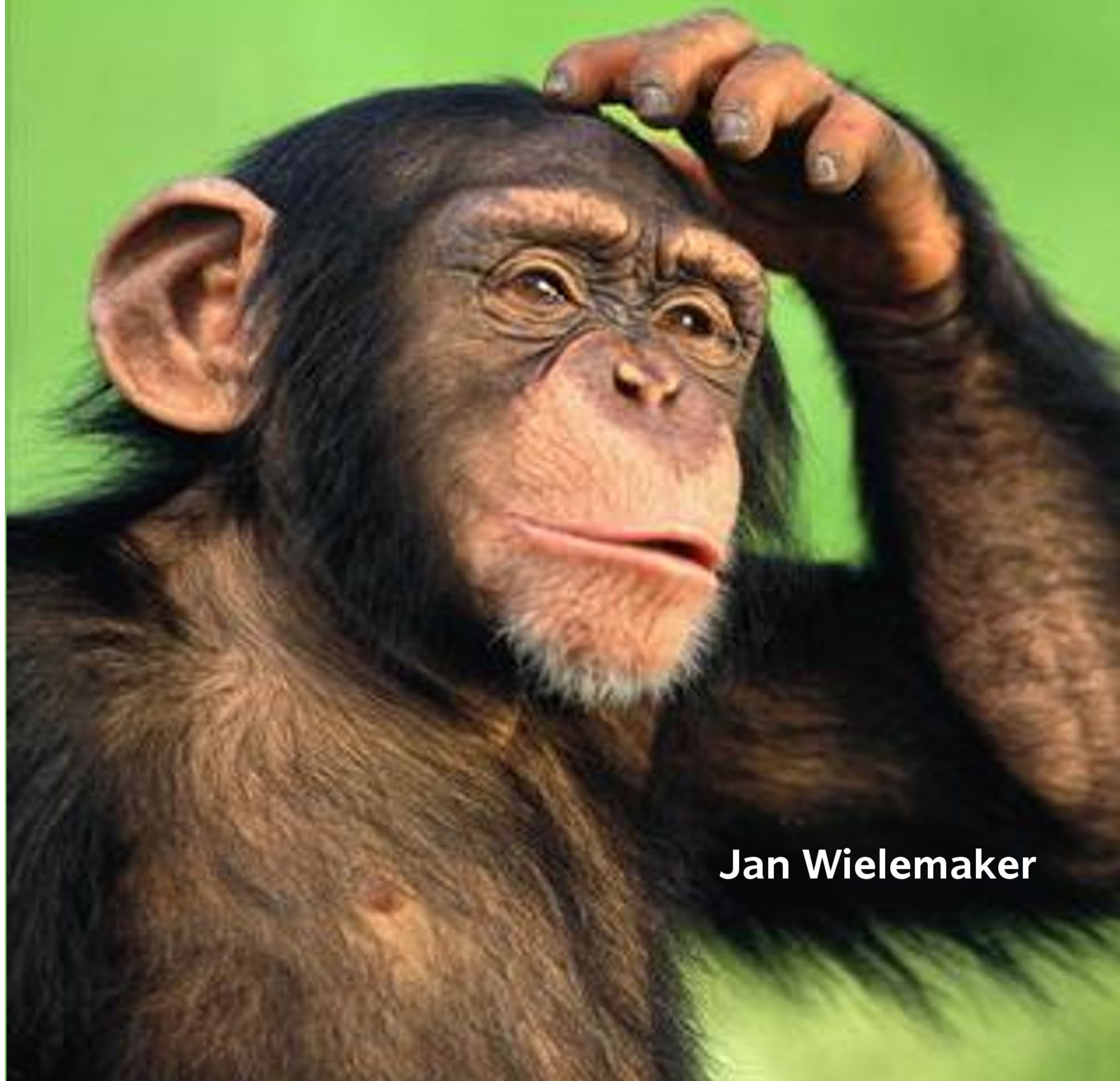
Logic programming for knowledge-intensive interactive applications



Jan Wielemaker

Logic programming for knowledge-intensive interactive applications

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op vrijdag 12 juni 2009, te 14.00 uur
door

Jan Wielemaker

geboren te Koudekerke

Promotoren:	prof. dr. B.J. Wielinga
	prof. dr. A.Th. Schreiber
Overige leden:	prof. dr. B. Demoen
	prof. dr. F.A.H. van Harmelen
	prof. dr. M.L. Kersten
	prof. dr. P.W. Adriaans
	dr. W.N.H. Jansweijer

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

SIKS Dissertation Series No. 2009-10

The research reported in this thesis has been carried out under the auspices of SIKS, the Dutch Research School for Information and Knowledge Systems.

Printed by Wörhman Print Service, Zutphen

Contents

Preface vii

1 Introduction 1
1.1 Application context . . . 1
1.2 Logic Programming . . . 3
1.3 Project context . . . 6
1.4 Research questions . . . 7
1.5 Approach . . . 9
1.6 Outline . . . 9

I Infrastructure for knowledge-intensive applications 13

2 Using triples for implementation: the Triple20 ontology-manipulation tool 15
2.1 Introduction . . . 15
2.2 Core technology: Triples in Prolog . . . 16
2.3 Design Principles . . . 17
2.4 Architecture . . . 18
2.4.1 Rules to define the GUI . . . 20
2.5 An overview of the Triple20 user interface . . . 21
2.6 Implementation . . . 23
2.6.1 The source of triples . . . 23
2.7 Scalability . . . 23
2.8 Related work . . . 25
2.9 Discussion . . . 26

3 Prolog-based Infrastructure for RDF: Scalability and Performance 27
3.1 Introduction 27
3.2 Parsing RDF/XML 28
3.3 Storing RDF triples: requirements and alternatives 30
3.3.1 Requirement from integrating different ontology representations . . 30
3.3.2 Requirements 31
3.3.3 Storage options 32
3.4 Realising an RDF store as C-extension to Prolog 33
3.4.1 Storage format 33
3.4.2 Concurrent access 35
3.4.3 Persistency and caching 35
3.4.4 API rationale 36
3.5 Querying the RDF store 38
3.5.1 RDFS queries 38
3.5.2 Application queries 40
3.5.3 Performance of rdf/3 and rdf_has/4 40
3.6 Performance and scalability comparison 41
3.6.1 Load time and memory usage 43
3.6.2 Query performance on association search 45
3.7 Conclusions 47

4 An optimised Semantic Web query language implementation in Prolog 49
4.1 Introduction 49
4.2 Available components and targets 51
4.3 RDF graphs and SeRQL queries graphs 51
4.4 Compiling SeRQL queries 52
4.5 The ordering problem 54
4.6 Estimating the complexity 57
4.7 Optimising the conjunction 58
4.8 Optional path expressions and control structures 59
4.9 Solving independent path expressions 60
4.10 Evaluation 61
4.11 Related Work 63
4.12 Discussion 64
4.13 Conclusions 64

5 An architecture for making object-oriented systems available from Prolog 67
5.1 Introduction 67
5.2 Approaches 68
5.3 Basic Prolog to OO System Interface 69
5.4 Extending Object-Oriented Systems from Prolog 72
5.5 Transparent exchange of Prolog data 75

5.5.1 Passing Prolog data to a method 76
5.5.2 Storing Prolog data in an XPCE instance variable 77
5.5.3 An example: create a graphical from a Prolog tree 78
5.5.4 Non-deterministic methods 78
5.6 Performance evaluation 80
5.7 Events and Debugging 80
5.8 Related Work 81
5.9 Conclusions and discussion 82

6 Native Preemptive Threads in SWI-Prolog 85
6.1 Introduction 85
6.2 Requirements 86
6.3 What is a Prolog thread? 87
6.3.1 Predicates 87
6.3.2 Synchronisation 88
6.3.3 I/O and debugging 89
6.4 Managing threads from Prolog 89
6.4.1 A short example 89
6.4.2 Prolog primitives 90
6.4.3 Accessing Prolog threads from C 93
6.5 Implementation issues 93
6.5.1 Garbage collection 94
6.5.2 Message queues 95
6.6 Performance evaluation 95
6.6.1 Comparing multi-threaded to single threaded version 96
6.6.2 A case study: Speedup on SMP systems 97
6.7 Related Work 100
6.8 Discussion and conclusions 100

7 SWI-Prolog and the Web 103
7.1 Introduction 104
7.2 XML and HTML documents 105
7.2.1 Parsing and representing XML and HTML documents 105
7.2.2 Generating Web documents 107
7.2.3 Comparison with PiLLoW 110
7.3 RDF documents 110
7.3.1 Input and output of RDF documents 112
7.3.2 Storing and indexing RDF triples 114
7.3.3 Reasoning with RDF documents 116
7.4 Supporting HTTP 117
7.4.1 HTTP client libraries 118
7.4.2 The HTTP server library 120

7.5 Supporting AJAX: JSON and CSS . . . 122
7.5.1 Producing HTML head material . . . 124
7.5.2 Representing and converting between JSON and Prolog . . . 126
7.6 Enabling extensions to the Prolog language . . . 127
7.6.1 Multi-threading . . . 128
7.6.2 Atoms and UNICODE support . . . 128
7.7 Case study — A Semantic Web Query Language . . . 129
7.8 Case study — XDIG . . . 132
7.8.1 Architecture of XDIG . . . 133
7.8.2 Application . . . 134
7.9 Case study — Faceted browser on Semantic Web database integrating multiple collections . . . 135
7.9.1 Multi-Faceted Browser . . . 135
7.9.2 Evaluation . . . 137
7.10 Conclusion . . . 138

II Knowledge-intensive applications **141**

8 PlDoc: Wiki style literate Programming for Prolog **143**
8.1 Introduction . . . 143
8.2 An attractive literate programming environment . . . 145
8.3 An example . . . 147
8.4 Description of PlDoc . . . 149
8.4.1 The PlDoc syntax . . . 149
8.4.2 Publishing the documentation . . . 151
8.4.3 IDE integration and documentation maintenance cycle . . . 151
8.4.4 Presentation options . . . 153
8.5 User experiences . . . 153
8.6 Related work . . . 155
8.7 The PlDoc LaTeX backend . . . 156
8.8 Implementation issues . . . 157
8.8.1 Collecting documentation . . . 158
8.8.2 Parsing and rendering comments . . . 158
8.8.3 Porting PlDoc . . . 158
8.9 Conclusions . . . 159

9 Semantic annotation and search **161**
9.1 Ontology-Based Photo Annotation . . . 161
9.1.1 Introduction . . . 161
9.1.2 Our approach . . . 162
9.1.3 Developing ontologies . . . 162

9.1.4 Annotating photographs using our multimedia information analysis tool . . . 168
9.1.5 Discussion . . . 170
9.2 Supporting Semantic Image Annotation and Search . . . 173
9.2.1 Introduction . . . 173
9.2.2 Approach . . . 173
9.2.3 Background Knowledge . . . 174
9.2.4 An Application Scenario . . . 177
9.2.5 Discussion . . . 180
9.3 Lessons learned . . . 182

10 Thesaurus-based search in large heterogeneous collections 185
10.1 Introduction . . . 186
10.2 Materials and use cases . . . 186
10.2.1 Metadata and vocabularies . . . 186
10.2.2 Use cases . . . 187
10.3 Required methods and components . . . 190
10.3.1 Using a set of fixed queries . . . 190
10.3.2 Using graph exploration . . . 190
10.3.3 Term search . . . 191
10.3.4 Literal matching . . . 192
10.3.5 Using SPARQL . . . 192
10.3.6 Summary of requirements for search . . . 193
10.4 The ClioPatria search and annotation toolkit . . . 194
10.4.1 Client-server architecture . . . 195
10.4.2 Output formats . . . 195
10.4.3 Web services provided by ClioPatria (API) . . . 196
10.5 Discussion and conclusion . . . 198

11 Conclusions 203
11.1 The research questions revisited . . . 204
11.2 Architectures . . . 206
11.3 Discussion: evaluation of our infrastructure . . . 206
11.3.1 Key decisions about the infrastructure . . . 208
11.4 Challenges . . . 212

Bibliography 215

Summary 227

Samenvatting 233

SIKS Dissertatiereeks 239

Preface

It was 1985, when I graduated from Eindhoven University. I had some options about what to do. Instead of picking one, I asked Bob Wielinga whether he had some short-term employment in the European 'Esprit' project that just had started. My plan was to wait for a nice opportunity to work on what I, then, considered a more fundamental artificial intelligence subject, such as machine learning. I started work on the KADS project, where Anjo Anjewierden tried to teach me the difference between hacking and programming. By studying and extending code from Anjo and Richard O'Keefe, whom I had met during a training period at Edinburgh University, I gradually discovered the beauty of proper programming. Anjo's motto was "the ultimate documentation is the source". For good source, handed to someone with basic knowledge of the programming language, this indeed comes close to the truth. Nevertheless, I am proud to have him as a co-author of a paper on a Prolog documentation system (chapter 8 of this thesis). The 'art of programming' had caught me and I abandoned my original plans doing AI research.

At least as important as the joy of striving for clear coding of large and complex systems, was the way Bob Wielinga dealt with 'scientific programmers'. Later, I realised that in most research groups this species was treated as a semi-automated code-generation facility. Instead, Bob involved Anjo and me in the research, gave us almost unlimited freedom and was always the first to try our prototypes. His never lasting energy to modify and comment on our work was a source of inspiration. I still wonder how Bob managed to demonstrate our prototypes at project reviews without making them crash.

During this period, I had plenty of time to play the Chinese game of Go and hack around on the University computer systems on anything I liked. This resulted in SWI-Prolog. Somehow, the developers in the KADS project quickly ignored the consortium decision to use the commercial Quintus Prolog system and started using my little toy. Well, possibly it helped that I promised a bottle of cognac for every 10 reported bugs. Although this resulted in endless arguments on whether a spelling error in a warning message was a bug or not, Huub Knops managed to earn his bottle. Anja van der Hulst joined SWI, and I soon had to deal with Anja, Go and SWI-Prolog. My support in realising a prototype for her research is reminded later in this preface.

At some point, Frank van Harmelen joined SWI and pointed us to great 'free' software, notably Emacs and LaTeX. Not really knowing what to do with SWI-Prolog, we decided to join the free software community and make it available from our FTP-server. We sold academic licenses for the accompanying graphical toolkit PCE, designed by Anjo and extended a bit by me. Selling software and dealing with legal issues around licenses is not the strength of a University, as Lenie Zandvliet and Saskia van Loo will agree. Sorry for all the extra work.

SWI-Prolog/PCE became a vehicle in the department to accumulate experience on building 'knowledge intensive interactive applications'. At some point in the early 90s, a wider academic community started to form around SWI-Prolog/PCE. An invitation to work as guest researcher for SERC by Peter Weijland allowed to initiate the ideas that lead to chapter 5 of this thesis and the research on version management systems filled a gap in my experience Anjo never conveyed to me: the use of a version management system. Robert de Hoog never thought very highly of Prolog, but he needed Anjo to work on the PIMS project and convinced me to port the system to Microsoft's operating systems. I first tried Windows 3.1, then Windows 95 and finally succeeded on Windows NT 3.5. It was no fun, but I must admit that it was probably the step that made SWI-Prolog a success: suddenly SWI-Prolog could run for free and without any strings attached on the University Unix networks as well as on the student's home PCs. When search engines became important on the internet, pointers from many University course-pages to SWI-Prolog greatly helped making the system popular. So, thank you, Robert!

Although SWI-Prolog was supported by a large and growing community, I started worrying. Projects tended to move to other programming languages. Although Bob managed to organise projects such that we could deploy Prolog to create 'executable specifications', this was a worrying development. Fortunately, a number of opportunities came along. Projects moved towards networked component-based designs, which allowed partners to use their language of choice instead of having to agree on a single language. At the ICLP in Mumbai (2003), Tom Schrijvers and Bart Demoen came with the deal where, with their help, I would provide the low-level support for constraint programming and their Leuven-based team would make their constraint systems available on SWI-Prolog. Without this cooperation SWI-Prolog might have lost contact with the logic programming community.

In the meanwhile SWI-Prolog started to attract serious commercial users. One came as a surprise. Considering the shift towards networking, I had added concurrency and Unicode to SWI-Prolog. Suddenly I was bombarded by mails from Sergey Tikhonov from Solvo in St. Petersburg. Without his bug-reports and patches, multi-threaded SWI-Prolog would never have become stable and this thesis would not have existed. Steve Moyle helped to turn this into a paper (chapter 6) and his company funded the development of SWI-Prolog testing framework and documentation system. Of course I should not forget to mention his elaborations on Oxford pubs and colleges. Mike Elston from SecuritEase in New Zeeland funded several projects and his colleagues Keri Harris and Matt Lilley have sent me many bug reports and fortunately more and more patches. I was pleased that he and his wife unexpectedly showed up at our doorstep a year ago. They showed me and Anja parts of

Amsterdam that were unknown to us.

The young and brilliant student Markus Triska showed up to work in Amsterdam during the summer. Unfortunately the apartment I managed to arrange for him turned out to house some companions in the form of flees, so at some point he preferred the University couch. Nevertheless, this didn't break his enthusiasm for SWI-Prolog, which resulted in various constraint programming libraries. He also managed to interest Ulrich Neumerkel. I will always admire his dedication to find bugs. His work surely contributed in realising 24×7 Prolog-based services.

Unfortunately, I cannot acknowledge everybody from the Prolog community. I will name a few and apologise to the others. Richard O'Keefe shows up again. He has followed the mailinglist for many years. Without his comments, the list would be much less informative and less fun. Then we have Paulo Moura, always trying to keep the herd of Prolog developers together. Considering that Prolog developers are more like cats than sheep, this effort cannot be left unacknowledged. Vitor Santos Costa's open mind, combined with some pressure by Tom Schrijvers has resulted in some level of cooperation between YAP and SWI-Prolog that we want to deepen in the near future. Paul Singleton connects Prolog to Java and hence allows me to survive in this world.

The most important opportunity for me was created by Guus Schreiber, who managed to participate in the MIA and MultimediaN projects on search and annotation of physical objects. These projects were run by an inspiring team from SWI, CWI and the VU. Guus' choice to use the emerging Semantic Web created a great opportunity for deploying Prolog. While I tried to satisfy their demands on the infrastructure, Michiel Hildebrand and Jacco van Ossenbruggen did most of the Prolog programming that created a price-winning demonstrator at ISWC-2006. More important for this thesis, these projects provided the opportunity and inspiration to write scientific papers. These papers, and especially the one published as chapter 10 made this thesis possible.

Writing software isn't always easy to combine with writing papers or a thesis. Somehow these two tasks require two incompatible states of mind. As there was no obvious need to write a thesis, many evaluations of my functioning at the University contained some sufficiently vague statement about a PhD, after which I and Anja had a good laugh and another year had passed. Until somewhere fall 2007, I was in the local pub with Bob Wielinga and Guus Schreiber. After some beers I ventilated some bold statements about the Semantic Web and Guus commented "you should write this down in a paper and then we can combine it with some of your recent papers and turn it into a PhD". Not completely sober, I turned home and told Anja: "They want me to write a PhD." Anja started laughing hilariously, expecting me to do the same. I didn't. Anja stared at me, confused and then slightly worried. This was not the standard act. When I asked for her support she replied: "Well, if you really want it, you should.", to continue with "As long as I don't have to read it." It was her well deserved revenge for my attitude towards reading specifications for software prototypes.

Next to work-wise, SWI has always been a great group to work in. For me, it started on the top-floor of what is now known as 'het Blauwe Huis' on the Herengracht in Amsterdam. SWI was a gathering of great people with whom I have spent many hours in the pubs nearby.

