).
add_queries_get_rows(SiteId, QAll, QueryFC, Rows) :-
sformat(SAll, QAll, [SiteId]),
construct_query_othersl(QueryFC, QL1),
mf_ns(NS),append(QL1, [NS], QL),
concat_atom([SAll|QL], Query),
ssm_query_bag(SiteId, Query, Rows).

query_select_all_short_fields_string(collections,
'select  Rec, RecTitle, RecThumb, CollSpec
from {SiteId} rdfs:label {"~w"};
            rdf:type {mfs:SiteSetup};
            mfs:collection-spec {CollSpec} mfs:record-type {RT};
            mfs:shorttitle-prop {TitleProp};
    mfs:thumbnail-prop {ThumbProp},
```

```
{Rec} rdf:type {RT}; TitleProp {RecTitle};ThumbProp {RecThumb}').
```

TBD: Either explain more of what takes place and show a typical result of the SeRQL query or take out!

8.4.2 HTML code generation

We chose to use the SWI-Prolog html_write library for our HTML-code generation after some evaluation of alternatives. The way the DCG-solution deals with opening and closing tag consistency seemed attractive. Although we have no extensive experience with alternatives, the html_write library proved to be very convenient.

There are three distinct kinds of web pages the multi faceted browser generates (following the Flamenco approach and wording**TBD: rephrase**):

Beginning: The page shown when entering the portal: An initial set of choices is presented along which the user can start searching,

Middle Game The page that shows an overview of (part of) the elements that fall under the current refinement and which may be selected individually, but also lists of choices**TBD: wording** along which the search can be refined.

End Game The page that shows the selected element of the collection with all details available.

These three templates share a common header. The generation of the Facet choice html code is shared between the first two pages.

The web page geration component consists of some 140 DCG rules, part of which are simple list traversing rules such as

```
objectstr([],_O, _Cols,_Perc,_Args) --> [].
objectstr([Objects|ObjectsList], Offset, Cols, Perc,Args) -->
html(tr(valign(top),\objectstd(Objects, Offset, Perc, Args))),
        { Offset1 is Offset + Cols },
objectstr(ObjectsList, Offset1, Cols, erc, Args).
```

a detail of the html code generation for presenting the matching objects in an HTML table.

Another example, the DCG-rule that generates the HTML code for one single facet:

```
mffacet(Tree, Args) -->
        {
            mffacetheadercode(Tree, Args, SubFacets, FData, FCode,
                              FacetClass, Ql, Options)
        },
        html(table([border(0),class(FacetClass), cellspacing(0),
                            cellpadding(0), width('100%')],
                           [\facetheadertr(FData,Options),
                            \facetstr(SubFacets,Ql,FCode,Options)])).
```

The DCG code generation library has not been used by use for such a long time that we developed higer order meta-constructs (cf. maplist) defining for example HTML tables.

Of course, DCG rules were reused for reoccurring HTML code parts, such as generating identical header code for the three kinds of web pages.

TBD: Where oh where? The design and implementation of the web server was such that practically all interaction with the user was stateless: User request details were completely encoded in the URL: The Root of the URL determines the configuration, the URL query parameters, the current user refinement plus action necessary.

A configuration defines what collections are part of the setup plus what facets are shown and very specific to our use case: what mappings are we taking into consideration.

This information is completely propagated via Prolog call arguments. This proved to make debugging very easy. If a bug was really hard to find, instead of activating the web server we simply called the SWI-Prolog reply/1 callback after setting some spypoint(s).

```
reply(Request) :-
        select(path(Path), Request, Request1),
        options(Options),
        reply(Path, Request1,Options).
tstsesame :-
tstsetup,
Options1 = [protocol='debuggingresult.html',sesamestore=default],
Options = [httpargs=[swi=1]|Options1],
Query = 'at:http_058_047_047www.telin.nl_047rdf_047topia_035Term27945',
reply('/SINGLEVIEW-ARIA-E-ALL', [search([q=Query,index=12])], Options).
```

TBD: Too detailed ad hoc example?

9 Conclusion

References

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., Eds. 2003. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Bechhofer, S., Möller, R., and Crowther, P. 2003. The DIG description logic interface. In *International Workshop on Description Logics (DL2003)*. Rome.

Broekstra, J., Kampman, A., and van Harmelen, F. 2002. Sesame: An architecture for storing and querying rdf and rdf schema. In *Proc. First International Semantic Web Conference ISWC 2002, Sardinia, Italy*. LNCS, vol. 2342. Springer-Verlag, 54–68.

Cabeza, D. and Hermenegildo, M. V. 2003. Distributed WWW programming using (ciao-)prolog and the piLLoW library. *CoRR cs.DC/0312031*.

Demoen, B. 2002. Dynamic attributes, their hProlog implementation, and a first evaluation. Report CW 350, Department of Computer Science, K.U.Leuven, Leuven, Belgium. oct. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW350.abs.html.

Deransart, P., Ed-Dbali, A., and Cervoni, L. 1996. *Prolog: The Standard*. Springer-Verlag, New York.

Haarslev, V. and Möller, R. 2001. Description of the racer system and its applications. In *Proceedings of the International Workshop on Description Logics (DL-2001)*. Stanford, USA, 132–141.

Huang, Z. and Stuckenschmidt, H. 2005a. Reasoning with multiversion ontologies. Project Report Deliverable D3.5.1, SEKT.

Huang, Z. and Stuckenschmidt, H. 2005b. Reasoning with multiversion ontologies: a temporal logic approach. In *Proceedings of the 2005 International Semantic Web Conference (ISWC2005)*.

Huang, Z., van Harmelen, F., and ten Teije, A. 2005. Reasoning with inconsistent ontologies. In *Proceedings of the International Joint Conference on Artificial Intelligence - IJCAI'05*.

Huang, Z. and Visser, C. 2004. Extended DIG description logic interface support for PROLOG. Deliverable D3.4.1.2, SEKT.

Leth, L., Bonnet, P., Bressan, S., and Thomsen, B. 1996. Towards ECLiPSe agents on the internet. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*.

Schlobachm, S. and Huang, Z. 2005. Inconsistent ontology diagnosis: Framework and prototype. Deliverable D3.6.1, SEKT.

Wielemaker, J. 2003. Native preemptive threads in SWI-Prolog. In *Practical Aspects of Declarative Languages*, C. Palamidessi, Ed. Springer Verlag, Berlin, Germany, 331–345. LNCS 2916.

Wielemaker, J. 2005. An optimised semantic web query language implementation in prolog. In *ICLP 2005*, M. Baggrielli and G. Gupta, Eds. Springer Verlag, Berlin, Germany, 128–142. LNCS 3668.

Wielemaker, J., Schreiber, G., and Wielinga, B. 2003. Prolog-based infrastructure for RDF: performance and scalability. In *The Semantic Web - Proceedings ISWC'03, Sanibel Island, Florida*, D. Fensel, K. Sycara, and J. Mylopoulos, Eds. Springer Verlag, Berlin, Germany, 644–658. LNCS 2870.