TLP **8** (3): 363–392, 2008. © 2008 Cambridge University Press
doi:10.1017/S1471068407003237 First published online 20 March 2008 Printed in the United Kingdom

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 AND LOURENS VAN DER MEIJ

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

submitted 28 April 2006; revised 23 August 2007; accepted 18 October 2007

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 a prime candidate for programming robust Web services. Although Prolog is commonly seen as a component in a Web application that is either embedded or communicates using a proprietary protocol, we propose an architecture where Prolog communicates to other components in a Web application using the standard HTTP protocol. By avoiding embedding in external Web servers, development and deployment become much easier. To support this architecture, in addition to the transfer protocol, we must also support parsing, representing and generating the key Web document types such as HTML, XML and RDF. This article motivates the design decisions in the libraries and extensions to Prolog for handling Web documents and protocols. The design has been guided by the requirement to handle large documents efficiently. The described libraries support a wide range of Web applications ranging from HTML and XML documents to Semantic Web RDF processing. The benefits of using Prolog for Web-related tasks are illustrated using three case studies.

KEYWORDS: Prolog, HTTP, HTML, XML, RDF, DOM, Semantic Web

1 Introduction

The Web is an exciting place offering new opportunities to artificial intelligence, natural language processing and logic programming. Information extraction from the Web, reasoning in Web applications and the Semantic Web (SW) are just a few examples. We have deployed Prolog in Web-related tasks over a long period. As most of the development on SWI-Prolog takes place in the context of projects that require new features, the system and its libraries provide extensive support for Web programming.

There are two views on deploying Prolog for Web-related tasks. In the most commonly used view, Prolog acts as an embedded component in a general Web processing

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

environment. In this role, it generally provides 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 applications. Components generally exchange XML if used as part of the backend or middleware services and HTML if used in the presentation layer.

The latter view in our vision is more attractive. Using HTTP and XML over HTTP, the service is cleanly isolated using standard protocols rather than proprietary 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. Using HTTP, automatic testing of the Prolog components can be done using any Web-oriented test framework. HTTP allows Prolog to be deployed in any part of the service architecture, including the realisation of complete Web applications in one or more Prolog processes.

When deploying Prolog in a Web application using HTTP, we must not only implement the HTTP transfer protocol but also support parsing, representing and generating the important document types used on the Web, especially HTML, XML and RDF. Note that, being widely used open standards, supporting these document types is also valuable outside the context of Web applications.

This article gives an overview of the Web infrastructure we have realised. Given the range of libraries and Prolog extensions that facilitate Web applications, we cannot describe them in detail. Details on the library interfaces can be found in the manuals available from the SWI-Prolog Web site.[1] Details on the implementation are available in the source distribution. The aim of this article is to give an overview of the required infrastructure to use Prolog for realising Web applications where we concentrate on scalability and performance. We describe our decisions for representing Web documents in Prolog and outline the interfaces provided by our libraries.

The benefits of using Prolog for Web-related tasks are illustrated using three case studies: (1) SeRQL, an RDF query language for metadata management, retrieval and reasoning; (2) XDIG, an eXtended DLG Description Logic interface, which provides ontology management and reasoning by processing DIG XML documents and communicating to external DL reasoners; and (3) faceted browser on SW databases integrating metadata from multiple collections of artworks. This case study serves as a complete SW application serving the end user.

This article is organised as follows. Sections 2 to 4 describe reading, writing and representation of Web-related documents. Section 5 describes our HTTP client and server libraries. Section 6 describes extensions to the Prolog language that facilitate use in Web applications. Sections 7 to 9 describe the case studies.

2 Parsing and representing XML and HTML documents

The core of the Web is formed by document standards and exchange protocols. Here we describe tree-structured documents transferred as SGML or XML. HTML,

[1] `http://www.swi-prolog.org`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

⟨*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*⟩, ⟨*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 list of terms of type ⟨*x*⟩.

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 *content*, a sequence of subelements and text (data).

XML is a rationalisation of SGML using the same tree model, but removing many rarely used features as well as abbreviations that were introduced in SGML to make the markup easier to type and read for humans. XML documents are used to represent text using custom application-oriented tags as well as a serialisation format 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 on the basis of the SP parser.[2] A stable Prolog term representation for SGML/XML trees plays a similar role as the DOM (*Document Object Model*) representation in use in the object-oriented world. The term-structure we use is described in Figure 1. Some issues have been subject to debate.

- 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 as motivated in Section 6.2. At the same time, SWI-Prolog stacks are limited to 128 MB each. 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. Using lists of character codes is another possibility adopted by both PiLLoW (Gras and Hermenegildo 2001) and ECLiPSe (Leth *et al.* 1996). Two observations make lists less attractive: lists use two cells per character, while practical experience shows text is frequently processed as a unit only. For (HTML) text documents, we profit from the compact representation of atoms. For XML documents representing serialised data structures, we profit from frequent repetition of the same value.

[2] `http://www.jclark.com/sp/.`

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

- Attribute values of multi-value 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 always true, 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 different floating point number notations used are lost after conversion. Moreover, Prolog systems with bounded arithmetic may not be able to represent all values. Still, automatic conversion is useful in many applications, especially those involving serialised data structures.
- Attribute values are represented as *Name=Value*. Using *Name*(Value) is an alternative. The *Name=Value* representation was chosen for its similarity to the SGML notation and because it avoids the need for univ (`=..`) for processing argument lists.

2.1 Implementation

The SWI-Prolog SGML/XML parser is implemented as a C-library that has been built from scratch to create a lightweight parser. Total source is 11,835 lines. The parser provides two interfaces. Most natural to Prolog is **load_structure**(*+Src, -DOM, +Options*), which parses a Prolog stream into a term as described above. Alternatively, **sgml_parse/2** provides an *event-based* parser making callbacks on Prolog for the SGML events. The callback mode can deal with unbounded documents in streaming mode. It can be mixed with the term-creation mode, where the handler for *begin* calls the parser to create a term representation for the content of the element. This feature is used to process long files with a repetitive record structure in limited memory. Section 4.1 describes how this is used to process RDF documents.

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

3 Generating Web documents

There are many approaches to generating Web pages from programs in general and Prolog in particular. We believe the preferred choice depends on various aspects.