From there we moved to the top-floor of the psychology building at the Roetersstraat. I will not forget the discussions in café Solo with Frank van Harmelen, Gertjan van Heijst, Manfred Aben, Dieter Fensel and many more. I have many good memories of the colleagues with whom I shared an office. John van den Elst was very noisy, so after a month I was happy he would be in Antibes for the next month and after a month in an empty office I was happy he returned, etc. Luckily Chris van Aart, though we never shared offices, made the months without John bearable with his admiration for Java. Sharing the office with Dennis Beckers and Hedderik van Rijn was great. In that period I lived in Leiden and many Wednesdays I enjoyed a *pizza quatro formaggi* with Carolien Metselaar in Palermo before visiting the Amsterdam Go-club. Noor Christoph learned me that finishing a PhD implied there was never a dull moment.

Chapter 1

Introduction

Traditionally, Logic Programming is used first of all for *problem solving*. Although an important task, implementing the pure problem solving task of a program is typically a minor effort in the overall development of an application. Often, *interaction* is responsible for a much larger part of the code. Where the logic programming community typically stresses the *declarative* aspects of the language, imperative languages provide *control*, an important aspect of interaction. In practice, the distinction is not that rigid. In problem solving we want some control over the algorithms actually used while rules play a role in defining interaction policies.

Mixing languages is a commonly used way to resolve this problem. However, mixing languages harms rapid prototyping because it requires the developers to master multiple languages and to define an interface between the interaction and problem solving components of the application. In this thesis we investigate using the Logic Programming paradigm for both the application logic and the interaction under the assumption that a united environment provides a productive programming environment for knowledge intensive interactive applications.

This introduction starts with a description of the context in which the research was carried out and a characterisation of the software built, followed by a brief overview of and motivation for the use of Logic Programming. After that, we give a historical overview that ends with a timeline of projects described in this thesis, followed by the research questions and an outline of the thesis.

1.1 Application context

This thesis studies the construction of a software infrastructure for building tools that help humans in understanding and modifying knowledge. With software infrastructure, we refer to language extensions and libraries developed to support this class of applications. Our infrastructure is the result of experience accumulated in building tools, a process that is discussed in more detail in section 1.3. Examples of knowledge we studied are conceptual models created by knowledge engineers, ontologies and collections of meta-data that are

based on ontologies. If humans are to create and maintain such knowledge, it is crucial to provide suitable *visualisations* of the knowledge. Often, it is necessary to provide multiple visualisations, each highlighting a specific aspect of the knowledge. For example, if we consider an ontology, we typically want a detailed property sheet for each concept or individual. Although this presentation provides access to all aspects of the underlying data, it is poorly suitable to assess aspects of the overall structure of the ontology such as the concept hierarchy, part-of structures or causal dependencies. These are much better expressed as trees (hierarchies) or graphs that present one aspect of the ontology, spanning a collection of concepts or individuals. The user must be able to interact with each of these representations to *edit* the model. For example, the user can change the location of a concept in the concept hierarchy both by moving the concept in the tree presentation and changing the parent in the property sheet.

Next to visualisations, a core aspect of tools for knowledge management is the actual representation of the knowledge, both internally in the tool and externally for storage and exchange with other tools. In projects that pre-date the material described in this thesis we have used various frame-based representations. Modelling knowledge as entities with properties is intuitive and maps much easier to visualisations than (logic) language based representations (Brachman and Schmolze 1985). In the MIA project (chapter 9, Wielemaker et al. 2003a) we adopted the Semantic Web language RDF (Resource Description Format, Lassila and Swick 1999) as our central knowledge representation format. RDF is backed by a large and active community that facilitates exchange of knowledge with tools such as Protégé (Grosso et al. 1999) and Sesame (Broekstra et al. 2002). RDF provides an extremely simple core data model that exists of triples:

$\langle$*Subject, Predicate, Object*$\rangle$

Informally, $\langle$*Predicate, Object*$\rangle$ tuples provide a set of name-value pairs of a given *Subject*. The power of RDF becomes apparent where it allows layering more powerful languages on top of it. The two commonly used layers are RDFS that provides classes and types (Brickley and Guha 2000) and OWL that provides *Description Logic* (DL, Horrocks et al. 2003; Baader et al. 2003). If the knowledge modelling is done with some care the same model can be interpreted in different languages, providing different levels of semantic commitment. Moreover, we can define our own extensions to overcome current limitations of the framework. RDF with its extensions turned out to be an extremely powerful representation vehicle. In this thesis we present a number of technologies that exploit RDF as an underpinning for our infrastructures and tools.

Most of the work that is presented in this thesis was executed in projects that aim at supporting *meta data* (annotations) for items in collections. Initially we used collections of photos, later we used museum collections of artworks. Knowledge management in this context consists of creating and maintaining the ontologies used to annotate the collection, creating and maintaining the meta data and exploring the knowledge base. Our knowledge bases consist of large RDF graphs in which many edges (relations) and vertices (concepts and instances) are only informally defined and most ‘concepts’ have too few attributes for

computers to grasp their meaning. In other words, the meaning of the graph only becomes clear if the graph is combined with a commonsense interpretation of the *labels*. This implies we have little use for formal techniques such as description logic reasoning (see section 10.5.0.5). Instead, we explore the possibilities to identify relevant sub-graphs based on a mixture of graph properties (link counts) and the semantics of a few well understood relations (e.g., identity, is-a) and present these to the user.

In older projects as well as in the photo annotation project (chapter 9) we aimed at applications with a traditional graphical user interface (GUI). Recently, our attention has switched to web-based applications. At the same time attention shifted from direct manipulation of multiple views on relatively small models to exploring vast knowledge bases with relatively simple editing facilities.

From our experience with projects in this domain we assembled a number of core requirements for the tools we must be able to create with our infrastructure:

1. *Knowledge*
   For exchange purposes, we must be able to read and write the standard RDF syntaxes. For editing purposes, we need reliable and efficient storage of (small) changes, together with a history on how the model evolved and support for *undo*. As different applications need different reasoning facilities, we need a flexible framework for experimenting with specialised reasoning facilities. Considering the size of currently available background knowledge bases as well as meta-data collections we estimate that support for about 100 million triples suffices for our experimental needs.

2. *Interactive web applications*
   Current emphasis on web applications requires an HTTP server, including concurrency, authorisation and session management. Creating web applications also requires support for the document serialisation formats that are in common use on the web.

3. *Interactive local* GUI *applications*
   Although recent projects needs have caused a shift towards web applications, traditional local GUI applications are still much easier to build, especially when aiming for highly interactive and complex graphical tools.

1.2 Logic Programming

Although selecting programming languages is in our opinion more a matter of 'faith'[1] than science, we motivate here why we think Logic Programming is especially suited for the development of knowledge-intensive interactive (web) applications.

> *Logic programming is, in its broadest sense, the use of mathematical logic for computer programming. In this view of logic programming, which can be*

[1] *The Oxford Concise English Dictionary*: "strong belief [in a religion] based on spiritual conviction rather than proof."

> *traced at least as far back as John McCarthy's (McCarthy 1969) advice-taker proposal, logic is used as a purely declarative representation language, and a theorem-prover or model-generator is used as the problem-solver. The problem-solving task is split between the programmer, who is responsible only for ensuring the truth of programs expressed in logical form, and the theorem-prover or model-generator, which is responsible for solving problems efficiently.*
>
> *Wikipedia, oct. 2008*

Defined as above, logic programming is obviously the perfect paradigm for expressing knowledge. However, when applying this paradigm we are faced with two problems: (1) there is no sufficiently expressive logic language with an accompanying efficient theorem prover and (2) although control may not be of interest for theorem proving, it *is* important when communicating with the outside world where ordering communication actions often matters.

An important break through in the field of logic programming was the invention of the *Prolog* programming language (Colmerauer and Roussel 1996; Deransart et al. 1996). The core of Prolog provides a simple resolution strategy (SLD resolution) for *Horn clauses*. The resulting language has a declarative reading, while its simple resolution strategy provides an imperative reading for a Prolog program at the same time. For example,[2] the program below can be read as "a(*X*,*Z*) is true if b(*X*,*Y*) and c(*Y*,*Z*) are true". Particularly if the literals b and c have exactly one solution, the first argument is *input* and the second *output*, it can also be read as "To create *Z* from *X*, first call b to create *Y* from *X* and then call c to create *Z* from *Y*.

```
a(X, Z) :-
        b(X, Y),
        c(Y, Z).
```

In our view, this dual interpretation of the same code greatly contributes to the value of Prolog as a programming language for interactive knowledge-intensive applications. Notably in chapter 4 we see the value of the declarative reading of RDF expressions to achieve optimisation. Declarative reading also helps in many simple rules needed both as glue for the knowledge and as rules in the interactive interface. On the other hand I/O, associated to interactivity, often asks for an imperative reading: we do not want to *prove* `write('hello world')` is true but we want to *execute* this statement.

Since its birth, the Prolog language has evolved in several directions. Notably with the introduction of the WAM (Hassan Aït-Kaci 1991) compiler technology has improved, providing acceptable performance. The language has been extended with extra-logical primitives, declarations, modules and interfaces to other programming languages to satisfy software engineering requirements. At the same time it has been extended to enhance its power as a declarative language by introducing extended unification (Holzbaur 1990) which initiated

[2] Throughout this thesis we use ISO Prolog syntax for code.

constraint logic programming (CLP) and new resolution techniques such as SLG resolution (Ramakrishnan et al. 1995) which ensures termination of a wider class of Horn clause programs.

Practical considerations In addition to the above motivation for a language that combines declarative and imperative notions there are other aspects of the language that make it suitable for prototyping and research. Many AI languages, both functional- and logic-based, share *reflexiveness*, *incremental compilation* and *safe execution* and lack of *destructive* operations.

Reflexiveness is the ability to process programs and goals as normal data. This feature facilitates program transformation, generation and inspection. Based on this we can easily define more application oriented languages. Examples can be found in section 3.2, defining a language to match XML trees and section 7.2.2.1, defining a language to generate compliant HTML documents and chapter 5 where Prolog syntax is translated partly into code for an external object oriented system. In chapter 4 we exploit the ability to inspect and transform goals for optimising RDF graph matching queries.

Incremental compilation is the ability to compile and (re-)load files into a running application. This is a vital feature for the development of the interactive applications we are interested in because it allows modifying the code while the application is at a certain state that has been reached after a sequence of user interactions. Without incremental compilation one has to restart the application and redo the interaction to reach the critical state again. A similar argument holds for knowledge stored in the system, where reloading large amounts of data may take long. Safe execution prevents the application from crashing in the event of program errors and therefore adds to the ability to modify the program under development while it is running. Prolog comes with an additional advantage that more permanent data is generally stored in clauses or a foreign extension, while volatile data used to pass information between steps in a computation is often allocated on the stacks, which is commonly discarded after incremental computation. This makes the incremental development cycle less vulnerable to changes of data formats that tend to be more frequent in intermediate results than in global data that is shared over a larger part of the application.

The lack of destructive operations on data is shared between functional and logical languages. Modelling a computation as a sequence of independent states instead of a single state that is updated simplifies reasoning about the code, both for programmers and program analysis tools.

Challenges Although the above given properties are useful for the applications we want to build, there are also vital features that are not or poorly supported in many Prolog implementations. Interactive applications need a user interface; dealing with RDF requires a scalable RDF triple store; web applications need support for networking, the HTTP protocol and document formats; internationalisation of applications needs UNICODE; interactivity and scalability require concurrency. Each of these features are provided to some extent in some implementations, but we need integrated support for all these features in one environment.

1.3 Project context

Our first project was Shelley (Anjewierden et al. 1990). This tool provided a comprehensive workbench managing KADS (Wielinga et al. 1992) models. The use of (Quintus-)Prolog was dictated by the project. We had to overcome two major problems.

- Adding graphics to Prolog.
- Establishing a framework for representing the complex KADS models and connecting these representations to the graphics layer.

Anjo Anjewierden developed PCE for Quintus Prolog based on ideas from an older graphics interface for C-Prolog developed by him. PCE is an object oriented foreign (C-)library to access external resources. We developed a simple frame-based representation model in Prolog and used the MVC (Model-View-Controller) architecture to manage models using direct manipulation, simultaneously rendering multiple views of the same data (Wielemaker and Anjewierden 1989).

At the same time we started SWI-Prolog, initially out of curiosity. Quickly, we identified two opportunities provided by—at that time—SWI-Prolog's unique feature to allow for recursive calls between Prolog and C. We could replace the slow pipe-based interface to PCE with direct calling and we could use Prolog to define new PCE classes and methods as described in chapter 5 (Wielemaker and Anjewierden 2002).

XPCE as it was called after porting the graphics to the X11 windowing system was the basis of the CommonKADS workbench, the followup of Shelley. The CommonKADS workbench used XPCE objects both for modelling the GUI and the knowledge. Extending XPCE core graphical classes in Prolog improved the design significantly. Using objects for the data, replacing the Prolog-based relational model of Shelley, was probably a mistake. XPCE only allows for non-determinism locally inside a method and at that time did not support logical variables. This is not a big problem for GUI programming, but loses too much of the power of Prolog for accessing a knowledge base.

In the MIA project (chapter 9, Wielemaker et al. 2003a; Schreiber et al. 2001) we built an ontology-based annotation tool for photos, concentrating on what is depicted on the photo (the *subject matter*). We decided to commit to the Semantic Web, which then only defined RDF and RDFS. The MIA tool is a stand-alone graphics application built in XPCE. All knowledge was represented using `rdf`(*Subject, Predicate, Object*), a natural mapping of the RDF data model. Rdf/3 is a pure Prolog predicate that was implemented using several dynamic predicates to enhance indexing. With this design we corrected the mistake of the CommonKADS Workbench. In the same project, we started version 2 of the RDF-based annotation infrastructure. The pure semantics of the `rdf/3` predicate was retained, but it was reimplemented in C to enhance performance and scalability as described in chapter 3 (Wielemaker et al. 2003b). We developed a minimal pure Prolog Object layer and associated this with XPCE to arrive at a more declarative approach for the GUI, resulting in the Triple20 ontology tool (chapter 2, Wielemaker et al. 2005).

In the HOPS project[3] we investigated the opportunities for RDF query optimisation (chapter 4, Wielemaker 2005) and identified the web, and particularly the Semantic Web, as a new opportunity for the Prolog language. We extended Prolog with concurrency (chapter 6, Wielemaker 2003a) and international character set support based on UNICODE and UTF-8 and added libraries to support this application area better. Motivated in chapter 7 (Wielemaker et al. 2008), we decided to provide native Prolog implementations of the HTTP protocol, both for the server and client. This infrastructure was adopted by and extended in the MultimediaN E-culture project that produced ClioPatria (chapter 10, Wielemaker et al. 2008). Extensions were demand driven and concentrated on scalability and support for a larger and decentralised development team. Scalability issues included concurrency in the RDF store and indexed search for tokens inside RDF literals. Development was supported by a more structured approach to bind HTTP paths to executable code, distributed management of configuration parameters and PlDoc, an integrated literal programming environment (chapter 8, Wielemaker and Anjewierden 2007).

Figure 1.1 places applications and infrastructure on a timeline. The actual development did not follow such a simple waterfall model. Notably ClioPatria was enabled by the previously developed infrastructure, but at the same time initiated extensions and refinements of this infrastructure.

1.4 Research questions

Except for section 1.2 and section 1.3, we consider the choice for Prolog and RDF a given. In this thesis, we want to investigate where the Prolog language needs to be extended and what design patterns must be followed to deploy Prolog in the development of *knowledge-intensive interactive* (web) *applications*. This question is refined into the three questions below.

1. *How to represent knowledge for interactive applications in Prolog?*
   The RDF triple representation fits naturally in the Prolog relational model. The size of RDF graphs and the fact that they are more limited than arbitrary 3-argument relations pose challenges and opportunities.

   (a) *How to store* RDF *in a scalable way?*
   Our RDF store must appear as a pure Prolog predicate to facilitate reasoning and at the same time be scalable, allow for reliable persistent storage and to allow for concurrent access to act as a building block in a web service.

   (b) *How to optimise complex queries on* RDF *graphs?*
   Naively executed, queries to match RDF graph expressions can be extremely slow. How can we optimise them and which knowledge can the low-level RDF store provide to help this process?

[3] http://www.bcn.es/hops/

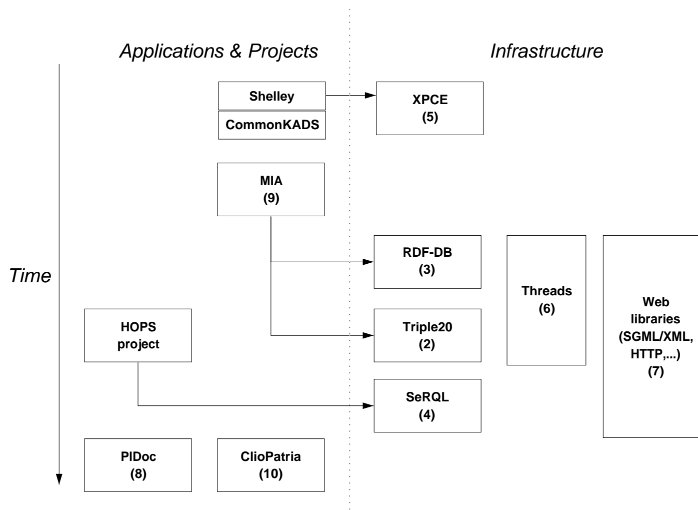


Figure 1.1: This diagram shows all projects and components in chronological order. Numbers between brackets are the chapter numbers describing the component. The arrows indicate the most important influence relations.

2. *How to support web applications in Prolog?*
   Since approximately 2004 our attention in projects shifted from stand-alone GUI applications to web-based applications. As we show in the related work sections of chapter 7, web support is recognised as an issue in the Prolog community, but the solutions are only partial.

   (a) *How to represent web documents?*
   The web defines data formats such as HTML, XML and RDF. What is the appropriate way to read, process and write these using Prolog?

   (b) *How to support web services?*
   A web service must bind HTTP requests to executable code that formulates a reply represented as a web document. What architecture is needed if this executable code is in Prolog? How do we realise an environment that allows for debugging, is scalable and simple to deploy?

3. *How to support graphical applications in Prolog?*
   As we have seen above, visualisation plays a key role in applications that support the user in managing knowledge. Graphics however does not get much attention in the

Prolog community and intuitively, like I/O, does not fit well with the declarative nature of Prolog.

(a) *How can we interface Prolog to external object oriented* GUI *systems?*
Virtually all graphics systems are object oriented and therefore a proper interface to an external object oriented system is a requirement for graphics in Prolog.

(b) *How to create new graphical primitives?*
Merely encapsulating an object oriented system is step one. New primitives can often only be created by deriving classes from the GUI base classes and thus we need a way to access this functionality transparently from Prolog.

(c) *How to connect an interactive* GUI *to the knowledge*
The above two questions are concerned with the low-level connection between Prolog and a GUI toolkit. This question addresses the interaction between knowledge stored in an RDF model and the GUI.

1.5 Approach

Development of software prototypes for research purposes is, in our setting, an endeavour with two goals: (1) satisfy the original research goal, such as evaluating the use of ontologies for annotation and search of multi-media objects and (2) establish a suitable architecture for this type of software and realise a reusable framework to build similar applications. As described in section 1.3, many projects over a long period of time contributed to the current state of the infrastructure. Development of infrastructure and prototypes is a cyclic activity, where refinements of existing infrastructure or the decision to build new infrastructure is based on new requirements imposed by the prototypes, experience in older prototypes and expectations about requirements in the future. Part II of this thesis describes three prototype applications that both provide an evaluation and *lessons learned* that guide improvement of the technical infrastructure described in Part I. In addition to this *formative evaluation*, most of the infrastructure is compared with related work. Where applicable this evaluation is quantitative (time, space). In other cases we compare our design with related designs, motivate our choices and sometimes use case studies to evaluate the applicability of our ideas. Externally contributed case studies are more neutral, but limited due do lack of understanding of the design patterns with which the infrastructure was developed or failure to accomplish a task (efficiently) due to misunderstandings or small omissions rather than fundamental flaws in the design. The choice between self-evaluation and external evaluation depends on the maturity of the software, where mature software with accompanying documentation is best evaluated externally. Chapter 7 and 8 include external experience.

1.6 Outline

Where figure 1.1 shows projects and infrastructure in a timeline to illustrate how projects influenced the design and implementation of the infrastructure, figure 1.2 shows how libraries,

tools and applications described in this thesis depend on each other.

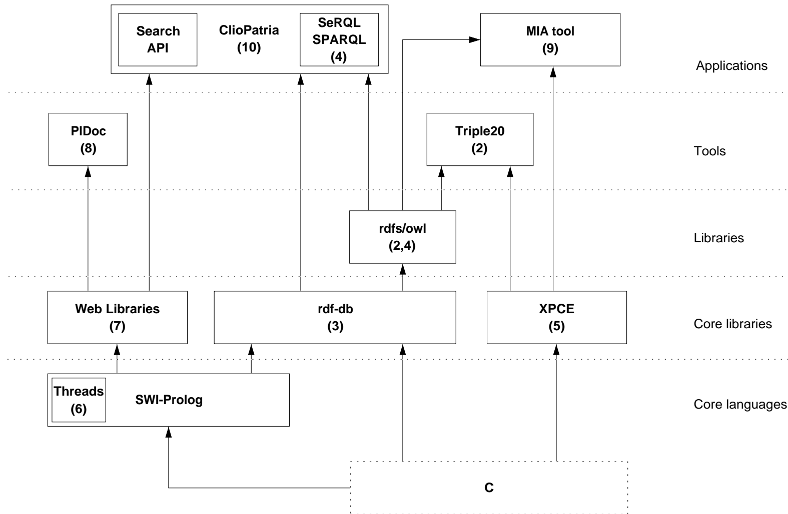


Figure 1.2: Overview of software described in this thesis. The arrows indicate 'implemented on top of', where we omitted references from the higher level libraries and applications written in Prolog. Numbers between brackets refer to the chapter that describe the component.

We start our survey near the top with *Triple20* (chapter 2, Wielemaker et al. 2005), a scalable RDF editor and browser. The paper explains how the simple and uniform RDF triple-based data model can be used as an *implementation* vehicle for highly interactive graphical applications. Triple20 can both be viewed as an application and as a library and is discussed first because it illustrates our approach towards knowledge representation and interactive application development. The paper is followed by four papers about enabling technology: chapter 3 (Wielemaker et al. 2003b) on RDF storage, chapter 4 (Wielemaker 2005) on query optimisation and RDF query language implementation, chapter 5 (Wielemaker and Anjewierden 2002) on handling graphics in Prolog and finally chapter 6 (Wielemaker 2003a) on a pragmatic approach to introduce concurrency into Prolog. The first part is completed with chapter 7 (Wielemaker et al. 2008), providing a comprehensive overview of the (semantic) web support in SWI-Prolog. Being an overview paper it has some overlap with earlier chapters, notably with chapters 3, 4 and 6.

In part II we describe three applications. The first application is *PlDoc* (chapter 8, Wielemaker and Anjewierden 2007), a web-based tool providing literate programming for Prolog. Although PlDoc itself is part of the infrastructure it is also an application of our Prolog-based

libraries for building web applications. Chapter 9, (Schreiber et al. 2001; Wielemaker et al. 2003a) describes the role of ontologies in annotation and search, an application that involves knowledge and a traditional GUI. This chapter ends with a lessons learned section that motivates a significant part of the infrastructure described in part I. We conclude with chapter 10 (Wielemaker et al. 2008), which discusses architectural considerations for Semantic Web applications aiming at handling heterogenous knowledge that is only in part covered by formal semantics. The described application (ClioPatria) involves large amounts of knowledge and a rich web-based interactive interface.

Part I

Infrastructure for knowledge-intensive applications

Chapter 2

Using triples for implementation: the Triple20 ontology-manipulation tool

About this chapter This chapter has been published at the ISWC-05, Galway (Wielemaker et al. 2005) and introduces Triple20, an ontology editor and browser. The Triple20 application is described in the beginning of this thesis to clarify our position in knowledge representation for interactive applications, addressing research question 3c. The infrastructure needed to build Triple20 is described in the subsequent chapters, notably chapter 3 (the RDF database), chapter 6 (multi-threading) and chapter 5 (XPCE, connecting object oriented graphics libraries to Prolog).

Depending on the context, we refer to Triple20 as a tool, library, browser or editor. It can be used as a stand-alone editor. It can be loaded in—for example—ClioPatria (chapter 10) to explore (browse) the RDF for debugging purposes, while tOKo (Anjewierden and Efimova 2006; Anjewierden et al. 2004) uses Triple20 as a library.

Abstract Triple20 is an ontology manipulation and visualisation tool for languages built on top of the Semantic-Web RDF triple model. In this article we introduce a triple-centred application design and compare this design to the use of a separate proprietary internal data model. We show how to deal with the problems of such a low-level data model and show that it offers advantages when dealing with inconsistent or incomplete data as well as for integrating tools.

2.1 Introduction

Triples are at the very heart of the Semantic Web (Brickley and Guha 2000). RDF, and languages built on top of it such as OWL (Dean et al. 2004) are considered *exchange* languages: they allow exchanging knowledge between agents (and humans) on the Semantic

Web through their *atomic* data model and well-defined semantics. The agents themselves often employ a data model that follows the design, task and history of the software. The advantages of a proprietary internal data model are explained in detail by Noy et al. 2001 in the context of the Protégé design.

The main advantage of a proprietary internal data model is that it is neutral to external developments. Noy et al. 2001 state that this enabled their team to quickly adapt Protégé to the Semantic Web as RDF became a standard. However, this assumes that all tool components commit to the internal data model and that this model is sufficiently flexible to accommodate new external developments. The RDF triple model and the higher level Semantic Web languages have two attractive properties. Firstly, the triple model is generic enough to represent *anything*. Secondly, the languages on top of it gradually increase the semantic commitment and are extensible to accommodate almost any domain. Our hypothesis is that a tool infrastructure using the triple data model at its core can profit from the shared understanding of the triple model. We also claim that, where the layering of Semantic Web languages provides different levels of understanding of the same document, the same will apply for tools operating on the triple model.

In this article we describe the design of Triple20, an ontology editor and browser that runs directly on a triple representation. First we introduce our triple store, followed by a description of how the model-view-controller design pattern (Krasner and Pope 1988, figure 2.1) can be extended to deal with the low level data model. In section 2.4.1 to section 2.6 we illustrate some of the Triple20 design decisions and functions, followed by some metrics, related work and discussion.

2.2 Core technology: Triples in Prolog

The core of our technology is Prolog-based. The triple-store is a memory-based extension to Prolog realising a compact and highly efficient implementation of `rdf/3` (chapter 3, Wielemaker et al. 2003b). Higher level primitives are defined on top of this using Prolog *backward chaining* rather than *transformation* of data structures. Here is a simple example that relates the title of an artwork with the name of the artist that created it:

```
artwork_created_by(Title, ArtistName) :-
    rdf(Work, vra:creator, Artist),
    rdf(Work, vra:title, literal(Title)),
    rdf(Artist, rdfs:label, literal(ArtistName)).
```

The RDF infrastructure is part of the Open Source SWI-Prolog system and used by many internal and external projects. Higher-order properties can be expressed easily and efficiently in terms of triples. Object manipulations, such as defining a class are also easily expressed in terms of adding and/or deleting triples. Operating on the same triple store, triples not only form a mechanism for *exchange* of data, but also for cooperation between *tools*. Semantic Web standards ensure consistent interpretation of the triples by independent tools.

2.3 Design Principles

Most tool infrastructures define a data model that is inspired by the tasks that have to be performed by the tool. For example, Protégé, defines a flexible metadata format for expressing the basic entities managed by Protégé: classes, slots, etc. The GUI often follows the model-view-controller (MVC) architecture (Krasner and Pope 1988, figure 2.1). We would like to highlight two aspects of this design:

- All components in the tool set must conform to the same proprietary data model. This requirement complicates integrating tools designed in another environment. Also, changes to the requirements of the data model may pose serious maintainability problems.

- Data is translated from/to external (file-)formats while loading/saving project data. This poses problems if the external format contains information that cannot be represented by the tool's data model. This problem becomes apparent of the external data is represented in *extensible* formats such as XML or RDF.

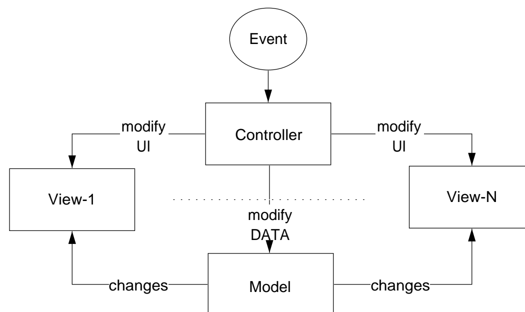


Figure 2.1: Model-View-Controller (MVC) design pattern. Controllers modify UI aspects of a *view* such as zooming, selection, etc. directly. During editing the *controller* modifies the *model* that in turn informs the views. Typically, the data structures of the *Model* are designed with the task of the application in mind.

The MVC design pattern is commonly used and successful. In the context of the Semantic Web, there is an alternative to the proprietary tool data model provided by the stable RDF triple model. This model was designed as an *exchange* model, but the same features that make it good for exchange also make it a good candidate for the internal tool data model. In particular, the *atomic* nature of the model with its standardised semantics ensure the cooperating tools have a sound basis.

In addition to providing a sound basis, the triple approach deals with some serious consistency problems related to more high-level data models. All Semantic Web data can be

expressed precisely and without loss of information by the toolset, while each individual tool can deal with the data using its own way to view the world. For example, it allows an RDFS tool to work flawlessly with an OWL tool, although with limited understanding of the OWL semantics. Different tools can use different subsets of the triple set, possibly doing different types of reasoning. The overall semantics of the triple set however is dictated by stable standards and the atomic nature of the RDF model should minimise interoperability problems. Considering editing and browsing tools, different tools use different levels of abstractions, viewing the plain triples, viewing an RDF graph, viewing an RDFS frame-like representation or an OWL/DL view (figure 2.4, figure 2.5).

Finally, the minimalist data model simplifies general tool operations such as *undo*, save/load, client/server interaction protocols, etc.

In the following architecture section, we show how we deal with the low-level data model in the MVC architecture.

2.4 Architecture

Using a high-level data model that is inspired by the tasks performed by the tools, mapping actions to changes in the data model and mapping these changes back to the UI is relatively straightforward. Using the primitive RDF triple model, mapping changes to the triple store to the views becomes much harder for two reasons. First of all, it is difficult to define concisely and efficiently which changes affect a particular view and second, often considerable reasoning is involved deducing the visual changes from the triples. For example, adding the triple below to a SKOS-based (Miles 2001) thesaurus turns the triple set representing a thesaurus into an RDFS class hierarchy:[1]

```
skos:narrower rdfs:subPropertyOf rdfs:subClassOf .
```

The widgets providing the 'view' have to be consistent with the data. In the example above, adding a single triple changes the semantics of each hierarchy relation in the thesaurus: changes to the triple set and changes to the view can be very indirect. We deal with this problem using *transactions* and *mediators* (Wiederhold 1992).

Both for journaling, undo management, exception handling and maintaining the consistency of views, we introduced transactions. A transaction is a sequence of elementary changes to the triple-base: *add*, *delete* and *update*,[2] labelled with an identifier and optional comments. The comments are used as a human-readable description of the operation (e.g., "Created class Wine"). Transactions can be nested. User interaction with a controller causes a transaction to be started, operations to be performed in the triple-store and finally the transaction to be committed. If anything unexpected happens during the transaction, the changes

[1] Whether this interpretation is desirable is not the issue here.

[2] The *update* change can of course be represented as a delete-and-add, but a separate primitive is more natural, requires less space in the journal and is easier to interpret while maintaining the view consistency.

are discarded, providing protection against partial and inconsistent changes by malfunctioning controllers. A successful transaction results in an *event*.

Simple widgets whose representation depends on one or more direct properties of a resource (e.g., a label showing an icon and label-text for a resource) register themselves as a direct representation of this resource. If an event involves a triple where the resource appears as *subject* or *object*, the widget is informed and typically refreshes itself. Because changes to the property hierarchy can change the interpretation of triples, all simple widgets are informed of such changes.

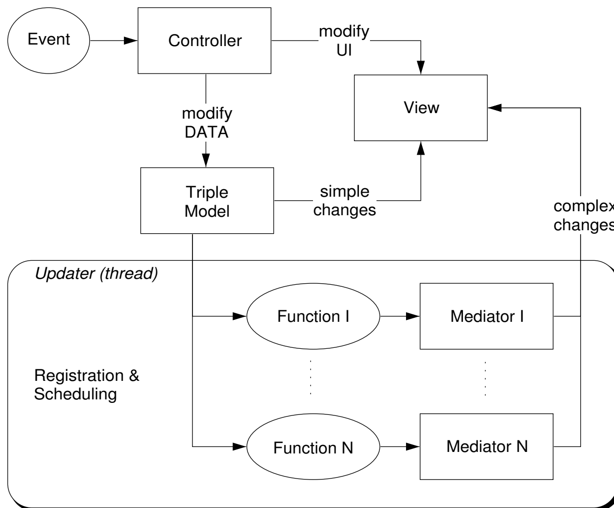


Figure 2.2: Introducing *mediators* to bridge the level of abstraction between triples and view. Update is performed in a different thread to avoid locking the UI.

Complex widgets, such as a hierarchical view, cannot use this schema as they cannot easily define the changes in the database that will affect them and recomputing and refreshing the widget is too expensive for interactive use. It is here that we introduce *mediators*. A *mediator* is an arbitrary Prolog term that is derived from the triple set through a defined function (see figure 2.2). For example, the mediator can be an ordered list of resources that appear as children of a particular node in the hierarchy, while the function is an OWL reasoner that computes the DL class hierarchy. Widgets register a mediator and accompanying function whenever real-time update is considered too expensive. If a mediator is different from the previous result, the controllers that registered the mediator are notified and will update using the high-level representation provided by the mediator. The function and its parameters are registered with the *updater*. The *updater* is running in a separate thread of execution

(chapter 6, Wielemaker 2003a), updating all mediators after each successfully committed transaction. This approach has several advantages.

- Because updating the mediators happens in a separate thread, the UI remains responsive during the update.
- Updates can be aborted as soon as a new transaction is committed.
- Multiple widgets depending on the same mediator require only one computation.
- The updater can schedule on the basis of execution time measured last time, frequency of different results and relation of dependent widgets to the 'current' widget.[3]
- One or multiple update threads can exploit multi-CPU (SMP) hardware as well as schedule updates over multiple threads to ensure that likely and cheap updates are not blocked for a long time by unlikely expensive updates.

2.4.1 Rules to define the GUI

The interface is composed of a hierarchy of widgets, most of them representing one or more resources. We have *compound* and *primitive* widgets. Each widget is responsible for maintaining a consistent view of the triple set as outlined in the previous section. Triple20 widgets have small granularity. For example, most resources are represented by an icon and a textual label. This is represented as a compound widget which controls the icons and displays a primitive widget for the textual label.

In the conventional OO interface each compound widgets decides which member widgets it creates and what their their configuration should be, thus generating the widget hierarchy starting at the outermost widget, i.e., the toplevel window. We have modified this model by having context-sensitive rule sets that are called by widgets to decide on visual aspects as well as define context sensitive menus and perform actions. Rule sets are associated with widget classes. Rules are evaluated similar to OO methods, but following the part-of hierarchy of the interface rather than the subclass hierarchy. Once a rule is found, it may decide to wrap rules of the same name defined on containing widgets similar to sending messages to a superclass in traditional OO.

The advantage of this approach is that widget behaviour can inherit from its containers as well as from the widget class hierarchy. For example, a compound widget representing a set of objects can define rules both for menu-items and the required operations at the data level that deal with the operation *delete*, deleting a single object from the set. Widgets inside the compound 'inherit' the menu item to their popup. This way, instances of a single widget class have different behaviour depending on its context in the interface.

Another example of using rules is shown in figure 2.3, where Triple20 is extended to show SKOS 'part-of' relations in the hierarchy widget using instances of the graphics class 'rdf_part_node', a subclass of 'rdf_node' that displays a label that indicates the part-of relation. The code fragment refines the rule for `child_cache/3`, a rule which defines the

[3] This has not yet been implemented in the current version.

mediator for generating the children of a node in the hierarchy window (shown on an example from another domain in the left frame of figure 2.5). The `display` argument says the rule is defined at the level of display, the outermost object in the widget part-of hierarchy and therefore acts as a default for the entire interface. The `part` argument identifies the new rule set. The first rule defines the mediator for 'parts' of the current node, while the second creates the default mediator. The call to `rdf_cache/3` registers a mediator that is a list of all solutions of *V* of the `rdf/3` goal, where the solutions are sorted alphabetically on their label. *Cache* is an identifier that can be registered by a widget to receive notifications of changes to the mediator.

```
:- begin_rules(display, part).

child_cache(R, Cache, rdf_part_node) :-
        rdf_cache(lsorted(V),
                  rdf(V, skos:broaderPartitive, R), Cache).
child_cache(R, Cache, Class) :-
        super::child_cache(R, Cache, Class).

:- end_rules.
```

Figure 2.3: Redefining the hierarchy expansion to show SKOS part-of relations. This rule set can be loaded without changing anything to the tool.

Rule sets are translated into ordinary Prolog modules using the Prolog preprocessor.[4] They can specify behaviour that is context sensitive. Simple refinement can be achieved by loading rules without defining new widgets. More complicated customisation is achieved by defining new widgets, often as a refinement of existing ones, and modify the rules used by a particular compound widget to create its parts.

2.5 An overview of the Triple20 user interface

RDF documents can be viewed at different levels. Our tool is not a tool to support a particular language such as OWL, but to examine and edit arbitrary RDF documents. It provides several views, each highlighting a particular aspect of the RDF data.

- The *diagram* view (figure 2.4) provides a graph of resources. Resources can be shown as a label (*Noun*) or expanded to a frame (*cycle*). If an elements from a frame is dropped on the diagram a new frame that displays all properties of the element is shown. This tool simply navigates the RDF graph and works on any RDF document.

[4] Realised using `term_expansion/2`.

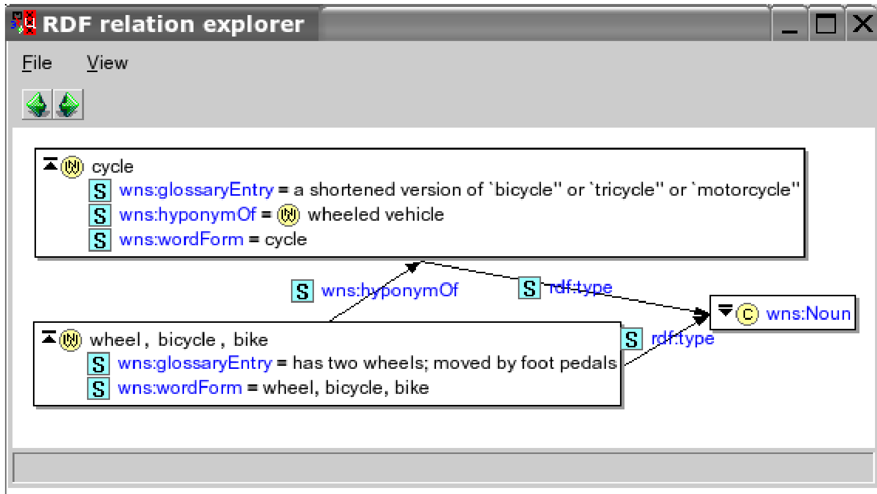


Figure 2.4: Triple20 graph diagram. Resources are shown using just their label or as a frame. Values or properties can be dragged from a frame to the window to expand them.

- The *hierarchy* view (figure 2.5, left window) shows different hierarchies (class, property, individuals) in a single view. The type of expansion is indicated using icons. Expansion can be controlled using *rules* as explained in section 2.4.1.
- A *tabular* view (figure 2.5, right window) allows for multiple resource specific representations. The base system provides an *instance* view and a *class* view on resources.