- How much of the document is generated from dynamic data and how much is static? Pages that are static except for a few strings are best generated from a template using variable substitution. Pages consisting of a table generated from dynamic data are best entirely generated from the program.
- For program-generated pages, we can choose between direct printing and generating using a language-native syntax, for example, `format('<b>bold</b>')` or `print_html(b(bold))`. The second approach can guarantee well-formed output, but the first requires the programmer to learn about **format/3** only.
- Documents that contain a significant static part are best represented in the markup language where special constructs insert program-generated parts.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

A popular approach implemented by PHP[3] and ASP[4] is to add a reserved element such as ⟨*script*⟩ and use the SGML/XML *programming instruction* written as `<?...?>`. The obvious name PSP (Prolog Server Pages) is in use by various projects taking this approach.[5] Another approach is PWP[6] (Prolog Well-formed Pages). It is based on the principle that the source is well-formed XML and interacts with Prolog through additional attributes. Output is guaranteed to be well-formed XML. Our infrastructure does not yet include any of these approaches.

- Page *transformation* is realised by parsing the original document into its tree representation, managing the tree and writing a new document from the tree. Managing the source text directly is not reliable, as due to character encoding choice, entity usage and SGML abbreviations, there are many different source texts that represent the same tree. The **load_structure/3** predicate described in Section 2 together with output primitives from the library `sgml_write.pl` provides this functionality. The XDIG case study described in Section 8 follows this approach.

3.1 Generating documents using DCG

The traditional method for creating Web documents is using print routines such as **write/1** or **format/2**. Although simple and easily explained to novices, the approach has serious drawbacks from a software engineering point of view. In particular, the user is responsible for HTML quoting, character encoding issues and proper nesting of HTML elements. Automated validation is virtually impossible using this approach.

Alternatively, we can produce a DOM term as described in Section 2 and use the library `sgml_write.pl` 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 (Section 3.2) to control the complexity. In our opinion, the result is not optimal due to the unnatural order of statements as illustrated in Figure 2. PiLLoW partly overcomes this shortcoming by defining a large number of 'utility terms' that are translated in a special way, as discussed in Section 6.2 of Gras and Hermenegildo (2001).

We introduced a DCG rule html//1.[7] This rule translates proper trees into a list of high-level HTML/XML commands that are handed to **html_print/1** to realise

[3] `www.php.net.`
[4] `www.microsoft.com.`
[5] `http://www.prologonlinereference.org/psp.psp`,
`http://www.benjaminjohnston.com.au/template.prolog?t=psp`,
`http://www.ifcomputer.com/inap/inap2001/program/inap_bartenstein.ps`
[6] `http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl.`
[7] The notation ⟨*name*⟩//⟨*arity*⟩ refers to the grammar rule ⟨*name*⟩ with the given ⟨*arity*⟩, and consequently the predicate ⟨*name*⟩ with arity ⟨*arity*⟩+2.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

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

Fig. 2. Building PiLLoW terms.

```
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.

proper quoting, character encoding and layout. The intermediate format is of no concern to the user and is similar in structure to the PiLLoW representation without using environments. Generated from the tree representation, however, consistent opening and closing of elements is guaranteed. In addition to variable substitution that is provided by Prolog we allow calling rules. Rules are invoked by a term *Rule* embedded in the argument of html//1. 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.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

Fig. 4. The html//1 argument specification.

In our current implementation 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 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. An overview of the argument to html//1 is given in figure 4.

3.2 Comparison with PiLLoW

The PiLLoW library (Gras and Hermenegildo 2001) is a well-established framework for Web programming based on Prolog. PiLLoW defines **html2terms/2**, converting between an HTML string and a document represented as a Herbrand term. There are fundamental differences between PiLLoW and the primitives described here.

- PiLLoW creates an HTML document from a Herbrand term that is passed to **html2terms/2**. Complex terms are composed of partial terms passed as Prolog variables. Frequent HTML constructs are supported by using reserved terms based on dedicated processing. We use DCGs and the *Rule* construct, which makes it eminent which terms directly refer to HTML elements and which function as a 'macro'. In addition, the user can define application-specific reusable fragments in a uniform way.
- 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 illustrated in Figure 5. Having a canonical DOM representation greatly simplifies processing parsed HTML documents.

4 RDF documents

While the data model of both HTML and XML is a tree structure with attributes, the data model of the SW RDF[8] language consists of {*Subject*, *Predicate*, *Object*}

[8] `http://www.w3.org/RDF/.`

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
[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.

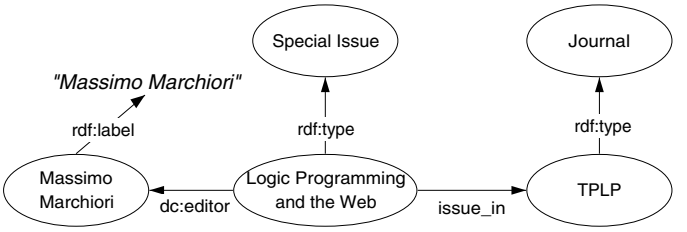


Fig. 6. Sample RDF graph. Ellipses are vertices representing URIs. Quoted text is a literal. Edges are labelled with URIs.

triples. Both *Subject* and *Predicate* are *URI*s.[9] *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 labelled with a URI (the *Predicate*) and each vertex is either a URI or a literal. Literals have no outgoing edges. Figure 6 illustrates this point.

A number of languages are layered on top of the RDF triple model. RDFS provides a frame-based representation. The OWL-dialects[10] provide three increasingly complex Web ontology languages. SWRL[11] is a proposal for a rule language. The W3C standard for exchanging these triple models is an XML application known as RDF/XML.

As there are multiple XML tree representations for the same triple set, RDF documents cannot be processed at the level of the XML-DOM as described in Section 2. A triple- or graph-based structure is the most natural choice for representating 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 languagesis equivalence test, using a Prolog atom is an obvious choice. One might consider using a term ⟨*namespace*⟩:⟨*localname*⟩, but given that decomposing a URI into its namespace and localname is only relevant during I/O we consider this an inferior choice. The RDF library comes with a compile time rewrite mechanism based on **goal_expansion/2** that allows for writing resources in Prolog sourcetext

[9] URI: *Uniform Resource Identifier* is like a URL, but need not refer to an existing resource on the Web.
[10] `http://www.w3.org/2004/OWL/`.
[11] `http://www.w3.org/Submission/SWRL/`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

Fig. 7. RDF types in Prolog.

```
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**.

as ⟨*ns*⟩:⟨*local*⟩. Literals are expressed as **literal**(*Value*). The full type description is shown in Figure 7.