Editing and browsing are as much as possible integrated in the same interface. This implies that most widgets building the graphical representation of the data are sensitive. Visual feedback of activation and details of the activated resource are provided. In general both menus and drag-and-drop are provided. Context-specific rules define the possible operations dropping one resource onto another. Left-drop executes the default operation indicated in the status bar, while right-drop opens a menu for selecting the operation after the drop. For example, the default for dropping a resource from one place in a hierarchy on another node is to *move* the resource. A right-drop will also offer the option to associate an additional parent.

Drag-and-drop can generally be used to add or modify properties. Before one can drop an object it is required to be available on the screen. This is often impractical and therefore many widgets provide menus to modify or add a value. This interface allows for typing the value using completion, selecting from a hierarchy as well as search followed by selection. An example of the latter is shown in figure 2.6.

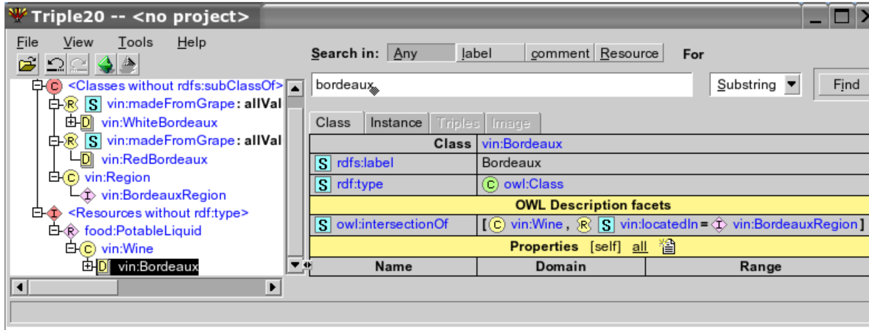


Figure 2.5: Triple20 main window after a search and select.

2.6 Implementation

2.6.1 The source of triples

Our RDF store is actually a quadruple store. The first three fields represent the RDF triple, while the last identifies the source or named graph it is related too. The source is maintained to be able to handle triples from multiple sources in one application, modify them and save the correct triples to the correct destination.

Triple20 includes a library of background ontologies, such as RDFS and OWL as well as some well-known public toplevel ontologies. When a document is loaded which references to one of these ontologies, the corresponding ontology is loaded and flagged 'read-only', meaning no new triples will be added to this source and it is not allowed to delete triples that are associated to it. This implies that trying to delete such a triple inside a transaction causes the operation to be aborted and the other operations inside the transaction to be discarded.

Other documents are initially flagged 'read-write' and new triples are associated to sources based on rules. Actions involving a dialog window normally allow the user to examine and override the system's choice, as illustrated in figure 2.7.

Triple20 is designed to edit triples from multiple sources in one view as it is often desirable to keep each RDF document in its own file(s). If necessary, triples from one file can be inserted into another file.

2.7 Scalability

The aim of Triple20 and the underlying RDF store is to support large ontologies in memory. In-memory storage is much faster than what can be achieved using a persistent store (chapter 3, Wielemaker et al. 2003b) and performance is needed to deal with the low-level reasoning at the triple level. The maximum capacity of the triple store is approximately 20

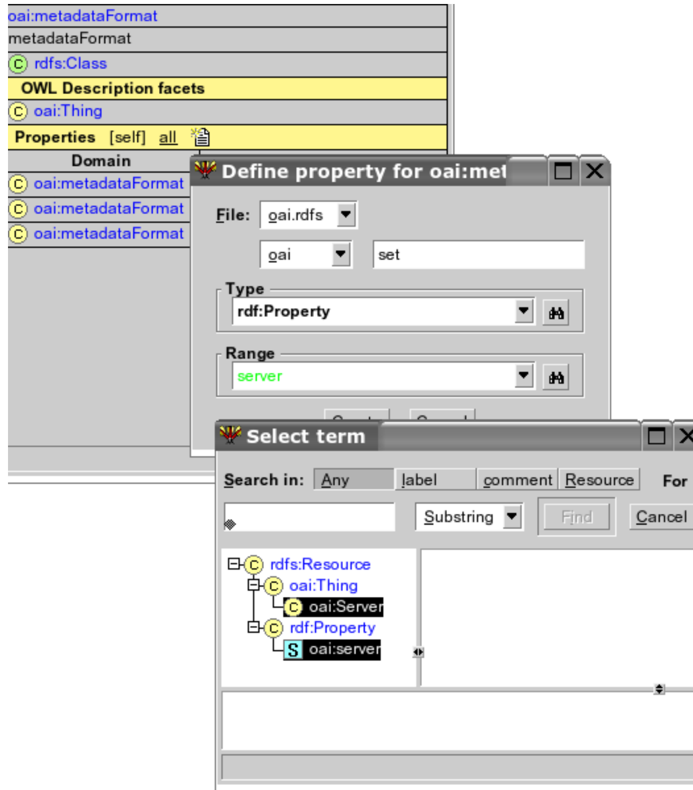


Figure 2.6: Create a property with name *set* and range `oai:Server`. While typing in the *Range* field, the style of the typed text is updated after each keystroke, where bold means 'ok and unique', red is 'no resource with this prefix exists' and green (showed) means 'multiple resources match'. Clicking the *binocular* icon shows all matches in the hierarchy, allowing the user to select.

million triples on 32-bit hardware and virtually unlimited on 64-bit hardware.

We summarise some figures handling WordNet 1.6 (Miller 1995) in RDF as converted by Decker and Melnik. The measurements are taken on a dual AMD 1600+ machine with 2Gb memory running SuSE Linux. The 5 RDF files contain a total of 473,626 triples. The results are shown in table 2.1. For the last test, a small file is added that defines the `wns:hyponymOf` property as a sub property of `rdfs:subClassOf` and defines `wns:LexicalConcept` as a subclass of `rdfs:Class`. This reinterprets the WordNet

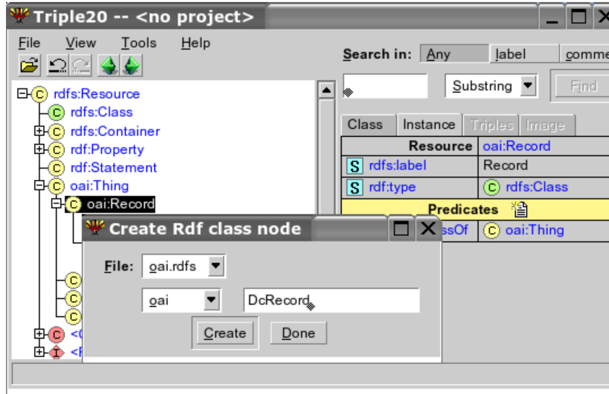


Figure 2.7: Create a new class *DcRecord*. The system proposes the file the class will be saved to (*oai.rdfs*) as well as the namespace (*oai-*)based on the properties of the super class. Both can be changed.

hierarchy as an RDFS class hierarchy. Note that this work is done by the separate update thread recomputing the mediators and thus does not block the UI.

Operation	**Time (sec)**
Load from RDF/XML	65.4
Load from cache	8.4
Re-interpret as class hierarchy	16.3

Table 2.1: Some figures handling WordNet on a dual AMD 1600+ machine. Loading time is proportional to the size of the data.

2.8 Related work

Protégé (Musen et al. 2000) is a landmark in the world of ontology editors. We have described how our design uses the RDF triple model as a basis, where Protégé uses a proprietary internal data model. As a consequence, we can accommodate any RDF document without information loss and we can handle multiple RDF sources as one document without physically merging the source material. Where Protégé is primarily designed as an *editor*, *browsing* is of great importance to Triple20. As a consequence, we have reduced the use of screen-space for controls to the bare minimum, using popup menus and drag-and-drop as primary inter-

action paradigm. Protégé has dedicated support for ontology engineering, which Triple20 lacks.

Miklos et al. 2005 describe how they reuse large ontologies by defining *views* using an F-logic-based mapping. In a way our *mediators*, mapping the complex large triple store to a manageable structure using Prolog can be compared to this, although their purpose is to map one ontology into another, while our purpose is to create a manageable structure suitable for driving the visualisation.

2.9 Discussion

We have realised an architecture for interactive tools that is based directly on the RDF triple model. Using the triples instead of an intermediate representation any Semantic Web document can be represented precisely and tools operating on the data can profit from established RDF-based standards on the same grounds as RDF facilitates exchange between applications. Interface components are only indirectly related to the underlying data model, which makes it difficult to apply the classical model-view-controller (MVC) design pattern for connecting the interface to the data. This can be remedied using *mediators*: intermediate data structures that reflect the interface more closely and are updated using background processing. Mediators are realised as Prolog predicates that derive a Prolog term from the triple database.

With Triple20, we have demonstrated that this design can realise good scalability, providing multiple consistent views (triples, graph, OWL) on the same triple store. Triple20 has been used successfully as a stand-alone ontology editor, as a component in other applications and as a debugging tool for applications running on top of the Prolog triple store, such as ClioPatria (chapter 10).

The presented design is applicable to interactive applications based on knowledge stored as RDF triples (research question 3c). The overall design is language independent, although the natural fit of RDF onto Prolog makes it particularly attractive for our purposes.

Software availability

Triple20 is available under Open Source (LGPL) license from the SWI-Prolog website.[5] SWI-Prolog with graphics runs on MS-Windows, MacOS X and almost all Unix/Linux versions, supporting both 32- and 64-bit hardware.

Acknowledgements

This work was partly supported by the ICES-KIS project “Multimedia Information Analysis” funded by the Dutch government. The Triple20 type-icons are partly taken from and partly inspired by the Protégé project.

[5] http://www.swi-prolog.org/packages/Triple20

Chapter 3

Prolog-based Infrastructure for RDF: Scalability and Performance

About this chapter The core of this chapter has been published at the ISWC-03 (Wielemaker et al. 2003b). This paper has been updated with material from Wielemaker et al. 2007. We also provide an update of performance and scalability figures in section 3.6.

This paper elaborates on research question 1, investigating implementation alternatives for `rdf/3` and associated predicates. The discussion is continued in chapter 4 on query optimisation, while chapter 10 discusses its role in building a Semantic Web search tool.

Abstract The Semantic Web is a promising application-area for the Prolog programming language for its non-determinism and pattern-matching. In this paper we outline an infrastructure for loading and saving RDF/XML, storing triples in memory, and for elementary reasoning with triples. A predecessor of the infrastructure described here has been used in various applications for ontology-based annotation of multimedia objects using Semantic Web languages. Our library aims at fast parsing, fast access and scalability for fairly large but not unbounded applications upto 20 million triples on 32-bit hardware or 300 million on 64-bit hardware with 64Gb main memory.

3.1 Introduction

Semantic-web applications will require multiple large ontologies for indexing and querying. In this paper we describe an infrastructure for handling such large ontologies, This work was done in the context of a project on ontology-based annotation of multi-media objects to improve annotating and querying (chapter 9, Schreiber et al. 2001), for which we use the Semantic Web languages RDF and RDFS. The annotations use a series of existing ontologies, including AAT (Peterson 1994), WordNet (Miller 1995) and ULAN (Getty 2000). To facilitate

this research we require an RDF toolkit capable of handling approximately 3 million triples efficiently on current desktop hardware. This paper describes the parser, storage and basic query interface for this Prolog-based RDF infrastructure. A practical overview using an older version of this infrastructure is in Parsia (2001).

We have opted for a purely memory-based infrastructure for optimal speed. Our tool set can handle the 3 million triple target with approximately 300 Mb. of memory and scales to approximately 20 million triples on 32-bit hardware. Scalability on 64-bit hardware is limited by available main memory and requires approximately 64Gb for 300 million triples. Although insufficient to represent "the whole web", we assume 20 million triples is sufficient for applications operating in a restricted domain such as annotations for a set of cultural-heritage collections.

This document is organised as follows. In section 3.2 we describe and evaluate the Prolog-based RDF/XML parser. Section 3.3 discusses the requirements and candidate choices for a triple storage format. In section 3.4 we describe the chosen storage method and the basic query API. In section 3.5.1 we describe the API and implementation for RDFS reasoning support. This section also illustrates the mechanism for expressing higher level queries. Section 3.6 evaluates performance and scalability and compares the figures to some popular RDF stores.

3.2 Parsing RDF/XML

The RDF/XML parser is the oldest component of the system. We started our own parser because the existing (1999) Java (SiRPAC[1]) and Pro Solutions Perl-based[2] parsers did not provide the performance required and we did not wish to enlarge the footprint and complicate the system by introducing Java or Perl components. The RDF/XML parser translates the output of the SWI-Prolog SGML/XML parser (see section 7.2) into a Prolog list of triples using the steps summarised in figure 3.1. We illustrate these steps using an example from the RDF Syntax Specification document (RDFCore WG 2003), which is translated by the SWI-Prolog XML parser into a Prolog term as described in figure 3.2.

The core of the translation is formed by the second step in figure 3.1, converting the XML DOM structure into an intermediate representation. The intermediate representation is a Prolog term that represents the RDF at a higher level, shielding details such as identifier generation for reified statements, rdf bags, blank nodes and the generation of linked lists from RDF collections from the second step. Considering the rather instable specification of RDF at the time this parser was designed, we aimed at an implementation where the code follows as closely as possible the structure of the RDF specification document.

Because the output of the XML parser is a nested term rather than a list we cannot use DCG. Instead, we designed a structure matching language in the spirit of DCGs, which we introduce with an example. Figure 3.3 shows part of the rules for the pro-

[1] `http://www-db.stanford.edu/~melnik/rdf/api.html`
[2] `http://www.pro-solutions.com/rdfdemo/`

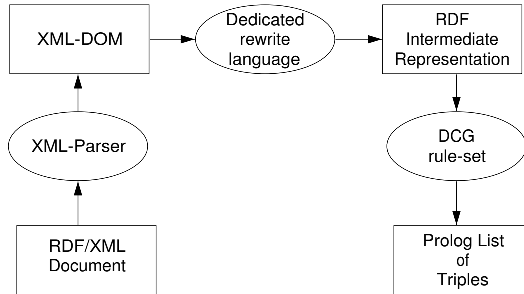


Figure 3.1: Steps converting an RDF/XML document into a Prolog list of triples.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:s="http://description.org/schema/">
  <rdf:Description about="http://www.w3.org/Home/Lassila">
     <s:Creator>Ora Lassila</s:Creator>
  </rdf:Description>
</rdf:RDF>
```

```
[element('http://www.w3.org/1999/02/22-rdf-syntax-ns#':'RDF',
         [xmlns:rdf = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#',
          xmlns:s = 'http://description.org/schema/'
         ],
         [element('http://www.w3.org/1999/02/22-rdf-syntax-ns#':'Description',
                  [about = 'http://www.w3.org/Home/Lassila' ],
                  [ element('http://description.org/schema/':'Creator',
                            [], [ 'Ora Lassila' ])
                  ])
         ])
]
```

Figure 3.2: Input RDF/XML document and output of the Prolog XML Parser, illustrating the input for the RDF parser

duction *parseTypeCollectionPropertyElt*[3] into a Prolog term (intermediate representation) `collection`(*Elements*), where *Elements* holds an intermediate representation for the collection-elements. The body of the rules guiding this process consists of the term that must be matched, optionally followed by raw Prolog code between $\{\ldots\}$, similar to DCG. The matched term can call rule-sets to translate a sub-term using a \ escape-sequence. In figure 3.3, the first rule (propertyElt) matches a term `element`(*Name, Attributes, Content*), iff *Attributes* matches the attribute specification and *Content* can be matched by the `nodeElementList` rule-set.

[3] http://www.w3.org/TR/rdf-syntax-grammar/#parseTypeCollectionPropertyElt

The intermediate representation is translated into a list of `rdf`(*Subject, Predicate, Object*) terms using a set of traditional DCG rules.

```
propertyElt(Id, Name, collection(Elements), Base) ::=
        element(Name,
                \attrs([ \parseCollection,
                         \?idAttr(Id, Base)
                       ]),
                \nodeElementList(Elements, Base)).

parseCollection ::=
        \rdf_or_unqualified(parseType) = 'Collection'.

rdf_or_unqualified(Tag) ::=
        Tag.
rdf_or_unqualified(Tag) ::=
        NS:Tag,
        { rdf_name_space(NS), !
        }.
```

Figure 3.3: Source code of the second step, mapping the XML-DOM structure into an intermediate representation derived from the RDF syntax specification. This fragment handles the `parseType=Collection` element

Long documents cannot be handled this way as both the entire XML structure and the resulting list of RDF triples must fit on the Prolog stacks. To avoid this problem the XML parser can be operated in *streaming* mode. In this mode the RDF parser handles RDF-Descriptions one-by-one, passing the resulting triples to a user-supplied Prolog goal.

The source-code of the parser counts 1170 lines, 564 for the first pass creating the intermediate state, 341 for the generating the triples and 265 for the driver putting it all together. The parser passes the W3C RDF Test Cases[4].

3.3 Storing RDF triples: requirements and alternatives

3.3.1 Requirement from integrating different ontology representations

Working with multiple ontologies created by different people and/or organisations poses some specific requirements for storing and retrieving RDF triples. We illustrate with an example from our own work on annotating images (chapter 9, Schreiber et al. 2001).

Given absence of official RDF versions of AAT and IconClass we created our own RDF representation, in which the concept hierarchy is modelled as an RDFS class hierarchy. We wanted to use these ontologies in combination with the RDF representation of WordNet created by Decker and Melnik.[5] However, their RDF Schema for WordNet defines classes and

[4] `http://www.w3.org/TR/2003/WD-rdf-testcases-20030123/`
[5] `http://www.semanticweb.org/library/`

properties for the metamodel of WordNet. This means that WordNet *synsets* (the basic WordNet concepts) are represented as instances of the (meta)class `LexicalConcept` and that the WordNet hyponym relations (the subclass relations in WordNet) are represented as tuples of the meta property `hyponymOf` relation between instances of `wns:LexicalConcept`. This leads to a representational mismatch, as we are now unable to treat WordNet concepts as classes and WordNet hyponym relations as subclass relations.

Fortunately, RDFS provides metamodeling primitives for coping with this. Consider the following two RDF descriptions:

```
<rdf:Description rdf:about="&wns;LexicalConcept">
  <rdfs:subClassOf rdf:resource="&rdfs;Class"/>
</rdf:Description>