The typical SW use scenario is to 'harvest' triples from multiple sources and collect them in a database before reasoning with them. Prolog can represent data as a Herbrand term on the stack or as predicates in the database. Given the relatively static nature of the RDF data as well as desired access from multiple threads, using the Prolog database is the most obvious choice. Here we have two options. One is the 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 chosen for **rdf/3** because it supports queries with uninstantiated predicates better and a single predicate is easier to manage than an unbounded set of predicates with unknown names.

4.1 Input and output of RDF documents

The RDF/XML parser is realised as a Prolog library on top of the XML parser described in Section 2. It has two interfaces as the XML parser. The predicate **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 callback/convert mode of the XML parser to process the RDF file one *description* (record) at a time, calling *Action* with a list of triples extracted from the description. Figure 8 illustrates 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**.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
<Painting rdf:about="...">
  <dimension>
     <Dimension width="45" height="50"/>
  </dimension>
</Painting>
```

Fig. 9. Blank node to express the compound dimension property.

In addition to named URIs, RDF resources can be *blank nodes*. A blank node (short *bnode*) is an anonymous resource that is created from an in-lined description. Figure 9 describes the dimensions of a painting as a compound instance of class *Dimension* with width and height properties. The *Dimension* instance has no URI. Our parser generates an identifier that starts with a double underscore, followed by the source and a number. The double underscore is used to identify bnodes. Source and number are needed to guarantee that the bnode is unique.

The parser from XML to RDF triples covers the full RDF specification, including Unicode handling, RDF datatypes and RDF language tags. The Prolog source is composed of 1,788 lines. It processes approximately 9,000 triples per second on an AMD 1600+ based computer. Implementation details and evaluation of the parser are described in Wielemaker *et al.* 2003.[12]

We have two libraries for writing RDF/XML. One, **rdf_write_xml**(*+Stream, +Triples*), provides 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 described in Section 4.2 and writes a database directly to a file or a stream. The first (**rdf_write_xml/2**) is used to exchange computed graphs to external programs using network communication, while the second (**rdf_save/2**) is used to save modified graphs back to file. The resulting code duplication is unfortunate, but unavoidable. Creating a temporary graph in a 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 are also considerably slower than a direct write.

4.2 Storage of RDF

Assuming the 'harvesting' use case, we need to implement a predicate **rdf**(*?S,?P,?O*). Indexing of the database is crucial for good performance. Table 1 illustrates the calling pattern from a real-world application counting 4 million triples. In addition Figure 7 describes our data. The RDF store was developed in the context of projects, which formulated the following requirements.

- Up to at least 10 million triples on 32-bit hardware.
- Fast graph traversal using any instantiation pattern.
- Case-insensitive search on literals.
- Prefix search on literals for completion in the user interface (UI).

[12] The parser described there did not yet support RDF datatypes and language tags, nor Unicode.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

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

- 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 on the basis of its source.
- Fast load/save of current state.

Our first version of the database used the Prolog database with secondary tables to improve indexing. As requirements pushed us against the limits of what is achievable in a 32-bit address space, we decided to implement the low-level store in C. Profiting from the known uniform structure of the data, we realised about two times more compact storage with better indexing than using a pure Prolog approach. We took the following design decisions for the C-based storage module:

- The RDF *predicates* are represented as unique entities and organised according to the rdfs:subPropertyOf relation in multiple hierarchies. The root of each 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 as the root.
- Literals are kept in an AVL tree, sorted as case-insensitive and case-preserving search (e.g. AaBb...). Numeric literals preceed all nonnumeric and are kept sorted on their numeric value. Storing literals in a separate sorted table allows for indexed search for prefixes and numeric values. It also allows for monitoring creation and destruction of literals to maintain derived tables such as stemming or double methaphone (Philips 2000) on the basis of **rdf_monitor/3** described below. The space overhead of maintaining the table is roughly cancelled by avoiding duplicates. Experience on real data ranges between −5% and +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 avoid both duplication of data and table lookup. We consider this as a crucial aspect.
- 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 +,+,+ and +,−,+. Queries using the pattern +,−,+ are rare. Fully

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

instantiated queries internally use the pattern +,+,−, assuming few values on the same property. Considering experience with real data, we will probably add a +,+,+ index in the future. The unindexed 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. If the transaction is ready for commit, the thread denies access to new readers and waits for all readers to vanish before updating the database. 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* iscalled. Modifying actions inside a transaction are called during the commit. Modifications by the monitors are collected in a new transaction that 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 serialised.

Although the 9,000 triples per second of the RDF/XML parser ranks it among the fast parsers, loading 10 million triples takes nearly 20 minutes. For this reason, we developed a binary format. The format is described in Wielemaker *et al.* (2003), and loads approximately 10 times faster than RDF/XML while using about the same space. The format is independent from byte order and word length, supporting both 32- and 64-bit hardware.

Persistency is achieved through the library `rdf_persistency.pl`, which uses **rdf_monitor/3** to maintain a set of files in a directory. Each source known to the database is represented by two files, one file representing the initial state using the quick-load binary format and one file containing Prolog terms representing changes, called the *journal*.

4.3 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 Section 7. 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*)

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
% triples                % entailment interface % RDFS interface
mary type woman.         ?-rdf(mary,type, X).   ?-rdfs_individual_of(mary, X).
woman type Class.        X = woman;             X = woman;
woman subClassOf human.  X = human;             X = human;
human type Class.        No                     No
```

Fig. 10. Different interface styles for RDFS.

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 forward reasoning, deducing new triples until this is no longer possible or by a combination of backward 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 10 illustrates the difference in these approaches.

4.4 Experience

The RDF infrastructure is used in two of the three case studies described at the end of this article. RDF has a natural representation in Prolog, either as a list of terms or as a pure predicate. Prolog nondeterminism greatly simplifies querying the database. Transitivity is easily expressed using recursion. However, as cycles in SW graphs are allowed and frequently, such algorithms must be protected against them. Cycle detection complicates the code and harms performance. We plan to investigate tabling (Ramakrishnan *et al.* 1995) to improve on this situation.

Although designed as RDF store for SW-based projects, the infrastructure is also commonly used to create RDF documents from other sources as well as for filtering and reorganising RDF documents. In the e-culture project,[13] it has been used to convert WordNet (Miller 1995) from its Prolog representation and the Getty thesauri[14] from XML (4 GB data) into RDF.

5 Supporting HTTP

HTTP, or HyperText Transfer Protocol, is the key W3C standard protocol for exchanging Web documents. All browsers and Web servers 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, etc.

[13] `e-culture.multimedian.nl`.
[14] `http://www.getty.edu/research/conducting_research/vocabularies/`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

Adding HTTP support in Prolog, we must consider both the client- and server-side reasoning. In both cases, our choice is between doing it in Prolog or reusing an existing application or library by providing an interface for it. We compare our work with 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 client consist of 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 reusing external code. Our core server library counts 1,784 lines.
- A single computer can only host one server at port 80 used by default for public HTTP. Using an alternate port for middleware and storage tier components is no problem, but use as a public server often conflicts with firewall or proxy settings. This can be solved using a proxy server such as the Apache *mod_proxy*[15] configured as *reverse proxy*.
- Servers by definition introduce security risks. Administrators are reluctant to see non-proven software in the role of a public server. Using a proxy as above also reduces this risk, especially if the proxy blocks malformed requests.

Despite these observations, we consider, like the ECLiPSe team, a pure Prolog-based server worthwhile. As argued in Section 6.1, many Prolog Web applications profit from using state stored in the server. Large resources such as WordNet (Miller 1995) cause long start-up times. In such cases, the use of CGI (Common Gateway Interface) is not appropriate as a new copy of the application starts for each request. PiLLoW resolves this issue by using *Active Modules*, where a small CGI application talks to a continuously running Prolog server using a private protocol. Using a Prolog HTTP server and optionally a reverse proxy has the same benefits, but based on a standard protocol, it is much more flexible.

Another approach is embedding Prolog in another server framework such as the Java-based Tomcat server. Although feasible, embedding non-Java based Prolog 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 to 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 Figure 11.

5.1 HTTP client libraries

We support two clients. The first is a lightweight client that supports only the HTTP GET method by means of **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

[15] `http://httpd.apache.org/docs/1.3/mod/mod_proxy.html`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

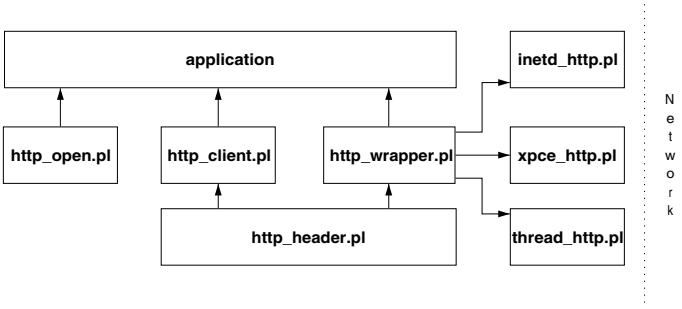


Fig. 11. Module dependencies of the HTTP library.

```
?- 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. 12. Fetching an HTML document.

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`, provides support for HTTP POST and a plug-in interface that allows for installing handlers for documents of specified MIME-types. It shares `http_header.pl` with the server libraries for DCG-based creation and parsing of HTTP headers. Currently provided plug-ins include `http_mime_plugin.pl` to handle multi-part MIME messages and `http_sgml_plugin.pl` for automatically parsing HTML, XML and SGML documents. Figure 12 shows the code for fetching a URL and parsing the returned HTML document it into a Prolog term as described in Section 2.

Both the PiLLoW and ECLiPSe approaches 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**, **http_post/4**). This interface avoids potentially large intermediate data structures and allows for processing unbounded documents.

5.2 The HTTP server library

Both to simplify reuse of application code and to make it possible to use the server without committing to a large infrastructure, we adopted the reply strategy of the CGI protocol, where the handler writes a page consisting of an HTTP header followed by the document content. Figure 13 provides a simple example that returns

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
:- 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).
```

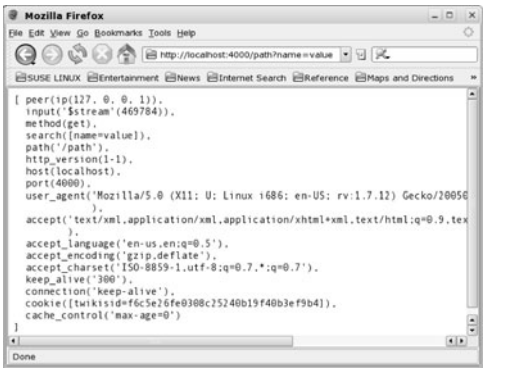


Fig. 13. A simple HTTP server. The right window shows the client and the format of the parsed request.

the request data to the client. By importing `thread_http.pl`, we implicitly selected the multi-threaded server model. Other models provided are `inetd_http`, causing the (Unix) inet daemon to start a server for each request, and `xpce_http`, which uses I/O multiplexing realising 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 a Prolog exception. Recognised replies are defined by the predicate **http_reply**(*+Reply, +Stream, +Header*). For example, to indicate that the user has no access to a page, we must use the following call.

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

Failure of the handler raises a '404 existence error' reply, while exceptions other than the ones described above raise a '500 Server error' reply.

5.2.1 Form parameters

The library `http_parameters.pl` defines **http_parameters**(*+Request, ?Parameters*) to fetch and type-check parameters transparently for both GET and POST requests. Figure 14 illustrates the functionality. Parameter values are returned as atoms. If large documents are transferred using a POST request, the user may wish to revert to **http_read_data**(*+Request, -Data, +Options*) underlying **http_get/3** to process arguments using plugins.

5.2.2 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*),

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

Fig. 14. Fetching HTTP form data.

```
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(_Done).

session_loop(State) :-                          % Session thread
        thread_get_message(handle(Request, Sender, CGIOut)),
        next_state(Request, State, NewState, CGIOut).
        thread_send_message(Sender, done).
```

Fig. 15. 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. Error handling for readability of the example is omitted.

http_session_retract(*?Data*) and **http_session_data**(*?Data*). In the current version, the data associated with sessions that have timed out are 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 required the user must create a thread that represents the session and set up communication from the HTTP worker thread to the session thread. Figure 15 illustrates the idea.

5.2.3 Evaluation

The presented server infrastructure is currently used by many internal and external projects. Coding a server is very similar to writing CGI handlers and running in the interactive Prolog process is much easier to debug. As Prolog is capable of reloading source files in the running system, handlers can be updated while the server is

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

Table 2. *HTTP performance executing a trivial query 10,000 times**.