<rdf:Description rdf:about="&wns;hyponymOf">
  <rdfs:subPropertyOf rdf:resource="&rdfs;subClassOf"/>
</rdf:Description>
```

The first statement specifies that the class `LexicalConcept` is a subclass of the built-in RDFS metaclass `Class`, the instances of which are classes. This means that now all instances of `LexicalConcept` are also classes. In a similar way, the second statement defines that the WordNet property `hyponymOf` is a subproperty of the RDFS subclass-of relation. This enables us to interpret the instances of `hyponymOf` as subclass links.

We expect representational mismatches to occur frequently in any realistic semantic-web setting. RDF mechanisms similar to the ones above can be employed to handle this. However, this poses the requirement on the toolkit that the infrastructure is able to interpret subtypes of `rdfs:Class` and `rdfs:subPropertyOf`. In particular the latter is important for our applications.

3.3.2 Requirements

Based on our lessons learned from earlier annotation experiments as described in section 9.3 we state the following requirements for the RDF storage module.

Efficient subPropertyOf handling As illustrated in section 3.3.1, ontology-based annotation requires the re-use of multiple external ontologies. The `subPropertyOf` relation provides an ideal mechanism to re-interpret an existing RDF dataset.

Avoid frequent cache updates In our first prototype we used secondary store based on the RDFS data model to speedup RDFS queries. The mapping from triples to this model is not suitable for incremental update, resulting in frequent slow re-computation of the derived model from the triples as the triple set changes.

Scalability At the time of design, we had access to 1.5 million triples in vocabularies. Together with actual annotations we aimed at storing 3 million triples on a (then) commodity notebook with 512 Mb main memory. Right now,[6] we have access to over 30 million triples and we plan to support 300 million triples in commodity server hardware with 8 cores and 64 Gb main memory.

Fast load/save At the time of design, the RDF/XML parsing and loading time for 1.5 million triples was 108 seconds. This needs to be improved by an order of magnitude to achieve reasonable startup times, especially for interactive development.

3.3.3 Storage options

The most natural way to store RDF triples is using facts of the format `rdf`(*Subject, Predicate, Object*) and this is, except for a thin wrapper improving namespace handling, the representation used in our first prototype. As standard Prolog systems only provide indexing on the first argument this implies that asking for properties of a subject is indexed, but asking about inverse relations is slow. Many queries involve reverse relations: "what are the sub-classes of X?". "what instances does Y have?", "what subjects have label L?" are queries commonly used in our annotation tool.

Our first tool (section 9.1) solved these problems by building a secondary Prolog database following the RDFS data model. The cached relations included `rdfs_class`(*Class, Super, Meta*). `rdfs_property`(*Class, Property, Facet*), `rdf_instance`(*Resource, Class*) and `rdfs_label`(*Resource, Label*). These relations can be accessed quickly in any direction. This approach has a number of drawbacks. First of all, the implications of even adding or deleting a single triple are potentially enormous, leaving the choice between complicated incremental update of the cache with the triple set or frequent slow total recompute of the cache. Second, storing the cache requires considerable memory resources and third, it is difficult to foresee for which derived relations caching is required because this depends on the structure and size of the triple set as well as the frequent query patterns.

In another attempt we used *Predicate*(*Subject, Object*) as database representation and stored the inverse relation as well in *InversePred*(*Object, Subject*) with a wrapper to call the 'best' version depending on the runtime instantiation. Basic triple query is fast, but queries that involve an unknown predicate or need to use the predicate hierarchy (first requirement) cannot be handled efficiently. When using a native Prolog representation, we can use Prolog syntax for caching parsed RDF/XML files. Loading triples from Prolog syntax is approximately two times faster than RDF/XML, which is insufficient to satisfy our fourth requirement.

Using an external DBMS for the triple store is an alternative. Assuming an SQL database, there are three possible designs. The simplest one is to use Prolog reasoning and simple `SELECT` statements to query the DB. This approach does not exploit query optimisation

[6] November 2008

and causes many requests involving large amounts of data. Alternatively, one could either write a mixture of Prolog and SQL or automate part of this process, as covered by the Prolog to SQL converter of Draxler (1991). Our own (unpublished) experiences indicate a simple database query is at best 100 and in practice often over 1,000 times slower than using the internal Prolog database. Query optimisation is likely to be of limited effect due to poor handling of transitive relations in SQL. Many queries involve `rdfs:subClassOf`, `rdfs:subPropertyOf` and other transitive relations. Using an embedded database such as BerkeleyDB[7] provides much faster simple queries, but still imposes a serious efficiency penalty. This is due to both the overhead of the formal database API and to the mapping between the in-memory Prolog atom handles and the RDF resource representation used in the database.

In the end we opted for a Prolog *foreign-language* extension: a module written in C to extend the functionality of Prolog.[8] A significant advantage using an extension to Prolog rather than a language independent storage module separated by a formal API is that the extension can use native Prolog atoms, significantly reducing memory requirements and access time.

3.4 Realising an RDF store as C-extension to Prolog

3.4.1 Storage format

Triples are stored as a C-structure holding the three fields and 6 'next' links, one for the linked list that represents all triples as a linear list and 5 hash-tables links. The 5 hash-tables cover all instantiation patterns with at least one field instantiated, except for all fields instantiated (+,+,+) and subject and object instantiated (+,-,+). Indexing of fully is less critical because a fully instantiated query never produces a choicepoint. Their lookup uses the (+,+,-) index. Subject and object instantiated queries use the (+,-,-) index.[9] The size of the hash-tables is automatically increased as the triple set grows. In addition, each triple is associated with a *source-reference* consisting of an atom (normally the filename) and an integer (normally the line-number) and a general-purpose set of flags, adding up to 13 machine words (52 bytes on 32-bit hardware) per triple, or 149Mb for the intended 3 million triples. Our reference-set of 1.5 million triples uses 890,000 atoms. In SWI-Prolog an atom requires 7 machine words overhead excluding the represented string. If we estimate the average length of an atom representing a fully qualified resource at 30 characters the atom-space required for the 1.8 million atoms in 3 million triples is about 88Mb. The required total of 237Mb for 3 million triples fits in 512Mb.

[7] `http://www.sleepycat.com/`

[8] Extending Prolog using modules written in the C-language is provided in most todays Prolog systems although there is no established standard foreign interface and therefore the connection between the extension and Prolog needs to be rewritten when porting to other implementation of the Prolog language (Bagnara and Carro 2002).

[9] On our usage pattern this indexing schema performs good. Given that that database maintains statistics on the used indexing, we can consider to make existence and quality (size of the hash-table) dynamic in future versions.

To accommodate active queries safely, deletion of triples is realised by flagging them as *erased*. Garbage collection can be invoked if no queries are active.

3.4.1.1 Indexing

Subjects and resource *Objects* use the immutable atom-handle as hash-key. The *Predicate* field needs special attention due to the requirement to handle `rdfs:subPropertyOf` efficiently. Each predicate is a first class citizen and is member of a *predicate cloud*, where each cloud represents a graph of predicates connected through `rdfs:subPropertyOf` relations. The cloud reference is used for indexing the triple. The cloud contains a reachability matrix that contains the transitive closure of the `rdfs:subPropertyOf` relations between the member predicates. The clouds are updated dynamically on assert and retract of `rdfs:subPropertyOf` triples. The system forces a re-hash of the triples if a new triple unites two clouds, both of which represent triples, or when deleting a triple splits a cloud in two non-empty clouds. As a compromise to our requirements, the storage layer must know the fully qualified resource for `rdfs:subPropertyOf` and must rebuild the predicate hierarchy and hash-tables if `rdfs:subPropertyOf` relations join or split non-empty predicate clouds. The index is re-build on the first indexable query. We assume that changes to the `rdfs:subPropertyOf` relations are infrequent.

RDF literals have been promoted to first class citizens in the database. Typed literals are supported using arbitrary Prolog terms as RDF object. All literals are kept in an AVL-tree, where

$$numericliterals < stringliterals < termliterals$$

. Numeric literals are sorted by value. String literals are sorted alphabetically, case insensitive and after removing UNICODE diacritics. String literals that are equal after discarding case and diacritics are sorted on UNICODE code-point. Other Prolog terms are sorted on Prolog standard order of terms. Sorted numeric literals are used to provide indexed search for dates. Sorted string literals are used for fast prefix search which is important for suggestions and disambiguation as-you-type with AJAX style interaction (chapter 10, Wielemaker et al. 2008). The core database provides indexed prefix lookup on the entire literals. Extended with the literal search facilities described above, it also supports indexed search on tokens and prefixes of tokens that appear in literals.

The literal search facilities are completed by means of *monitors*. Using `rdf_monitor`(*:Goal, +Events*) we register a predicate to be called at one or more given events. Monitors that trigger on literal creation and destruction are used to maintain a word-index for the literals as well as an index from stem to word and *metaphone* (Philips 2000) key to word.

The above representation provides fully indexed lookup using any instantiation pattern (mode) including sub-properties. Literal indexing is case insensitive and supports indexed prefix search as well as indexed search on ranges of numerical values.

3.4.2 Concurrent access

Multi-threading is supported by means of *read-write locks* and *transactions*. During normal operation, multiple readers are allowed to work concurrently. Transactions are realised using `rdf_transaction`(*:Goal, +Context*). If a transaction is started, the thread waits until other transactions have finished. It then executes *Goal*, adding all write operations to an agenda. During this phase the database is not actually modified and other readers are allowed to proceed. If *Goal* succeeds, the thread waits until all readers have completed and updates the database. If *Goal* fails or throws an exception the agenda is discarded and the failure or error is returned to the caller of `rdf_transaction/2`. Note that the `rdf/3` transaction-based update behaviour differs from the multi-threaded SWI-Prolog logical update behaviour defined for dynamic predicates:

- *Prolog dynamic predicates*
  In multi-threaded (SWI-)Prolog, accessing a dynamic predicate for read or write demands synchronisation only for a short time. In particular, readers with an open choice-point on the dynamic predicate allow other threads to update the same predicate. The standard Prolog logical update semantics are respected using time-stamps and keeping erased clauses around. Erased clauses are destroyed by a garbage collector that is triggered if the predicate has a sufficient number of erased clauses and is not in use.

- RDF-DB *transactions*
  Multiple related modifications are bundled in a transaction. This is often desirable as many high-level (RDFS/OWL) changes involve multiple triples. Using transactions guarantees a consistent view of the database and avoids incomplete modifications if a sequence of changes is aborted.

3.4.3 Persistency and caching

Loading RDF from RDF/XML is slow, while quickly loading RDF databases is important to reduce application startup times. This is particularly important during program development. In addition, we need a persistent store to accumulate loaded RDF and updates such as human edits. We defined a binary file format that is used as a cache for loading external RDF resources and plays a role in the persistent storage.

Although attractive, storing triples using the native representation of Prolog terms (i.e., terms of the form `rdf`(*Subject, Predicate, Object*)) does not provide the required speedup, while the files are, mainly due to the expanded namespaces, larger than the RDF/XML source. Unpublished performance analysis of the Prolog parser indicates that most of the time is spent in parsing text to atoms used to represent RDF resources. Typically, the same resource appears in multiple triples and therefore we must use a serialisation that performs the expensive conversion from text to atom only once per unique resource. We use a simple incremental format that includes the text of the atom only the first time that the atom needs

to be saved. Later references to the same atom specify the atom as the N-th atom seen while loading this file. An atom on the file thus has two formats: `A` ⟨*length*⟩ ⟨*text*⟩ or `X` ⟨*integer*⟩. Saving uses a hash table to keep track of already saved atoms, while loading requires an array of already-loaded atoms. The resulting representation has the same size as the RDF/XML within 10%, and loads approximately 20 times faster.

The predicate `rdf_load`(+*SourceURL*) can be configured to maintain a cache that is represented as a set of files in the above format.

A persistent backup is represented by a directory on the filesystem and one or two files per named graph: an optional *snapshot* and an optional *journal*. Both files are maintained through the monitor mechanism introduced in section 3.4.1.1 for providing additional indices for literals. If an initial graph is loaded from a file, a *snapshot* is saved in the persistent store using the fast load/save format described above. Subsequent addition or deletion of triples is represented by Prolog terms appended to the *journal* file. The journal optionally stores additional information passed with the transaction such as the time and user, thus maintaining a complete changelog of the graph. If needed, the current journal can be merged into the snapshot. This reduces disk usage and enhances database restore performance, but looses the history.

3.4.4 API rationale

The API is summarised in table 3.2. The predicates support the following datatypes:

- *Resource*
  A fully qualified resource is represented by an atom. Prolog maps atoms representing a string uniquely to a handle and implements comparison by comparing the handles. As identity is the only operation defined on RDF resources, this mapping is perfect. We elaborate on namespace handling later in this section.

- `literal`*(Literal)*
  A literal is embedded in a term `literal/1`. The literal itself can be any Prolog term. Two terms have a special meaning: `lang`(*LangID, Text*) and `type`(*TypeIRI, Value*).

The central predicate is `rdf/3`, which realises a *pure*[10] Prolog predicate that matches an edge in the RDF graph. In many cases it is desirable to provide a match specification other than simple exact match for literals, which is realised using a term `literal`(+*Query*, -*Value*) (see `rdf/3` in table 3.2). RDFS `subPropertyOf` entailment (requirement one of section 3.3.2) is realised by `rdf_has/4`.

[10] A pure predicate is a predicate that behaves consistently, regardless of the instantiation pattern. Conjunctions of pure predicate can be ordered freely without affecting the semantics.

Declarativeness Many types of reasoning involve transitive relations such as `rdfs:subClassOf` which are allowed to contain cycles. Normal Prolog non-recursion to express the transitive closure of a relation as illustrated in figure 3.4 (top) does not terminate if the relation contains cycles. Simple cases can be fixed easily by maintaining a set of visited resources. Complicated mutual recursive definitions can be handled transparently in Prolog systems that provide tabling (Ramakrishnan et al. 1995). Currently, we provide `rdf_reachable/3` which provides cycle-safe breadth-first search to simplify some of the coding (figure 3.4, bottom).

```
rdfs_subclass_of(Class, Class).
rdfs_subclass_of(Class, Super) :-
        rdf_has(Class, rdfs:subClassOf, Super0),
        rdfs_subclass_of(Super0, Super).
```

```
rdfs_subclass_of(Class, Super) :-
        rdf_reachable(Class, rdfs:subClassOf, Super).
```

Figure 3.4: Coding a transitive relation using standard Prolog recursion does not terminate (top) because most relations may contain cycles. The predicate `rdf_reachable/3` (bottom) explores transitive relations safely.

Namespace handling Fully qualified resources are long, hard to read and difficult to maintain in application source-code. On the other hand, representing resources as atoms holding the fully qualified resource is attractive because it is compact and compares fast.

We unite the advantage of fully qualified atoms with the compactness in the source of ⟨*NS*⟩:⟨*Identifier*⟩ using macro-expansion based on Prolog `goal_expansion/2` rules. For each of the arguments that can receive a resource, a term of the format ⟨*NS*⟩:⟨*Identifier*⟩, where ⟨*NS*⟩ is a registered abbreviation of a namespace and ⟨*Identifier*⟩ is a local name, is mapped to the fully qualified resource.[11] The predicate rdf_db:ns/2 maps registered short local namespace identifiers to the fully qualified namespaces. The initial definition contains the well-known abbreviations used in the context of the Semantic Web. See table 3.1. The mapping can be extended using `rdf_register_ns/2`.

With these declarations, we can write the following to get all individuals of `http://www.w3.org/2000/01/rdf-schema#Class` on backtracking:

```
?- rdf(X, rdf:type, rdfs:'Class').
```

[11] In a prototype of this library we provided a more powerful version of this mapping at runtime. In this version, output-arguments could be split into their namespace and local name as well. After examining actual use of this extra facility in the prototype and performance we concluded a limited compile-time alternative is more attractive.

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

Table 3.1: Initial registered namespace abbreviations

3.5 Querying the RDF store

3.5.1 RDFS queries

Queries at the RDFS level are provided as a library implemented using Prolog rules exploiting the API primitives in table 3.2. For example the code in, figure 3.5 realises testing and generating of individuals. The first rule tests whether an individual belongs to a given class or generates all classes the individual belongs to. The second rule generates all individuals that belong to a specified class. The last rule is called in the unbound condition. There is not much point generating all classes and all individuals that have a type that is equal to or a subclass of the generated class and therefore we generate a standard Prolog exception.

```
rdfs_individual_of(Resource, Class) :-
        nonvar(Resource), !,
        rdf_has(Resource, rdf:type, MyClass),
        rdfs_subclass_of(MyClass, Class).
rdfs_individual_of(Resource, Class) :-
        nonvar(Class), !,
        rdfs_subclass_of(SubClass, Class),
        rdf_has(Resource, rdf:type, SubClass).
rdfs_individual_of(_Resource, _Class) :-
        throw(error(instantiation_error, _)).
```

Figure 3.5: Implementation of rdfs_individual_of/2

rdf(_?Subject, ?Predicate, ?Object_**)**
Elementary query for triples. _Subject_ and _Predicate_ are atoms representing the fully qualified URL of the resource. _Object_ is either an atom representing a resource or `literal`(_Text_) if the object is a literal value. For querying purposes, _Object_ can be of the form `literal`(+_Query, -Value_), where _Query_ is one of

exact(+_Text_**)**
Perform exact, but case-insensitive match. This query is fully indexed.

substring(+_Text_**)**
Match any literal that contains _Text_ as a case-insensitive substring.

word(+_Text_**)**
Match any literal that contains _Text_ as a 'whole word'.

prefix(+_Text_**)**
Match any literal that starts with _Text_.

rdf_has(_?Subject, ?Predicate, ?Object, -TriplePred_**)**
This query exploits the `rdfs:subPropertyOf` relation. It returns any triple whose stored predicate equals _Predicate_ or can reach this by following the transitive `rdfs:subPropertyOf` relation. The actual stored predicate is returned in _TriplePred_.

rdf_reachable(_?Subject, +Predicate, ?Object_**)**
True if _Object_ is, or can be reached following the transitive property _Predicate_ from _Subject_. Either _Subject_ or _Object_ or both must be specified. If one of _Subject_ or _Object_ is unbound this predicate generates solutions in breadth-first search order. It maintains a table of visited resources, never generates the same resource twice and is robust against cycles in the transitive relation.

rdf_subject(_?Subject_**)**
Enumerate resources appearing as a subject in a triple. The reason for this predicate is to generate the known subjects _without duplicates_ as one would get using `rdf`(_Subject, _, __). The storage layer ensures the first triple with a specified _Subject_ is flagged as such.

rdf_transaction(_:Goal, +Context_**)**
Run _Goal_, recording all database modifying operations. Commit the operations if _Goal_ succeeds, discard them otherwise. _Context_ may associate addition information for the persistency layer as well as providing feedback to the user in interactive applications. See section 3.4.2.

rdf_assert(_+Subject, +Predicate, +Object_**)**
Assert a new triple into the database. _Subject_ and _Predicate_ are resources. _Object_ is either a resource or a term `literal`(_Value_).

rdf_retractall(_?Subject, ?Predicate, ?Object_**)**
Removes all matching triples from the database.

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

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

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

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

Table 3.2: API summary for accessing the triple store

3.5.2 Application queries

We study the Prolog implementation of some queries on WordNet 1.6 (Miller 1995) in RDF as converted by Decker and Melnik to get some insight in how queries are formulated and what performance considerations must be taken into account. Timing is performed on an AMD 1600+ processor. Consider the question 'Give me an individual of WordNet 'Noun' labelled *right*'. This non-deterministic query can be coded in two ways, which are semantically equivalent:

```
right_noun_1(R) :-
        rdfs_individual_of(R, wns:'Noun'),
        rdf_has(R, rdfs:label, literal(right)).