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

*Times are in seconds. Localhost, dual AMD 1600+ running SuSE Linux 10.0

running. Handlers running during the update are likely to die on an exception though. We plan to resolve this issue by introducing read/write locks. The protocol overhead of the multi-threaded server is illustrated in Table 2.

6 Enabling extensions to the Prolog language

SWI-Prolog has been developed in the context of projects, many of which caused the development to focus on managing Web documents 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 crucial for scalable and comfortable deployment of Prolog as an agent in a Web centred world.

6.1 Multi-threading

Concurrency is necessary for applications for the following reasons:

- Network delays may cause communication of a single transaction to take very long compitition time. It is not acceptable if such incidents block access for other clients. This can be achieved using multiplexed I/O, 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 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 mentioned 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. The advantages of this are described in Szeredi *et al.* (1996), using the or-parallel Aurora to serve multiple clients. This is particularly interesting for accessing the RDF database described in Section 4.2.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

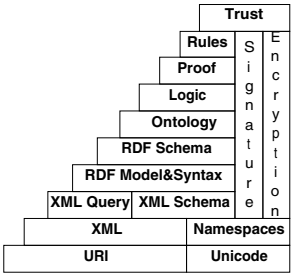


Fig. 16. The Semantic Web layer cake by Tim Burners Lee.

6.2 Atoms and Unicode support

Unicode[16] is a character-encoding system that assigns unique integers (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 can handle documents from different scripts as well as documents that contain multiple scripts in a single uniform representation, an important feature in applications processing Web data. Traditional HTML applications commonly insert special symbols through entities such as the copyright (©) sign, Greek and mathematical symbols, etc. Using Unicode we can represent all entity values as plain text. As illustrated in the famous *SW layer cake* in Figure 16, Unicode is at the heart of the SW.

HTML documents can be represented using Prolog strings because Prolog integers can represent all Unicode code points. As we have claimed in Section 2, however, using Prolog strings is not the most obvious choice. XML attribute names and values can contain arbitrary Unicode characters, which require the unnatural use of strings for these as well. If we consider RDF, URIs can have arbitrary Unicode characters[17] and we want to represent URIs as atoms to exploit compact storage as well as fast equivalence testing. Without Unicode support in atoms we would have to encode Unicode in the atom using escape sequences. All this patchwork can be avoided if we demand the following properties for Prolog atoms:

- Atoms represent text in Unicode
- Atoms have no limit on their length
- The Prolog implementation allows for a large number of atoms, both to represent URIs and to represent text in HTML/XML documents. SWI-Prolog's limit is 2^{25} (32 million).
- Continuously running servers cannot allow memory leaks, and therefore processing dynamic data using atoms requires atom garbage collection.

[16] `http://www.Unicode.org/`.
[17] `http://www.w3.org/TR/rdf-concepts/#section-Graph-URIref`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

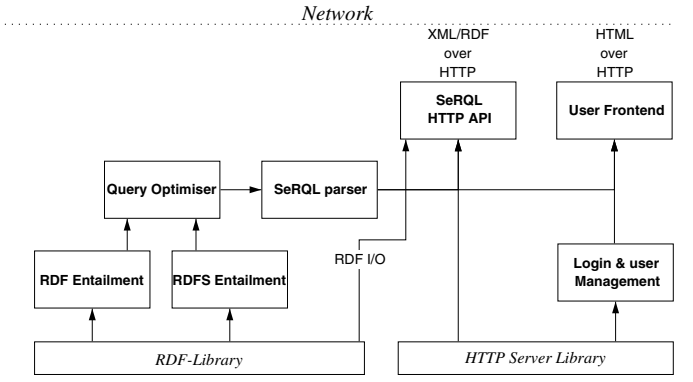


Fig. 17. Module dependencies of the SeRQL system. Arrows denote 'imports from' relations.

```
:- 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. 18. RDF entailment module.

7 Case study: A SW query language

In this case study, we describe the SWI-Prolog SeRQL implementation.[18] SeRQL is an RDF query language developed as part of the Sesame project[19] (Broekstra *et al.* 2002). SeRQL uses HTTP as its access protocol. Sesame consists of an implementation of the server as a Java servlet 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 has contributed significantly to the development of the infrastructure. Figure 17 lists the main components of the server.

The *entailment* modules are plug-ins that implement the entailment approach to RDF reasoning described in Section 4.3. They implement **rdf/3** as a pure predicate, adding implicit triples to the raw triples loaded from RDF/XML documents. Figure 18 shows the somewhat simplified entailment module for RDF. The multifile

[18] `http://www.swi-prolog.org/packages/SeRQL`.
[19] `http://www.openrdf.org`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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

Fig. 19. 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).

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 reloading allow for updating such reasoning modules on the live server.

The SeRQL parser is a DCG-based parser translating a SeRQL query text into a compound goal calling **rdf/3** and predicates from the SeRQL runtime library that provide comparison and functions built into the SeRQL language. The resulting control structure is passed to the query optimiser (Wielemaker 2005) that 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 19 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-centred portal consists of HTML pages with forms to administer the server as well as view statistics, load and unload documents and run SeRQL queries interactively, presenting the result as an HTML table. Dynamic pages are generated using the `html_write.pl` library described in Section 3.1. 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 of 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 uses `html_write.pl`. The RDF/XML format uses **rdf_write_xml/2** described in Section 4.1. Both **rdf_write_xml/2** and the other XML output format use straight calls **format/3** to write the document, where quoting values is realised by quoting primitives provided by the SGML/XML parser described in Section 2. Using direct writing instead of techniques described in Section 3.1 avoids potentially large intermediate data structures and is not very complicated given the very simple structure of the documents.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

7.1 Evaluation

The SeRQL server and the SWI-Prolog library development is too closely integrated to be used in an evaluation of the functionality provided by the Web-enabling libraries. We compared our server with Sesame, written in Java. The source code of the Prolog-based server is 6,700 lines,compared with 86,000 for Sesame. As both systems have very different coverage in functionality and can reuse libraries at different levels, it is hard to judge these figures. Both answer trivial queries in approximately 5 ms on a dual AMD 1600+ PC running Linux 2.6. On complex queries, the two systems perform very differently. Sesame's forward reasoning makes it to 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, optimising 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** allows for a trivial translation of the query to a non-deterministic program that produces the answers on backtracking.

The server depends only on the standard SWI-Prolog distribution, and therefore runs unmodified on all systems supporting SWI-Prolog. It hasbeen tested on Windows, Linux and MacOS X.

All infrastructure described herein 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** from the library `sgml_write.pl`.

8 Case study: XDIG

In Section 7, we have discussed the case study how SWI-Prolog is used for an RDF query system, that is, a metadata management and reasoning system. In this section, we describe a Prolog-powered system for ontology management and reasoning based on Description Logics (DL). DL has greatly influenced the design of the W3C ontology language OWL. The DL community, called DIG (DL Implementation Group) have developed astandard for accessing DL reasoning engines called the DIG description logic interface[20] (Bechhofer *et al.* 2003), DIG interface for short. Many DL reasoners such as Racer (Haarslev and Möller 2001) and FACT (Horrocks 1999) support the DIG interface, allowing for the construction of highly portable and reusable DL components or extensions.

In this case study, we describe XDIG that has been implemented on top of the SWI-Prolog Web libraries. The DIG interface uses an XML-based messaging protocol on top of HTTP. Clients of a DL reasoner communicate by means of HTTP POST requests. The body of the request is an XML-encoded message that

[20] `http://dl.kr.org/dig/`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
<equalc>
  <catom name='mad+cow'/>
  <and>
    <catom name='cow'/>
    <some>
      <ratom name='eats'/>
      <and>
        <catom name='brain'/>
        <some>
          <ratom name='part+of'/>
          <catom name='sheep'/>
</some></and></some></and></equalc>
```