right_noun_2(R) :-
        rdf_has(R, rdfs:label, literal(right)),
        rdfs_individual_of(R, wns:'Noun').
```

The first query enumerates the subclasses of `wns:Noun`, generates their 66025 individuals and tests each for having the literal 'right' as label. The second generates the 8 resources in the 0.5 million triples with label 'right' and tests them to belong to `wns:Noun`. The first query requires 0.17 seconds and the second 0.37 milli-seconds to generate all alternatives, a $460\times$ speedup. Query optimisation by ordering goals in a conjunction is required for good performance. Automatic reordering of conjunctions is discussed in chapter 4 (Wielemaker 2005) in the context of implementing the Semantic Web query languages SeRQL (Broekstra et al. 2002) and SPARQL (Prud'hommeaux and Seaborne 2008) on top of this library.

3.5.3 Performance of rdf/3 and rdf_has/4

The most direct way to describe the query performance is to describe the metrics of the core API functions: `rdf/3` and `rdf_has/4`. Other reasoning must be defined on top of these primitives and as we are faced with a large variety of potential tasks that can be fulfilled using a large variety of algorithms that are not provided with the RDF-DB library it is beyond the scope of this paper to comment on the performance for high-level queries.

We examined two queries using WordNet 1.6, executed on an AMD 1600+ CPU with 2Gb main memory. First we generated all solutions for `rdf`(*X, rdf:type, wns:'Noun'*). The 66,025 nouns are generated in 0.046 seconds (1.4 million alternatives/second). Second we asked for the type of randomly generated nouns. This deterministic query is executed at 526,000 queries/second. Tests comparing `rdf/3` with `rdf_has/4`, which exploits the `rdfs:subPropertyOf` relation show no significant difference in performance. In summary, the time to setup a query and come with the first answer is approximately $2\mu s$ and the time to generate subsequent solutions is approximately $0.7\mu s$ per solution.

3.6 Performance and scalability comparison

Comparing the performance of RDF stores has several aspects. First, there is the data. Here we see a number of reference datasets,[12] such as the Lehigh University Benchmark (Guo et al. 2004). Second, there are the queries, which are closely related to the tasks for which we wish to use the RDF store. Examples of tasks are graph-exploration algorithms such as discussed in chapter 10, graph pattern matching as supported by SPARQL and (partial) OWL reasoning as supported by OWLIM (Kiryakov et al. 2005). Third, there is the different basic design which depends in part on the intended usage. We identified six design dimensions as listed below. This list has some overlap with section 3.3.3, but where the discussion in section 3.3.3 is already focused because of the requirements (section 3.3.2) and the choice for Prolog, the list below covers design choices made by a wider range of RDF stores which we can consider to include into our comparison.

- *Memory vs. disk-based*
  Disk-based storage allows for much larger amounts of triples and can ensure short application startup times. However, access is several orders of magnitude slower. This can be compensated for by using caching, pre-computing results (forward chaining) and additional indexes. Because our aim is to provide a platform where we can prototype different reasoning mechanisms and support updates to the database, pre-computing is unattractive. Many RDF stores can be configured for either disk-based or memory-based operation. Because of the totally different performance curve and storage limits there is not much point comparing our memory-based infrastructure with disk-based implementations.

- *Node vs. relational model*
  RDF can be stored as a graph in memory, where nodes link directly to their neighbours or as a table of RDF triples. Both these models can be used on disk and in memory. The SWI-Prolog RDF store uses the relational model. We have no information on the other stores.

- *Read-only vs. read/write*
  Systems differ widely in the possibility to update the database. Some offer no update at all (e.g., BRAHMS, Janik and Kochut 2005); systems using forward reasoning provide much slower updates than systems doing backward reasoning only, notably on delete operations. Because annotation is one of our use cases, changes are not very frequent but must be supported. Read-only databases are generally faster and provide more compact storage because they can use fixed-size data structures (e.g., arrays) and require no locking for concurrent access.

- *Forward vs. backward reasoning*
  Forward reasoning systems pre-compute (part of) the entailment and can therefore

[12] See also `http://esw.w3.org/topic/RdfStoreBenchmarking`

answer questions that require that entailment instantaneously. The price is a higher memory footprint, slower loading, slow updates and more difficult (or lack of) support to enable/disable specific entailment (see section 10.5.0.5). This model fits poorly with our use cases and the Prolog backward reasoning bias.

- *Built-in support for* RDFS, OWL*, etc.*
  Especially stores using backward reasoning may have built-in support for specific inferences. Our store has special support for `rdfs:subPropertyOf` (rdf_has/4) and transitive properties (rdf_reachable/3). Others provide support for identity mapping using `owl:sameAs`.

- *Direct access to the triple store vs. query-language only*
  Some stores can be deployed as a library which provided direct access to the triple store. Typically these stores provide an *iterator* that can be used to iterate over all triples in the store that match some pattern. This is comparable to our `rdf/3` predicate that iterates on backtracking over matching triples. Other stores only provide access through a query language such as SPARQL and some can be used in both modes.

Each value on each of the above six dimensions have advantages and disadvantages for certain classes of RDF-based applications, and it should therefore not come as a surprise that it is hard to compare RDF stores.

In 2005, Janik and Kochut (2005) compared the main-memory RDF store BRAHMS on load time, memory usage and two graph-exploration tasks to several RDF stores that provide a direct API to the triple store from the language in which they are implemented (see last bullet above). A direct API is important for their use case: searching RDF graphs for long associative relations. This use case compares well to the general purpose graph-exploration we aim at (see section 10.3) and therefore we decided to use the same RDF stores and datasets. The data (except for the 'small synthetic' mentioned in the paper) is all publically available. For the RDF stores we downloaded the latest stable version. We added SwiftOWLIM (OWLIM using main memory storage) because it is claimed[13] to be a very fast and memory efficient store. Data, stores and versions are summarised in table 3.3.

Our hypothesis was that the hardware used Janik and Kochu (dual Intel Xeon@3.06Ghz) is comparable to ours (Intel X6800@2.93Ghz) and we could add our figures to the tables presented in their paper. After installing BRAHMS and re-running some of the tests we concluded this hypothesis to be false. For example, according to Janik and Kochut (2005), BRAHMS required 363 seconds to load the 'big SWETO' dataset while our timing is 37 seconds. BRAHMS search tests run 2-4 times faster in our setup with a significant variation. Possible explanations for these differences are CPU, compiler (gcc 3.2.2 vs. gcc 4.3.1), 32-bit vs. 64-bit version and system load. Dependency on system load cannot be excluded because examining the source for the timings on BRAHMS learned us that the reported time is wall time. Possible explanations for the much larger differences in load time are enhancements to the Raptor parser (the version used in the BRAHMS paper is unknown) and I/O

[13] `http://www.ontotext.com/owlim/OWLIMPres.pdf`

System	Remarks	
Jena version 2.5.6 (McBride 2001)		
Sesame version 2.2.1 (Broekstra et al. 2002)	Storage (*sail*): built-in memory	
Sesame version 2.2.1 (Broekstra et al. 2002)	Storage (*sail*): OWLIM 3.0beta	
Redland version 1.0.8 (Beckett 2002)	Parser: Raptor 1.4.18; in-memory trees	
BRAHMS version 1.0.1 (Janik and Kochut 2005)	Parser: Raptor 1.4.18	
SWI-Prolog version 5.7.2		
Dataset	**file size**	**#triples**
Small SWETO (Aleman-Meza et al. 2004)	14Mb	187,505
Big SWETO	244Mb	3,196,692
Univ(50,0) (Guo et al. 2004)	533Mb	6,890,961

Table 3.3: Systems and datasets used to evaluate load time and memory usage. The Raptor parser is part of the Redland suite described in the cited paper.

speed. Notably because time was measured as wall time differences between, for example local disk and network disk can be important.

Given availability of software and expertise to re-run the load time and memory usage experiments, we decided to do so using the currently stable version of all software and using the same platform for all tests. Section 3.6.1 describes this experiment.

All measurements are executed on an Intel core duo X6800@2.93Ghz, equipped with 8Gb main memory and running SuSE Linux 11.0. C-based systems where compiled with gcc 4.3.1 using optimisation settings as suggested by the package configure program and in 64-bit mode (unless stated otherwise); Java-based systems where executed using SUN Java 1.6.0 (64-bit, used -Xmx7G to specify 7Gb heap limit).

3.6.1 Load time and memory usage

Load time and memory usage put a limit on the scalability of main-memory-based RDF stores. Load time because it dictates the startup time of the application and memory usage because it puts a physical limit on the number of triples that can be stored. The amount of memory available to 32-bit applications is nowadays dictated by the address space granted to user processes by the OS and varies between 2Gb and 3.5Gb. On 64-bit machines it is limited by the amount of physical memory. In 2008, commodity server hardware scales to 64Gb; higher amounts are only available in expensive high-end hardware.

Table 3.4 shows both the load times for loading RDF/XML and for loading the proprietary cache/images formats. Redland was not capable of processing the Univ(50,0) dataset due to a ‘fatal error adding statements’. We conclude that our parser is the slowest of the tested ones; only by a small margin compared to Jena and upto five times slower compared to Raptor+BRAHMS. Based on analysing the SWI-Prolog profiler output we conclude that

the interpreted translation from XML to RDF triples (section 3.2) and especially the translation of resources to their final IRI format provides significant room for improvement. Janik and Kochut 2005 show extremely long load times for Univ(50,0) for both Jena and Sesame (Sesame: 2820 seconds for Univ(50,0) vs. 158 for big SWETO; nearly 18 times longer for loading only a bit over 2 times more data). A likely cause is garbage collection time because both systems where operating close to their memory limit on 32-bit hardware. We used a comfortable 7Gb Java heap limit on 64-bit hardware and did not observe the huge slowdown reported by Janik and Kochu. SWI-Prolog's infrastructure for loading RDF/XML is not subject to garbage collection because the parser operates in *streaming* mode and the RDF store provides its own memory management.

The SWI-Prolog RDF-DB cache format loads slower than BRAHMS images. One explanation is that BRAHMS internalises RDF resources as integers that are local to the image. In SWI-Prolog, RDF resources are internalised as Prolog atoms that must be resolved at load time. BRAHMS can only load exactly one image, while the SWI-Prolog RDF-DB can load any mixture of RDF/XML and cache files in any order.

Table 3.5 shows the memory usage each of the tested RDF stores. Results compare well to Janik and Kochu when considering the 32-bit vs. 64-bit versions of the used software. SWI-Prolog RDF-DB uses slightly more memory than BRAHMS, which is to be expected as BRAHMS is a read-only database, while SWI-Prolog's RDF-DB is designed to allow for updates. It is not clear to us why the other systems use so much more memory, particularly so because the forward reasoning of all systems has been disabled. Table 3.5 also provides the figures for the 32-bit version of SWI-Prolog. As all memory except for some flag fields and the resource string consists of pointers, memory usage is almost 50% lower.

	Small SWETO	Big SWETO	Univ(50,0)
Jena	7.4	120	180
Sesame	5.7	88	127
Sesame – OWLIM	4.1	63	104
Redland	3.1	66	–
BRAHMS – create image	1.8	37	69
BRAHMS – load image	0.1	1	1
SWI – initial	8.2	161	207
SWI – load cache	0.4	11	19
SWI – Triples/second	22,866	19,887	29,701

Table 3.4: RDF File load time from local disk in seconds. Reported time is CPU time, except for the BRAHMS case, which is wall time because we kept the original instrumentation. System load was low.

	Small SWETO	Big SWETO	Univ(50,0)
Jena	304	2669	2793
Sesame	141	2033	3350
Sesame – OWLIM	143	1321	2597
Redland	96	1514	–
BRAHMS	31	461	714
SWI	36	608	988
SWI – 32-bit	24	365	588

Table 3.5: Memory usage for RDF file load in Mb. Except for the last row, all systems were used in 64-bit mode.

3.6.2 Query performance on association search

Janik and Kochut 2005 evaluate the performance by searching all paths between two given resources, without considering paths to schema information, in an RDF graph. They use two algorithms for this: depth-first search (DFS) and bi-directional breadth first search (bi-BFS). We implemented both algorithms in Prolog on top of the SWI-Prolog RDF-DB store. Implementing DFS is straightforward, but the implementation of bi-BFS leaves more options. We implemented a faithful translation of the C++ bi-BFS implementation that is part of the examples distributed with BRAHMS.

DFS We replicated the DFS experiment with BRAHMS and SWI-Prolog using the software and hardware described on page 43. The results for the other stores are the values reported by Janik and Kochu, multiplies by the performance difference found between their report on BRAHMS and our measurements ($3.3\times$). We are aware that these figures are only a rough approximations. In the Prolog version, `rdf/3` is responsible for almost 50% of the CPU time. Memory requirements for DFS are low and the algorithm does not involve garbage collection. BRAHMS performs best on this task. Read-only access, no support for concurrent access and indexing that is optimised for this task are a few obvious reasons. For the other stores we see a large variation with SWI-Prolog on a second position after Sesame.

bi-BFS Similar to the load test and unlike the DFS test, the figures in Janik and Kochut 2005 on bi-BFS performance show large and hard-to-explain differences. Garbage collection and swapping[14] could explain some of these, but we have no information on these details. We decided against replicating the association search experiment for all stores on the observation that for bi-BFS, only 5% of the time is spent on `rdf/3`. In other words, the bi-BFS tests are actually about comparing the implementation of bi-BFS in 4 different languages (C, C++,

[14] If all measurements are wall time. We confirmed this for BRAHMS by examining the source code but have no access to the source code used to test the other stores.

Association length	9	10	11	12
	Measured			
DFS SWI	1.6	2.1	7	28
DFS BRAHMS	0.1	0.1	0.9	2.2
	Scaled down 3.3×			
DFS Jena	*23*	*32*	*53*	–
DFS Sesame	*0.6*	*0.6*	*3*	*16*
DFS Redland	*5*	*8*	*16*	*62*
Found paths	47	61	61	61

Table 3.6: Search time on small SWETO in seconds. The numbers in the top half of the table are measured on the hardware described on page 43. The numbers on the bottom half of the table are scaled versions of the measurements by Janik and Kochu.

Java and Prolog). Implementation details do matter: our final version of bi-BFS is an order of magnitude faster than the initial version. Such optimisations are easily missed by non-experts in the programming language and/or the RDF store's API.

We executed all bi-BFS experiments on SWI-Prolog and make some qualitative comments. All tests by Janik and Kochu were executed on the 32-bit versions of the RDF stores and many of the tests could not be executed by Jena, Sesame or Redland due to memory exhaustion. As SWI-Prolog memory usage is comparable to BRAHMS, even the 32-bit version can complete all tests. On smaller tests, SWI-Prolog is approximately 10 times slower than BRAHMS. In addition to the argument for DFS, an additional explanation for SWI-Prolog being slower than BRAHMS on bi-BFS can be found in sorting the bi-BFS agendas before joining the paths from both directions, a task on which our implementation spends 10-30% of the time. This step implies sorting integers (representing resources) for BRAHMS and sorting atoms for Prolog. Built-in alphabetical comparison of atoms is considerably slower, especially so because resources only differ after comparing the often equal XML namespace prefix. As the number of paths grows, SWI-Prolog slows down due to excessive garbage collection, particularly so because the current version of SWI-Prolog lacks an incremental garbage collector.

After applying the scale factor of 3.3 found with the DFS experiment, SWI-Prolog is significantly faster than the other systems in the comparison except for BRAHMS. For a fair experiment, the search algorithms must be implemented with involvement of experts in the programming language and the API of the store. Considering the low percentage of the time spent in fetching neighbouring nodes in the RDF graph, this would still largely be a cross-language and programming-skill competition rather than a competition between RDF stores.

3.7 Conclusions

We have outlined different ways to implement an RDF store in Prolog before describing our final implementation as a C library that extends Prolog. The library scales to approximately 20M triples on 32-bit hardware or 300M triples on a 64-bit machine with 64Gb main memory. The 300M triples can be loaded from the internal cache format in approximately 30 minutes, including concurrent rebuilding of the full text search index. The performance of indexed queries is constant with regard to the size of the triple set. The time required for not-indexed queries and re-indexing due to changes in the property hierarchy is proportional to the size of the triple set.

We compared our RDF store on load time, memory footprint and depth-first search for paths that link two resources with four other main memory RDF stores: BRAHMS, Jena, Sesame and Redland. BRAHMS performs best in all these tests, which can be explained because it is designed specially for this job. In particular, BRAHMS is read-only and single-threaded. Compared to Sesame, Jena and Redland, SWI-Prolog's RDF infrastructure has a slower parser (1.1 to 5 times), requires significantly less memory (3 to 8 times) and has competitive search performance (2 times slower to 7 times faster). Profiling indicates that there is significant room for improving the parser. Replacing our parser with Raptor (currently the fastest parser in our test-set) is also an option.

Experience has indicated that the queries required for our annotation and search process (see chapter 9 and 10) can be expressed concisely in the Prolog language. In chapter 4 (Wielemaker 2005) we show how conjunctions of RDF statements can be optimised for optimal performance as part of the implementation of standard RDF query languages (SeRQL and SPARQL) and making these accessible through compliant HTTP APIs. This optimisation is in part based on metrics about the triple set that is maintained by this library.

Acknowledgements

We would like to think Maciej Janik of the BRAHMS project for his help in getting the datasets and clarifying the details of their bi-BFS implementation and Eyal Oren for his help with configuring Sesame and Jena for the evaluation. This work was supported by the ICES-KIS project "Multimedia Information Analysis" funded by the Dutch government and the MultimediaN[15] project funded through the BSIK programme of the Dutch Government.

[15] www.multimedian.nl

Chapter 4

An optimised Semantic Web query language implementation in Prolog

About this chapter This chapter was published at the ICLP-05 (Wielemaker 2005). It continues on research question 1 on knowledge representation, providing an answer to question 1b on efficient matching of RDF graph expressions. From section 4.6 onwards the paper has been updated to reflect the current state of the optimiser, enhance the presentation and create links to the remainder of this thesis in the conclusions.

Abstract

The Semantic Web is a rapidly growing research area aiming at the exchange of *semantic* information over the World Wide Web. The Semantic Web is built on top of RDF, an XML-based *exchange* language representing a triple-based data model. Higher languages such as RDFS and the OWL language family are defined on top of RDF. Making inferences over triple collections is a promising application area for Prolog.

In this article we study query translation and optimisation in the context of the SeRQL RDF query language. Queries are translated to Prolog goals, which are optimised by reordering *literals*. We study the domain specific issues of this general problem. Conjunctions are often large, but the danger of poor performance of the optimiser can be avoided by exploiting the nature of the triple store. We discuss the optimisation algorithms as well as the information required from the low level storage engine.

4.1 Introduction

The Semantic Web (Berners-Lee et al. 2001) initiative provides a common focus for Ontology Engineering and Artificial Intelligence based on a simple uniform triple-based data model. Prolog is an obvious candidate language for managing graphs of triples.

Semantic Web languages, such as RDF (Brickley and Guha 2000), RDFS and OWL, (Dean et al. 2004) define which new triples can be deduced from the current triple set (i.e., are *entailed* by the triples under the language). In this paper we study our implementation of the SeRQL (Broekstra et al. 2002) query language in Prolog. SeRQL provides a declarative search specification for a sub-graph in the deductive closure under a specified Semantic Web language of an RDF triple set. The specification can be augmented with conditions to match literal text, do numerical comparison, etc.

The original implementation of the SeRQL language is provided by Sesame (Broekstra et al. 2002), a Java-based client/server system. Sesame realises entailment reasoning by computing the complete deductive closure under the currently activated Semantic Web language and storing this either in memory or in an external database (*forward reasoning*).

We identified several problems using the Sesame implementation. Sesame stores both the explicitely provided triples and the triples that can be derived from them given de semantics of a specified Semantic Web language (e.g., RDFS) in one database. This implies that changing the language to (for example) OWL-DL requires deleting the derived triples and computing the deductive closure for the new language. Also, where the full deductive closure for RDFS is still fairly small, it explodes for more expressive languages like OWL. Sesame is sensitive to the order in which path expressions are formulated in the query, which is considered undesirable for a declarative query language. Finally we want the reasoning in Prolog because we assume Prolog is a more suitable language for expressing application specific rules.

To overcome the above mentioned problems we realised a server hosting multiple reasoning engines realised as Prolog modules. Queries can be formulated in the SeRQL language and both queries and results are exchanged through the language independent Sesame HTTP-based client/server protocol. We extend the basic storage and query system described in Wielemaker, Schreiber, and Wielinga (2003b) with SeRQL over HTTP and a query optimiser.

Naive translation of a SeRQL query to a Prolog program is straightforward. Being a declarative query language however, authors of SeRQL queries should not have to pay attention to efficient ordering of the path expressions in the query and therefore the query compiler has to derive the optimal order of joins (often called *query planning*). This problem as well as our solution is similar to what is described by Struyf and Blockeel (2003) for Prolog programs generated by an ILP (Muggleton and Raedt 1994) system. We compare our work in detail with Struyf in section 4.11.

In section 4.2 and section 4.3 we describe the already available software components and introduce RDF. Section 4.4 to section 4.9 discuss naive translation of SeRQL to Prolog and optimising the naive translation through reordering of literals.

4.2 Available components and targets

Sesame[1] and its query language SeRQL is one of the leading implementations of Semantic Web RDF storage and query systems (Haase et al. 2004). Sesame consists of two Java-based components. The *server* is a Java *servlet* providing HTTP access to manage the RDF store and run queries on it. The *client* provides a Java API to the HTTP server.

The SWI-Prolog SemWeb package (chapter 3, Wielemaker et al. 2003b) is a library for loading and saving triples using the W3C RDF/XML standard format and making them available for querying through the Prolog predicate `rdf/3`. After several iterations (see section 3.3.3) we realised the memory-based triple-store as a foreign language extension to SWI-Prolog. Using foreign language (C), we optimised the data representation and indexing for RDF triples, dealing with upto about 20 million triples on 32-bit hardware or virtually unlimited on 64-bit hardware. The SWI-Prolog HTTP client/server library (chapter 7, Wielemaker et al. 2008) provides a multi-threaded (chapter 6, Wielemaker 2003a) HTTP server and client library.

By reimplementing the Sesame client/server architecture in Prolog we make our high performance triple store available to the Java world. Possible application scenarios are illustrated in figure 4.1. In our project we needed access from Java applications to the Prolog server. Other people are interested in fetching graphs from huge Sesame hosted triple sets stored in an external database to Prolog for further processing.

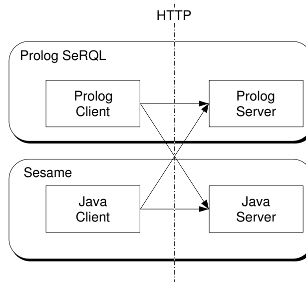


Figure 4.1: With two client/server systems sharing the same HTTP API we have created four scenarios for cooperation: Java or Prolog client connected to Java or Prolog server.

4.3 RDF graphs and SeRQL queries graphs

In this section we briefly introduce RDF graphs and SeRQL queries. The RDF data model is a set of triples of the format <*Subject Predicate Object*>. The model knows about two

[1] `http://www.openrdf.org`

data types:[2] *resources* and *literals*. Resources are *Internationalised Resource Identifiers* (IRI, rfc3987[3]), in our toolkit represented by Prolog atoms. Representing resources using atoms exploits the common representation of atoms in Prolog implementations as a unique handle to a string. This representation avoids duplication of the string and allows for efficient equality testing, the only operation defined on resources. Literals are represented by the term `literal`(*Value*), where *Value* is one of `type`(IRI, *Text*), `lang`(*LangID, Text*) or plain *Text*, and *Text* is the canonical textual value of the literal expressed as an atom (chapter 3, Wielemaker et al. 2003b).

A triple informally states that *Subject* has an attribute named *Predicate* with value *Object*. Both *Subject* and *Predicate* are resources, *Object* is either a resource or a literal. As a resource appearing as *Object* can also appear as *Subject* or *Predicate*, a set of triples forms a *graph*. A simple RDF graph is shown in figure 4.2 (from Beckett and McBride 2004).

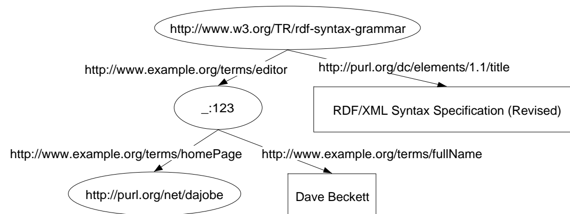


Figure 4.2: A simple RDF graph. Ellipses are resources. Rectangles are literal values. Arrows point from *Subject* to *Object* and are labelled with the *Predicate*.

RDF triples are naturally expressed using the Prolog predicate `rdf`(*Subject, Predicate, Object*). Finding a subgraph with certain properties is now expressed as a Prolog conjunction. The example below finds the home page for a named person in the graph of figure 4.2.