Fig. 20. a DIG statement on MadCow.

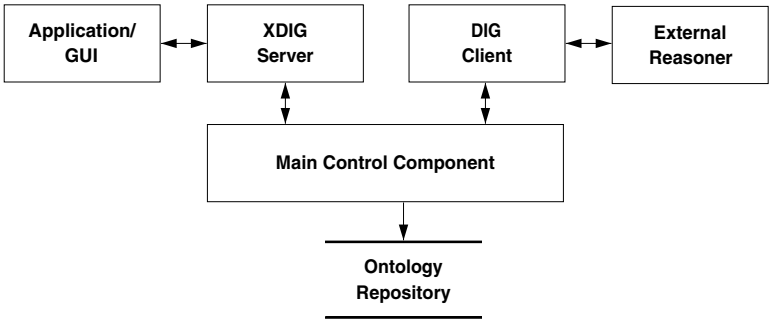


Fig. 21. Architecture of XDIG.

corresponds to the DL concept language. While OWL is based on the triple model described in Section 4, DIG statements are grounded directly in XML. Figure 20 shows a DIG statement that defines the concept *MadCow* as a cow that eats brains, part of sheep.

8.1 Architecture of XDIG

The XDIG libraries form a framework to build DL reasoners that have additional reasoning capabilities. XDIG serves as a regular DL reasoner via its corresponding DIG interface. An intermediate XDIG server can make systems independent from application-specific characteristics. A highly decoupled infrastructure significantly improves the reusability and applicability of software components.

The general architecture of XDIG is shown in Figure 21. It consists of the following components:

XDIG server: The XDIG server deals with requests from ontology applications. It supports our extended DIG interface, that is, it supports not only standard DIG/DL requests such as 'tell' and 'ask' but also additional processing features such as changing system settings. The library `dig_server.pl` implements the XDIG protocol on top of the Prolog HTTP server described in Section 5.2. The

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
direct_concept_relevant(element(catom, Atts, _), Concept) :-
    memberchk(name=Concept, Atts).
direct_concept_relevant(element(_, _, Content),  Concept) :-
    direct_concept_relevant(Content, Concept).
direct_concept_relevant([H|T], Concept) :-
    (   direct_concept_relevant(H, Concept)
    ;   direct_concept_relevant(T, Concept)
    ).
```

Fig. 22. The checks on direct_concept_relevant predicate that a concept is referenced by a DIG statement.

predicate **dig_server**(*+Request*) is called from the HTTP server to process a client's *Request* as illustrated in Figure 13. XDIG server developers have to define the predicate **my_dig_server_processing**(*+Data, -Answer, +Options*), where *Data* is the parsed DIG XML requests and *Answer* is term **answer**(*-Header, -Reply*). *Reply* is the XML-DOM term representing the answer to the query.

DIG client: XDIG is designed to rely on an external DL reasoner. It implements a regular DIG interface client and calls the external DL reasoner to access the standard DL reasoning capabilities. The predicate **dig_post**(*+Data, -Reply, +Options*) posts the data to the external DIG server. The predicates are defined in terms of the predicate **http_post/4** and others in the HTTP and XML libraries.

Main control component: The library `dig_process.pl` provides facilities to analyse DIG statements such as finding concepts, individuals and roles, but also decide of *satisfiability* of concepts and consistency. Some of this processing is done by analysing the XML-DOM representation of DIG statements in the local repository, while satisfiability and consistency checking is achieved by accessing external DL reasoners through the DIG client module.

Ontology repository: The Ontology Repository serves as an internal knowledge base (KB), 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 parts from the ontologies to post them to an external DL reasoner and obtain the corresponding answers. This internal KB is also used to store system settings.

As DIG statements are XML based, XDIG stores statements in the local repository using the XML-DOM representation described in Section 2. The tree model of XDIG data has been proved to be convenient for DIG data management.

Figure 22 shows a piece of code from the library XDIG defining the predicate **direct_concept_relevant**(*+DOM, ?Concept*) that checks if a set of *Statements* is directly relevant to a *Concept*, namely, the *Concept* appears in the body of a statement in the list. The predicate **direct_concept_relevant/2** has been used to develop PION for reasoning with inconsistent ontologies and DION for inconsistent ontology debugging.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

8.2 Application

XDIG has been used to develop several DL reasoning services. PION is a reasoning system that deals with inconsistent ontologies[21] (Huang and Visser 2004; Huang *et al.* 2005). MORE is a multi-version ontology reasoner[22] (Huang and Stuckenschmidt 2005). DION is a debugger of inconsistent ontologies[23](Schlobach and Huang 2005). With the support of an external DL reasoner such as Racer, DION can serve as an inconsistent ontology debugger using a bottom-up approach.

9 Case study: Faceted browser on SW database integrating multiple collections

In this case study, we describe a pilot for the STITCH project[24] whose main aim is studying and finding solutions for the problem of integrating controlled vocabularies such as thesauri and classification systems in the Cultural Heritage domain. The pilot consists of the integration of two collections—the Medieval Illuminations of the Dutch National Library (Koninklijke Bibliotheek) and the Masterpieces collection from the Rijksmuseum—and development of a UI for browsing the merged collections. One requirement within the pilot is to use 'standard SW techniques' during all stages, so as to be able to evaluate their added value. An explicit research goal was to evaluate existing 'ontology mapping' tools.

The problem can be split into three main tasks:

- Gathering data, that is, records of the collections and controlled vocabularies they use, and transforming it into RDF.
- Establishing semantic links between the vocabularies using off-the-shelf ontology mapping tools.
- Building a prototype UI to access (search and browse) the integrated collections and experiment with different ways to access them using a Web server.

SWI-Prolog has been used in all three tasks; to illustrate the use of the SWI-Prolog Web libraries in the pilot, in the next section we focus on their application in the prototype UI because it is the largest subsystem using these libraries.

9.1 Multi-faceted browser

Multi-faceted browsing is a search and browse paradigm where a collection is accessed by refining multiple (preferably) structured aspects – called facets – of its elements. For the UI and user interaction, we have been influenced by the approach of Flamenco (Hearst *et al.* 2002). The Multi-faceted browser is implemented in SWI-Prolog. All data are stored in an RDF database, which can be either an external SeRQL repository or an in-memory SWI-Prolog RDF database. The system consists

[21] `http://wasp.cs.vu.nl/sekt/pion`.
[22] `http://wasp.cs.vu.nl/sekt/more`.
[23] `http://wasp.cs.vu.nl/sekt/dion`.
[24] `http://stitch.cs.vu.nl`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
SELECT  Rec, RecTitle, RecThumb, CollSpec
FROM {SiteId} rdfs:label {"ARIATOPS-NONE"};
            mfs:collection-spec {CollSpec} mfs:record-type {RT};
            mfs:shorttitle-prop {TitleProp};
            mfs:thumbnail-prop {ThumbProp},
{Rec} rdf:type {RT}; TitleProp {RecTitle};
      ThumbProp {RecThumb},
{Rec} rdf:type {<http://www.telin.nl/rdf/topia#AnimalPieces>},
{Rec} rdf:type {<http://www.telin.nl/rdf/topia#Paintings>}
USING NAMESPACE skos = <http://www.w3.org/2004/02/skos/core#>,
mfs = <http://www.cs.vu.nl/STITCH/pp/mf-schema#><br>
```