```
homepage_of(Name, HomePage) :-
    rdf(Author, 'http://www.example.org/terms/fullName', literal(Name)),
    rdf(Report, 'http://www.example.org/terms/homePage', HomePage).
```

SeRQL is a language with a syntax inspired by SQL, useful to represent target subgraphs as a set of edges, possibly augmented with conditions. An example is given in figure 4.3.

4.4 Compiling SeRQL queries

The SWI-Prolog SeRQL implementation translates a SeRQL query into a Prolog goal, where edges on the target subgraph are represented as calls to `rdf`(*Subject, Predicate, Object*)

[2]Actually literals can be typed using a subset of the XML Schema primitive type hierarchy, but this refinement is irrelevant to the discussion in this paper.

[3]`http://www.faqs.org/rfcs/rfc3987.html`

and the WHERE clause is represented using natural Prolog conjunction and disjunction of predicates provided in the SeRQL runtime support module. The compiler is realised by a DCG parser, followed by a second pass resolving SeRQL namespace declarations and introducing variables. We illustrate this translation using an example from the SeRQL documentation,[4] shown in figure 4.3.

```
SELECT Painter, FName
FROM {Painter} <rdf:type>        {<cult:Painter>};
               <cult:first_name> {FName}
WHERE FName like "P*"
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

Figure 4.3: Example SeRQL query asking for all resources of type cult:Painter whose name starts with the capital P.

Figure 4.5 shows the naive translation represented as a Prolog clause and modified for better readability using the variable names from the SeRQL query. To solve the query, this clause is executed in the context of an *entailment module* as illustrated in figure 4.4. An entailment module is a Prolog module that provides a pure implementation of the predicate rdf/3 that can generate as well as test all triples that can be derived from the actual triple store using the Semantic Web language the module defines. This implies that the predicate can be called with any instantiation pattern, will bind all arguments and produce all alternatives that follow from the entailment rules on backtracking. If rdf/3 satisfies these criteria, any naive translation of the SeRQL query is a valid Prolog program to solve the query. Primitive conditions from the WHERE clause are mapped to predicates defined in the SeRQL runtime module which is imported into the entailment module. As the translation of the WHERE clause always follows the translation of the path expression, all variables have been instantiated. The call to serql_compare/3 in figure 4.5 is an example of calling a SeRQL runtime predicate that implements the like operator.

Optional path expressions SeRQL path expressions between square brackets ([...]) are *optional*. They bind variables if they can be matched, but they do not change the graph matched by the non-optional part of the expression. Optional path expressions are translated using the SWI-Prolog *soft-cut* control structure represented by *->.[5] Figure 4.6 shows a SeRQL query (top) that finds instances of class *cult:Painter* and enriches the result with the painter's first name(s) if known. The bottom of this figure shows the translation into Prolog,

[4] http://www.openrdf.org/sesame/serql/serql-examples.html

[5] Some Prolog dialects (e.g., SICStus) call this construct if/3.

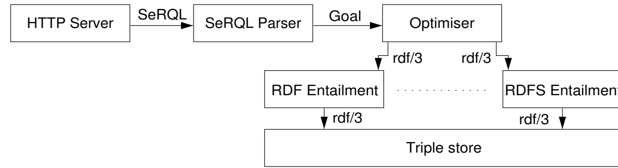


Figure 4.4: Architecture, illustrating the role of *entailment modules*. These modules provide a pure implementation of `rdf/3` for the given Semantic Web language.

```
q(row(Painter, FName)) :-
    rdf(Painter,
        'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
        'http://www.icom.com/schema.rdf#Painter'),
    rdf(Painter,
        'http://www.icom.com/schema.rdf#first_name',
        FName),
    serql_compare(like, FName, 'P*').
```

Figure 4.5: Naive translation of the query of figure 4.3. The row-term in the head describes the bindings of a single result row of the SELECT query.

which leaves *FName* unbound if no first name is known and enumerates all known first names otherwise.

4.5 The ordering problem

Given the purely logical definition of `rdf/3`, conjunctions of RDF goals can be placed in any order without changing the result if we consider two results equivalent if they represent the same *set* of solutions (set-equivalence, Googley and WAH 1989). Literals that result from the WHERE clause are side-effect free boolean tests and can be executed as soon as their arguments have been instantiated.

To study the ordering problem in more detail we will consider the example query in figure 4.7 on WordNet (Miller 1995). The query finds words that can be interpreted in at least two different lexical categories. WordNet is organised in *synsets*, an abstract entity roughly described by the associated *wordForms*. Synsets are RDFS instances of one of the subclasses of LexicalConcept. We are looking for a wordForm belonging to two synsets of a different subtype of LexicalConcept. Figure 4.8 illustrates a query result and gives some relevant metrics on WordNet. The pseudo property `serql:directSubClassOf`

```
SELECT Painter, FName
FROM   {Painter}  <rdf:type>         {<cult:Painter>} ;
                 [<cult:first_name> {FName}]
USING NAMESPACE
        cult = <!http://www.icom.com/schema.rdf#>
```

```
q(row(Painter, FName)) :-
        rdf(Painter,
            'http://www.w3.org/1999/02/22-rdf-syntax-ns#type',
            'http://www.icom.com/schema.rdf#Painter'),
        (   rdf(Painter,
                'http://www.icom.com/schema.rdf#first_name',
                FName)
        *-> true
        ;   true
        ).
```

Figure 4.6: Compilation of a SeRQL optional path expression into the SWI-Prolog soft-cut control structure.

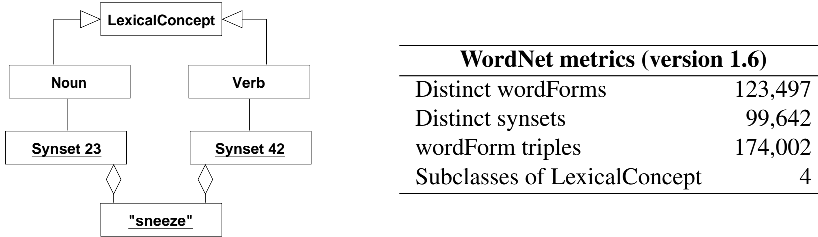
is defined by SeRQL as a non-transitive version of `rdfs:subClassOf`.

```
SELECT DISTINCT L
   FROM {S1} <wns:wordForm> {L},
        {S2} <wns:wordForm> {L},
        {S1} <rdf:type> {C1},
        {S2} <rdf:type> {C2},
        {C1} <serql:directSubClassOf> {<wns:LexicalConcept>},
        {C2} <serql:directSubClassOf> {<wns:LexicalConcept>}
   WHERE not C1 = C2
   USING NAMESPACE
         wns = <!http://www.cogsci.princeton.edu/~wn/schema/>
```

Figure 4.7: Example of a SeRQL query on WordNet

To illustrate the need for optimisation as well as to provide material for further discussion we give two translations of this query. Figure 4.9 shows the direct translation (s1), which requires 3.58 seconds CPU time on an AMD 1600+ processor as well as an alternative ordering (s2) which requires 8,305 CPU seconds to execute, a slowdown of 2,320 times. Note that this translation could be the direct translation of another SeRQL query with the same semantics.

Before we start discussing the alternatives for optimising the execution we explain *why*



WordNet metrics (version 1.6)	
Distinct wordForms	123,497
Distinct synsets	99,642
wordForm triples	174,002
Subclasses of LexicalConcept	4

Figure 4.8: According to WordNet, the word "sneeze" can be interpreted as a *noun* as well as a *verb*. The tabel to the right gives some metrics of WordNet.

```
s1(L) :-
        rdf(S1, wns:wordForm, L),
        rdf(S2, wns:wordForm, L),
        rdf(S1, rdf:type, C1),
        rdf(S2, rdf:type, C2),
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2.

s2(L) :-
        rdf(C1, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        C1 \== C2,
        rdf(S1, rdf:type, C1),
        rdf(S2, rdf:type, C2),
        rdf(S1, wns:wordForm, L),
        rdf(S2, wns:wordForm, L).
```

Figure 4.9: Two translations for our query on WordNet. The first executes in 3.58 seconds, the second in 8,305.

the execution times of these equivalent programs differ. Suppose we have a conjunction of completely independent literals A, B, C, where independent means no variables are shared between the members of the conjunction. If $b()$ denotes the number of solutions for a literal, the total solution space is $b(A) \times b(B) \times b(C)$ and therefore independent of the order. If we take the number of calls+redos (Prolog *logical inferences*) rather than the solution space as a measure the formula becomes

$$\boxed{b(A)} + \boxed{b(A) \times b(B)} + \boxed{b(A) \times b(B) \times b(C)}$$

Logical inferences are a good measure for the expected execution time (Escalante 1993). It suggests to place literals with the smallest number of alternatives first, but as the last

component of the above formula is dominant the difference is not large and certainly cannot explain the difference between the two translations shown in figure 4.9. In fact the second is ordered on the branching factor *without considering dependencies* and as we will see below, dependencies are key to the problem.

Executing an `rdf/3` literal causes all its arguments to be grounded, reducing the number of solutions for `rdf/3` literals sharing the grounded variables. What is really important is how much the set of solutions of a literal is reduced by executing another literal before it. The order of `s1/1` in figure 4.9 executes the most unbound literal first (174,002 solutions), but wins because after the execution of this literal not much further branching is left.

4.6 Estimating the complexity

The first step towards optimising is having an estimate of the complexity of a particular translation. We use the number of logical inferences as an estimate for the execution time, ignoring the (small) differences in time required to execute the different `rdf/3` literals. Crucial to this is to estimate the number of solutions for an `rdf/3` literal. Our estimate is based on information extracted from the low-level database we have realised in the C-language. For this estimate we must consider the *mode* (instantiation pattern) and, if an argument is instantiated, we must distinguish the case where it is bound to a known value by the query and the case where is bound to an unknown value by a preceeding `rdf/3` literal. Below we enumerate the possible modes, where we use '-' for unbound, '+' for bound to a known value, '*' for bound to an unknown value and '@' for bound (known or unknown).

rdf(@,@,@) If all arguments are bound, the call will not create a choicepoint and can succeed or fail. We estimate the the number of solutions at 0.5.

rdf(-,-,-) If no argument is bound, we have a good estimate provided by the number of triples that is maintained by the database. The actual number can be smaller due to duplicates, but this is generally a small proportion.

rdf(-,+,-) Bound to a known predicate we have a good estimate in the number of triples per predicate that is maintained by the database too.

rdf(*,+,-) This is a common case where the subject is bound, but to an unknown value. Here, we are interested in the *average* number of objects associated to a subject with the given predicate. This is the total number of triples for this predicate divided by the total number of *distinct* subject values, which we will call the *subject branching factor* or *sbf*:

$$sbf(P) = \frac{triples(P)}{distinctSubjects(P)}$$

Where the total number of triples is known, the number of distinct subjects must be computed. The *sbf* is determined on the first request. The value is stored and only recomputed if the number of triples has significantly changed since it was determined.

rdf(-,+,*) The *object branching factor* or *obf* definition is completely analogous to the above.

rdf(*,-,-) As the database keeps track of the total number of distinct subjects, this can be computed much like the *sbf* with the following formula: $\frac{triples}{subjects}$

Otherwise If none of the above holds we use information from the hash-based index. With known values for some of the arguments we locate the hash chain that would be used for this query and assume that the length is a good estimate for the total number of solutions. This assumes a well distributed hash function.[6] The length of the hash chains is maintained incrementally by the database primitives.

The above handles '*' as '-' for some (rare) patterns, which can cause a much too high estimate. We currently ignore this issue.

Boolean tests resulting from the WHERE clause cannot cause branching. They can succeed or fail and their branching factor is, as ground `rdf/3` calls, estimated as 0.5 which gives preference to locations early in the conjunction. This number may be wrong but, as we explained in section 4.5, reordering of independent members of the conjunction only has marginal impact on the execution time of the query. If not all arguments to a test are sufficiently instantiated computation of the branching factor fails, causing the conjunction permutation generator to generate a new order.

The total complexity of a conjunction is now expressed as the summed sizes of the search spaces after executing $1, 2, \ldots n$ steps of the conjunction (see formula in section 4.5). The branching factor for each step is deduced using symbolic execution of the conjunction, replacing each variable in a literal with a Skolem instance. Skolem instantiation is performed using SWI-Prolog *attributed variables* (Demoen 2002).

4.7 Optimising the conjunction

With a precise and quick method to compute the complexity of a particular order, the optimisation problem is reduced to a generate-and-test problem. A conjunction of N members can be ordered in $N!$ different ways. As we have seen actual examples of N nearing 40, naive permutation is not an option. However, we do not have to search the entire space as the order of sub-conjunctions that do not share any variables can be established independently, after which they can be ordered on the estimated number of solutions.

[6] We use MurmurHash from `http://murmurhash.googlepages.com/`

Initially, for most conjunctions in a query all literals are related through shared variables.[7] As execution of the conjunction progresses, more and more variables are bound. This may cause the remainder of the conjunction to break into multiple independent sub-conjunctions. For example, the query below is initially fully related over *Painter*. After executing one of the literals, *Painter* is bound and the two remaining literals become independent. Independent sub-conjunctions can be optimised separately.

```
q(row(Painter, FirstName, Painting)) :-
        rdf(Painter, rdf:type, cult:'Painter'),
        rdf(Painter, cult:first_name, Name),
        rdf(Painter, cult:creator_of, Painting).
```

The algorithm in figure 4.10 generates on backtracking alternative orderings with an estimate for their time complexity and number of solutions. The predicate `estimate_complexity/3` estimates the number of solutions as described above, while the time estimate is 1 (i.e., our unit of time is the time an `rdf/3` call needs to generate or test a triple). The predicate `combine/3` concatenates the ordered sub-conjunctions and computes the combined complexity.

Using the generator above, we can enumerate all permutations and select the fastest one. The returned order is guaranteed to be optimal if the complexity estimate is perfect. In other words, the maximum performance difference between the optimal order and the computed order is the error of our estimation function. We have seen in section 4.6 that the error margin varies, depending on the types of `rdf/3` calls. We do not have access to a sufficiently large set of real-world SeRQL queries to assess the accuracy in any detail.

4.8 Optional path expressions and control structures

As explained in section 4.4, SeRQL optional path expressions are compiled into (Goal `*->` true ; true), where *Goal* is the result of compiling the path expression. We preserve control structures and apply our ordering algorithm to conjunctions that are embedded in the control structures.

Optional path expressions do not change the result set of the obligatory part of the query. It can only produce more variable bindings. Therefore we can simplify the optimisation process of a conjunction by first splitting it into its obligatory and optional part and then optimise the obligatory part followed by the optional part as shown below. The predicate `combine/2` is the same as from figure 4.10. We must do the Skolem binding of the obligatory part because it affects the result when ordering conjunctions in the optional part of the expression.

[7] This is not necessarily the case, but a query that consists of independent sub-queries is answered by the Cartesian product of the answers for each of the sub-queries (see section 4.9). Processing the results of each independent sub-query is more efficient and generally easier.

```
%%      order(+Conjunction, -Result) is nondet.
%
%       @param Result   o(Time, Solutions, Order)

order([One], o(T,N,[One])) :- !,
        estimate_complexity(One, T, N),
order(Conj, Result) :-
        make_subgraphs(Conj, SubConjs),
        (   SubConjs = [_,_|_]
        ->  maplist(order, SubConjs, OSubs),
            sort(OSubs, OList),
            combine(OList, Result)
        ;   select(First, Conj, Rest),
            skolem_bind(First),
            order(Rest, o(T0,N0,O0)),
            estimate_complexity(First, T1, N1),
            T is T1+N1*T0,
            N is N0*N1,
            Result = o(T,N,[First|O0])
        ).
```

Figure 4.10: Order a conjunction. Alternative orderings are generated due to the non-deterministic `select/3`

```
order(Goal, Result) :-
        split_optional(Goal, Obligatory0, Optional0),
        order(Obligatory0, Obligatory),
        skolem_bind(Obligatory),
        order(Optional0, Optional),
        combine([Optional, Optional], Result.
```

4.9 Solving independent path expressions

As we have seen in section 4.7, the number of distinctive permutations is much smaller than the number of possible permutations of a goal due to the fact that after executing a few literals the remainder of the query breaks down into independent subgraphs. Independent subgraphs can be solved independently and the total solution is the Cartesian product of all partial solutions. This approach has several advantages:

- The complexity of solving two independent goals A and B separately is $b(A) + b(B)$ rather than $b(A) + b(A) \times b(B)$.

- The subgoals can be solved in parallel.
- If any of the independent goals has no solutions we can abort the whole query and report it has no solutions.
- The solution can be expressed much more concisely as the Cartesian product of partial results. This feature can be used to reduce the communication overhead.