Fig. 23. An example of a SeRQL query, which returns details of records matching two facet values (AnimalPieces and Paintings).

of three components, *RDF-interaction*, which deals with RDF-database storage and access, *HTML-code generation*, for the creation of Web pages, and the *Web server* component, implementing the HTTP server. They are discussed in the following sections.

9.1.1 RDF interaction

We first describe the content of the RDF database before explaining how to access it. The RDF database contains:

- 750 records from the Rijksmuseum and 1,000 from the Koninklijke Bibliotheek.
- For RDF representation of the hierarchically structured facets, we use SKOS,[25] a model dedicated to the represention of controlled vocabularies.
- Mappings between SKOS Concept Schemes used in the different collections.
- Portal-specific information as *Site Configuration Objects*, identified by URIs with properties defining what collections are part of the set-up, what facets are shown, and also values for the constant text in the Web page presentation and other UI configuration properties. Multiple such as *Site Configuration Objects* may be defined in a repository.

The in-memory RDF store contains, depending on the number of mappings and structured vocabularies that are stored in the database, about 300,000 RDF triples. The Sesame store contains more triples – 520,000 – as its RDFS-entailment implementation implies generation of derived triples (see Section 7).

RDF database access Querying the RDF store for more complex results on the basis of URL query arguments consisted of three steps: (1) building SeRQL queries from URL query arguments, (2) passing them on to the SeRQL engine, gathering the result rows, and (3) finally post-processing the output, for example, counting elements and sorting them. Figure 23 shows an example of a generated SeRQL query. Finding matching records involves finding records annotated by the facet

[25] `http://www.w3.org/2004/02/skos/.`

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

```
objectstr([],_O, _Cols,_Args) --> [].
objectstr([RowObjects|ObjectsList], Offset, Cols, Args) -->
        { Percentage is 100/Cols },
        html(tr(valign(top),\objectstd(RowObjects,Offset,Percentage,Args))),
        { Offset1 is Offset + Cols },
        objectstr(ObjectsList, Offset1, Cols, Args).

objectstd([], _, _, _) --> [].
objectstd([Url|RowObjects], Index, Percentage, Args) -->
       {   ..
           construct_href_index(..., HRef),
           missing_picture_txt(Url, MP)
       },
       html(td(width(Percentage),a([href(HRef)],img([src(Url),alt(MP)])))),
       { Index1 is Index + 1 },
       objectstd(RowObjects, Index1, Percentage, Args).
```

Fig. 24. Part of the HTML-code generation for displaying all the images of a query result in an HTML table.

value or by a value that is a hierarchical descendant of facet value. We implemented this by interpreting records as instances of SKOS concepts and using the transitive and reflexive properties of the rdfs:subClassOf property. This explains for example `{Rec} rdf:type {<http://www.telin.nl/rdf/topia#Paintings>}` in Figure 23.

The SeRQL-query interface contains timing and debugging facilities for single queries; for flexibility it provides access to an external SeRQL server[26] for which we used Sesame,[27] but also to the in-memory store of the SWI-Prolog SeRQL implementation described in Section 7.

9.1.2 HTML-code generation

We used the SWI-Prolog `html_write.pl` library described in Section 3.1 for our HTML-code generation. There are three distinct kinds of Web pages that the multi-faceted browser generates, the portal access page, the refinement page and the single collection-item page. The DCG approach to generating HTML code made it easy to share HTML-code generating procedures such as common headers and HTML code for refinement of choices. The HTML-code generation component contains some 140 DCG rules (1,200 lines of Prolog code of which 800 lines are DCG rules), part of which are simple list-traversing rules such as the example given in Figure 24.

9.1.3 Web server

The Web server is implemented using the HTTP server library described in Section 5.2. The Web server component itself is very small. It follows the skeleton code

[26] We used the `sesame_client.pl` library that provides an interface to external SeRQL servers, packaged with the SWI-Prolog SeRQL library.
[27] `http://www.openrdf.org`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

described in Figure 13. In our case, the **reply/1** predicate extracts the URL root and parameters from the URL. The *Site Configuration Object*, which is introduced in Section 9.1.1, is returned by the RDF-interaction component on the basis of the URL root. It is passed on to the HTML-code generation component that generates Web content as shown in **reply_page/1** in Figure 3.

9.2 Evaluation

This case study shows that SWI-Prolog is effective for building applications in the context of the SW. In a single month, a fully functional prototype portal has been created providing structured access to multiple collections. The independence of any external libraries and the full support of all libraries on different platforms made it easy to develop and install in different operating systems. All case study softwares have been tested to install and run transparently both on Linux and on Microsoft Windows.

At the start of the pilot project, we briefly evaluated existing environments for creating multi-faceted browsing portals: We considered the software available from the Flamenco Project (Hearst *et al.* 2002) and the OntoViews Semantic Portal Creation Tool (Mäkelä *et al.* 2004). The Flamenco software would require developing a translation from RDF to the Flamenco data representation. OntoViews heavily uses SW techniques, but the software was unnecessarily complex for our pilot, requiring a number of external libraries. This together with our need for flexible experiments with various set-ups made us to build our own prototype.

The prototype allowed us to easily experiment with and develop various interesting ways of providing users access to integrated heterogeneous collections (van Gendt *et al.* 2006). The pilot project portal is accessible online for evaluation at `http://stitch.cs.vu.nl/demo.html`.

10 Conclusion

We have presented an overview of the libraries and Prolog language extensions we have implemented and that we provide to the LP community as Open Source resources. As the presented libraries cover very different functionality, we have compared our approach with related work throughout the document. We have demonstrated that Prolog, equipped with an HTTP server library, libraries for reading and writing markup documents, multithreading, Unicode support, unbounded atoms and atom garbage collection, becomes a flexible component in a multi-tier serverarchitecture. Automatic memory management and the absence of pointers and other dangerous language constructs justify the use of Prolog in security-sensitive environments. The middleware, typically dealing with the application logic, is the most obvious tier for exploiting Prolog. In our case studies, we have seen Prolog active as the storage (Section 7), middleware (Section 8) and components presentation tier (Section 9).

Development in the near future is expected to concentrate on SW reasoning, such as the translation of SWRL rules to logic programs. Such translations will

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

benefit from tabling to reach at more predictable response times and allow for more declarative programs. We plan to add more support for OWL reasoning, possibly supporting vital relations for ontology mapping such as `owl:sameAs` in the low-level store. We also plan to add PSP/PWP-like (Section 3) page-generation facilities.

Acknowledgements

Recent development and the preparation of this article was supported by the MultimediaN[28] project funded through the BSIK programme of the Dutch Government. Older projects that provided a major contribution to this research include the European projects IMAT and GRASP and the Dutch project MIA. The XDIG case study is supported by FP6 Network of Excellence European project SEKT (IST-506826). The faceted browser case study is funded by CATCH, a program of the Dutch Organization NWO. We acknowledge the SWI-Prolog community for feedback on the presented facilities, in particular Richard O'Keefe for his extensive feedback on SGML support.

References

Bechhofer, S., Möller, R. and Crowther, P. 2003. The DIG description logic interface. In *International Workshop on Description Logics (DL2003)*, Rome, vol. 81 (electronic).

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*. Lecture notes on computer science, vol. 2342. Springer-Verlag, Berlin, 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. 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.

Gras, D. C. and Hermenegildo, M. V. 2001. Distributed WWW programming using (ciao-) prolog and the piLLoW library. *Theory and Practice of Logic Programming 1,* 3, 251–282.

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 University, CA, 132–141.

Hearst, M., English, J., Sinha, R., Swearingen, K. and Yee, K. 2002. Finding the flow in Web site search. *Communications of the ACM 45,* 9 (September), 42–49.

Horrocks, I. 1999. Fact and ifact. In *Proceedings of the International Workshop on Description Logics (DL'99)*, Sweden, 133–135.

Huang, Z. and Stuckenschmidt, H. 2005. Reasoning with multiversion ontologies: A temporal logic approach. In International Semantic Web Conference, Y. Gil, E. Motta, V. R. Benjamins, and M. A. Musen, Eds. Lecture Notes in Computer Science, vol. 3729. Springer, 398–412.

[28] `www.multimedian.nl`.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press

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*. L. P. Kaelbling and A. Saffiotti, Eds. Professional Book Center, Edinburgh, Scotland, 454–459.

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, Bonn, Germany*.

Mäkelä, E., Hyvönen, E., Saarela, S. and Viljanen, K. 2004. Ontoviews – a tool for creating Semantic Web portals. In *International Semantic Web Conference*, S. A. McIlraith, D. Plexousakis and F. van Harmelen, Eds. Lecture Notes in Computer Science, vol. 3298. Springer, New York, 797–811.

Miller, G. 1995. WordNet: A lexical database for english. *Communication of the ACM 38,* 11 (November), 39–41.

Philips, L. 2000. The double metaphone search algorithm. C/C++ Users Journal *18,* 6 (June), 38–43.

Ramakrishnan, I. V., Rao, P., Sagonas, K., Swift, T. and Warren, D. S. 1995. Efficient tabling mechanisms for logic programs. In *Proceedings of the 12th International Conference on Logic Programming*, L. Sterling, Ed. MIT Press, Cambridge, MA. 697–714.

Schlobach, S. and Huang, Z. 2005. *Inconsistent Ontology Diagnosis: Framework and Prototype*. Deliverable D3.6.1. SEKT. URL:http://www.cs.vu.nl/~huang/sekt/sekt361.pdf

Szeredi, P., Molnár, K. and Scott, R. 1996. Serving multiple HTML clients from a Prolog application. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications, JICSLP'96*, Bonn. URL:http://clement.info.umoncton.ca/~lpnet/lpnet9.html

van Gendt, M., Isaac, A., van der Meij, L. and Schlobach, S. 2006. Semantic web techniques for multiple views on heterogeneous collections: A case study. In *Research and Advanced Technology for Digital Libraries*. Lecture Notes in Computer Science, vol. 4172. Springer, Berlin, 426–437.

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

Wielemaker, J. 2005. An optimised Semantic Web query language implementation in prolog. In *ICLP 2005*, M. Baggrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, Berlin, 128–142.

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. Lecture Notes in Computer Science, vol. 2870. Springer-Verlag, Berlin, 644–658.

https://doi.org/10.1017/S1471068407003237 Published online by Cambridge University Press