This optimisation can be achieved by replacing the calls to `sort/2` and `combine/2` in the algorithm of figure 4.10. The replacement performs two task: identify which (independent) projection variables of the query appear in each of the sub-conjunctions and create a call to the runtime routine `serql_carthesian`(+*GoalsAndVars*, -*Solution*) that is responsible for planning and executing the independent goals.

4.10 Evaluation

We evaluated three aspects of the system. The amount of source code illustrates the power of Prolog for this type of task. Next, we study the optimisation of the WordNet query given in figure 4.7 and finally we examine 3 queries originating from a real-world project that triggered this research.

Source code metrics The total code size of the server is approximately 6,700 lines. Major categories are show in table 4.1. We think it is not meaningful to compare this to the 86,000 lines of Java code spread over 439 files that make up Sesame. Although both systems share considerable functionality, they differ too much in functionality and how much is reused from the respective system libraries to make a detailed comparison feasible. As often seen, most of the code is related to I/O (57%), while the core (query compiler and optimiser) is responsible for only 26% of the code.

Category	lines
HTTP server actions	2,521
Entailment modules (3)	281
Result I/O (HTML, RDF/XML, Turtle)	1,307
SeRQL runtime library	192
SeRQL parser and naive compiler	874
Optimiser	878
Miscellaneous	647
Total	6,700

Table 4.1: Size of the various components, counted in lines. RDF/XML I/O is only a wrapper around the SWI-Prolog RDF library.

Evaluating the optimiser We have evaluated our optimiser on two domains: the already mentioned WordNet and an RDF set with accompanying queries from an existing application. Measurements have been executed on a dual AMD 2600+ machine running SuSE Linux and SWI-Prolog 5.5.15.

First we study the example of figure 4.9 on page 56. The code for `s1/1` was handcrafted by us and can be considered an educated guess for best performance. The described optimiser converts both `s1/1` and `s2/1` into `o1/1` as shown in figure 4.11. Table 4.2 confirms that the optimiser produced a better version than our educated guess and that the time to optimise this query is negligible.

```
o1(L) :-
        rdf(S1, rdf:type, C1),
        rdf(S1, wns:wordForm, L),
        rdf(C2, rdfs:subClassOf, wns:'LexicalConcept'),
        rdf(S2, rdf:type, C2),
        rdf(S2, wns:wordForm, L),
        C1 \== C2,
        rdf(C1, rdfs:subClassOf, wns:LexicalConcept))
```

Figure 4.11: Machine optimised WordNet query

Translation	opt. time	exec time	total time
`s1/1`	-	3.58	3.58
`o1/1`	0.09	2.10	2.19

Table 4.2: Timing of human (s1/1) and machine (o1/1) optimised translations of the query of figure 4.7. *Opt. time* is the time used optimise the query and *exec time* is the time used to execute the query.

The second test-set consisted of three queries on a database of 97,431 triples coming from a real project carried out at Isoco.[8] These queries were selected because Sesame (Broekstra et al. 2002) could not answer them (2 out of 3) or performed poorly. Later examination revealed these queries consisted of multiple largely independent sub-queries, turning the result in a huge Cartesian product. Splitting them into multiple queries turned them into manageable queries for Sesame. Exploiting the analysis of independent path expressions described in section 4.9, our server does not need this rewrite. The results are shown in table 4.3. *Edges* is the number of graph edges that appear in the query. The actual time spent in the optimiser (3th column) is again negligible. The next three columns

[8] www.isoco.com

present the initial complexity estimate, the estimate for the optimised query and the relative optimisation based on these estimates. Because the initial queries do not terminate in a reasonable time we can not verify the estimated speedup ratio. The *time* column gives the total query processing time for both our implementation and Sesame. For Sesame we only have results for the second query as the others did not terminate overnight. The last column gives the total number of rows in the output table. The solutions of the first and last queries are Cartesian products (see section 4.9).

Id	Edges	time optimise	complexity initial	final	speedup	time Us	Sesame	solutions
1	38	0.01	1.4e16	1.4e10	1.0e6	2.48	-	79,778,496
2	30	0.01	2.0e13	1.3e05	1.7e8	0.51	132	3,826
3	29	0.01	1.4e15	5.1e07	2.7e7	11.7	-	266,251,076

Table 4.3: Results on optimising complex queries. Time fields are in seconds.

4.11 Related Work

Using logic for Semantic Web processing has been explored by various research groups. See for example (Patel and Gupta 2003) which exploits *Denotational Semantics* to provide a structured mapping from language to semantics. Most of these approaches concentrate on correctness, while we concentrate on engineering issues and performance.

Much work has been done on optimising Prolog queries as well as database joins by reordering. We specifically refer to the work of Struyf and Blockeel (Struyf and Blockeel 2003) because it is recent and both the problem and solution are closely related. They describe the generation of programs through ILP (Muggleton and Raedt 1994). The ILP system itself does not consider ordering for optimal execution performance, which is similar to compiling declarative SeRQL statements not producing optimal programs. In ILP, the generated program must be used to test a large number of positive and negative examples. Optimising the program before running is often worthwhile.

The described ILP problem differs in some places. First of all, for ILP one only has to prove that a certain program, given a certain input, succeeds or fails, i.e., goals are ground. This implies they can use the cut to separate independent parts of the conjunction (section 4.2 of Struyf and Blockeel (2003)). As we have non-ground goals and are interested in all distinct results we cannot use cuts but instead use the Cartesian product approach described in section 4.9. Second, Struyf and Blockeel claim complexity of generate-and-test (order $N!$) is not a problem with the observed conjunctions with a maximum length of 6. We have seen conjunctions with 40 literals. We introduce breaking the conjunctions dynamically in independent parts (section 4.7) can deal with this issue. Finally, the uniform nature of our data gave us the opportunity to build the required estimates for non-determinism into the

low-level data structures and maintain them at low cost (section 4.6).

4.12 Discussion

Current Semantics Web query languages deal with graph expressions and entailment under some Semantics Web language, which is typically implemented by computing the deductive closure under this language. As we demonstrated, graph expressions and conditions are easily translated into pure Prolog programs that can be optimised using reordering. For languages that are not expressive such as RDFS, entailment can be realised both using forward chaining and backward chaining. Figure 4.4 describes how we realise multiple reasoning schemes using Prolog modules and backward chaining. As long as the entailment rules are simple, the optimiser described here can be adapted to deal with the time and solution count estimates related to executing these rules.

As the Semantic Web evolves with more powerful formal languages such as OWL and SWRL,[9] it becomes unlikely we can compile these easily into efficient Prolog programs and provide estimates for their complexity. TRIPLE (Sintek and Decker 2002) is an example of an F-logic-based RDF query language realised in XSB Prolog (Freire et al. 1997). We believe extensions to Prolog that facilitate more declarative behaviour will prove necessary to deal with the Semantic Web. Both XSB's tabling and constraint logic programming, notably CHR (Frühwirth 1998; Schrijvers and Demoen 2004) are promising extensions.

4.13 Conclusions

In chapter 3 (Wielemaker et al. 2003b) we have demonstrated the performance and scalability of an RDF storage module for use with Prolog. In this paper we have demonstrated the feasibility of realising an efficient implementation of the declarative SeRQL RDF query language in Prolog.

The algorithm for optimising the matching process of SeRQL queries reaches optimal results if the complexity estimate is perfect. The worse case complexity of ordering a conjunction is poor, but for tested queries the optimisation time is shorter than the time needed to execute the optimised query. For trivial queries this is not the case, but here the response time is dictated by the HTTP protocol overhead and parsing the SeRQL query.

The fact that Prolog is a reflexive language (a language where programs and goals can be expressed as data) together with the observation that a graph pattern expressed as a list of triples with variables is equivalent to a Prolog conjunction of `rdf/3` statements provides a natural API to RDF graph expressions and the optimiser: translating a complex Prolog goal into an optimised one. Prolog's non-determinism greatly simplifies exploration of the complex search space in the generate-and-test cycle for finding the optimal program. The optimisation relies (section 4.6) on metrics maintained by the RDF store described in chapter 3 (Wielemaker et al. 2003b). Many of these metrics only apply to RDF, which justifies

[9] http://www.daml.org/2003/11/swrl

our decision to implement rdf/3 as a foreign language extension in favour of adding optimisations to Prolog that make rdf/3 feasible as a normal Prolog dynamic predicate.

The HTTP frontend we developed to create a Sesame compliant HTTP API has been the basis for the development of ClioPatria chapter 10 (Wielemaker et al. 2008). The server has been extended with support for the W3C standard SPARQL language and HTTP API and is now an integral component of ClioPatria (figure 10.5). Reasoning inside ClioPatria uses direct Prolog queries on the RDF store and the optimisation library described in this paper is used for dynamic optimisation of goals with embedded rdf/3 statements where the ordering is not obvious to the programmer.

Acknowledgements

We would like to thank Oscar Corcho for providing real-life data and queries. This research has been carried out as part of the HOPS project,[10] IST-2002-507967. Jeen Broekstra provided useful explanations on SeRQL.

[10] http://www.hops-fp6.org

Chapter 5

An architecture for making object-oriented systems available from Prolog

About this chapter This chapter has been published at the WLPE-02 (Wielemaker and Anjewierden 2002). It examines research question 3: "How to support graphical applications in Prolog?" by describing how XPCE, an object oriented graphics library defined in C can be connected to Prolog. XPCE is used by Triple20 (chapter 2, Wielemaker et al. 2005), the MIA tools (chapter 9, Schreiber et al. 2001; Wielemaker et al. 2003a) as well as most of the development tools of SWI-Prolog (Wielemaker 2003b).

Abstract It is next to impossible to develop real-life applications in just pure Prolog. With XPCE (Wielemaker and Anjewierden 1992) we realised a mechanism for integrating Prolog with an external object-oriented system that turns this OO system into a natural extension to Prolog. We describe the design and how it can be applied to other external OO systems.

5.1 Introduction

A wealth of functionality is available in object-oriented systems and libraries. This paper addresses the issue of how such libraries can be made available from Prolog, in particular libraries for creating user interfaces.

Almost any modern Prolog system can call routines in C and be called from C. Also, most systems provide ready-to-use libraries to handle network communication. These primitives are used to build bridges between Prolog and external libraries for (graphical) user-interfacing (GUIs), connecting to databases, embedding in (web-)servers, etc. Some, especially most GUI systems, are object-oriented (OO). The rest of this paper concentrates on GUIs, though the arguments apply to other systems too.

GUIs consist of a large set of entities such as windows and controls that define a large number of operations. Many of the operations involve destructive changes to the involved

entities. The behaviour of GUI components normally involves handling spontaneous input in the form of *events*. OO techniques are well suited to handle this complexity. A concrete GUI is generally realised by sub-classing base classes from the GUI system. In —for example— Java and C++, the same language is used for both the GUI and the application. This is achieved by either defining the GUI base classes in this language or by encapsulating foreign GUI classes in classes of the target language. This situation is ideal for application development because the user can develop and debug both the GUI and application in one language.

For Prolog, the situation is more complicated. Diversity of Prolog implementations and target platforms, combined with a relatively small market share of the Prolog language make it hard to realise the ideal situation sketched above. In addition, Prolog is not the most suitable language for implementing fast and space-efficient low-level operations required in a GUI.

The main issue addressed in this paper is how Prolog programmers can tap on the functionality provided by object-oriented libraries without having to know the details of such libraries. Work in this direction started in 1985 and progresses to today. Over these years our understanding of making Prolog an allround programming environment has matured from a basic interface between Prolog and object-oriented systems (section 5.3), through introducing object-oriented programming techniques in Prolog (section 5.4), and finally to make it possible to handle Prolog data transparently (section 5.5). These ideas have been implemented in XPCE. Throughout this paper we will use examples based on XPCE and discuss how these principles can be applied to other OO systems.

5.2 Approaches

We see several solutions for making OO systems available from Prolog. One is a rigid separation of the GUI, developed in an external GUI development environment, from the application. A narrow bridge links the external system to Prolog. Various styles of 'bridges' are used, some based on TCP/IP communication and others on local in-process communication. ECLiPSe (Shen et al. 2002) defines a generic interface for exchanging messages between ECLiPSe and external languages that exploits both in-process and remote communication. Amzi! (Merritt 1995) defines a C++ class derived from a Prolog-vendor defined C++/Prolog interface class that encapsulates the Prolog application in a set of C++ methods, after which the GUI can be written as a C++ application.

Using a narrow bridge between data processing in Prolog and a GUI in some other language provides modularity and full reuse of GUI development tools. Stream-based communication is limited by communication protocol bandwidth and latency. Whether or not using streams, the final application consists of two programs, one in Prolog and one in an external language between which a proper interface needs to be defined. For each new element in the application the programmer needs to extend the Prolog program as well as the GUI program and maintain the interface consistency.

For applications that require a wide interface between the application and GUI code we

would like to be able to write the application and GUI both in Prolog. Here we see two approaches. One is to write a Prolog layer around the (possibly extended) API of an existing GUI system (Kim 1993; SICS 1998). This option is unattractive as GUI systems contain a large number of primitives and data types, which makes development and maintenance of the interface costly. An alternative is to go for a minimal interface based on the OO message passing (control) principles. For this, we consider OO systems where methods can be called from the C-language[1] based on their *name*. The ability to access methods by name is available in many (OO) GUI toolkits as part of their *reflexion* capabilities. Fully compiled languages such as traditional C++ lack this facility, but the availability of many compiler extensions and libraries to add reflexion to C++ indicate that the limited form of reflexion that we demand is widely available. In the remainder of this article we concentrate on OO systems that can call methods by name.

5.3 Basic Prolog to OO System Interface

The simplest view on an OO system is that it provides a set of methods (functions) that can be applied to objects. This uniformity of representation (objects) and functionality (methods) makes it possible to define a small interface between Prolog and an OO system. All that is required to communicate is to represent an object by a unique identifier in Prolog, translate Prolog data to types (objects) of the OO system, invoke a method and translate the result back to Prolog. This model is illustrated in figure 5.1. We first describe how objects are manipulated from Prolog ('Activate'), then how we can represent Prolog as an object ('Call back') and finally we discuss portability and limitations of this model.

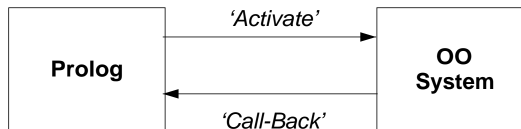


Figure 5.1: A simple view on Prolog to OO interface

Object Manipulation from Prolog We added four predicates to Prolog to manipulate objects in the OO system:

new(*?Reference, +Class(...Arg...)*)

Create an object as an instance of *Class* using the given arguments to initialise the object. Make the resulting instance known as *Reference* from Prolog. If *Arg* is atomic, convert it to a natural counterpart in XPCE (integer, float, atom). If it is compound,

[1] Or another language Prolog can communicate with. C is the de-facto interface language for many high-level languages.

create an instance using the functor as the class name and the arguments as initialising arguments. Unbound variables have no sensible counterpart in XPCE and therefore raise an exception. The example below creates a box (graphical rectangle) with specified width and height.

```
?- new(X, box(100,100)).

X = @459337
```

send(+*Reference*, +*Method(...Arg...)***)**
Given a reference to an object, invoke the named method with the given arguments. The arguments are translated as with `new/2`, supporting the functional notation shown in the code below. XPCE uses `send/2` for methods returning either no value or a boolean success/failure indication. The example creates a picture (graphics window) and displays a box at a given location.

```
?- new(P, picture),
   send(P, display(box(100,50), point(20,20))).
```

get(+*Reference*, +*Method(...Arg...)*, -*Result***)**
Used for methods that return their result as an object or primitive data type. If the returned value is primitive, it is converted to a Prolog integer, float or atom. In all other cases an object reference is returned. The following code gets the left-edge of the visible part of a graphics window (*picture*). *Visible* is a instance of class *area* that describes the visual part and defines get-methods *x,y,w,h* to obtain the dimensions of the area object.

```
?- new(P, picture),
   get(P, visible, Visible),
   get(Visible, x, X).

P = @1072825
Visible = @957733
X = 0
```

free(+*Reference***)**
Destroy the referenced object.

Minimalists may claim that three of these four primitives are redundant. Generating an instance of a class is an operation of the class and can thus be performed by a method of the

class if classes can be manipulated as objects. In XPCE, `new/2` can be defined as `get`(*Class, instance(...Arg...), Reference*). Destroying an object is an operation on the object itself: `send`(*Reference, free*). Most object systems do not distinguish 'send' and 'get': methods that do not need to return a value return either boolean truth or the object to which the method was sent. Send and get are distinguished in the design of XPCE as well as in the interface because we perceive the distinction between *telling* an object to perform an action without interest for the result (e.g., *move* to a location) and *asking* an object to compute or create something, improves readability.

Object systems define rules that describe the lifetime of objects. These rules can be based on scoping, membership of a 'container' or garbage collection. This 'life-time' of the object is totally unrelated to the life-time of the Prolog reference and the user must be aware of the object life-time rules of the object system to avoid using references to deleted objects. XPCE defines scoping rules and provides a reference-based garbage collector. This takes care of the objects, but the programmer must be careful not to use Prolog object-references to access objects that have been deleted by XPCE. The SWI-Prolog Java interface JPL[2] represents Java objects as unique Prolog atoms and exploits a hook into the SWI-Prolog atom garbage collector to inform the interface about Java objects that are no longer referenced by Prolog. The XPCE route is lightweight but dangerous. The JPL route is safe, but atom garbage collection is a costly operation and Java objects often live much longer than needed.

Prolog as an Object The four predicates above suffice to invoke behaviour in the OO system from Prolog. Notable OO systems for GUI programming generally define *events* such as clicking, typing or resizing a window that require Prolog to take action. This can be solved by defining a class in the OO system that encapsulates Prolog and define a method *call* that takes a predicate name and a list of OO arguments as input. These arguments are translated to Prolog in the same way as the return value of `get/3` and the method is executed by calling the predicate.

In XPCE, class *prolog* has a single instance with a public reference (@prolog). GUI classes that generate events, such as a push button, can be associated with a *message* object. A message is a dormant method invocation that is activated on an event. For example, the creation arguments for a push button are the label and a message object that specifies an action when the user clicks the button. Together with the introduction of @prolog, we can now create a push button that writes `Hello World` in the Prolog console when clicked with the code below.

```
?- new(B, button(hello,
                 message(@prolog, call,
                         writeln, 'Hello World'))).
```

[2] `http://www.swi-prolog.org/packages/jpl/`

Portability The above has been defined and implemented around 1985 for the first XPCE/Prolog system. It can be realised for any OO system that provides runtime invocation of methods by name and the ability to query and construct primitive data types.

Limitations The basic OO interface is simple to implement and use. Unfortunately it also has some drawbacks as it does not take advantage of some fundamental aspects of object-oriented programming: specialisation through sub-classing and the ability to create abstractions in new classes. These drawbacks become apparent in the context of creating interactive graphical applications.

Normally, application GUI-objects are created by sub-classing a base class of the toolkit and refining methods such as *OnDraw* and *OnClick* to perform the appropriate application actions. Using the interface described above, this means we need to program the OO system, which implies The programmer has to work in Prolog and the OO language at the same time. In section 5.2 we discussed scenarios that required programming in two languages and resulted in two applications that communicated with a narrow bridge, providing good modularity. If we create a system where classes can be created transparently from Prolog we can achieve a much more productive development environment. This is especially true if the OO system does not allow for (re-)compilation in a running application, but Prolog can provide for this functionality. Unfortunately this environment no longer *forces* a clean modular separation between GUI and application, but it still *allows* for it.

5.4 Extending Object-Oriented Systems from Prolog

If we can extend the OO system from Prolog with new classes and methods that are executed in Prolog we have realised two major improvements. We gain access to functionality of the OO system that can only be realised by refining methods and, with everything running in Prolog, we can develop the entire application in Prolog and profit from the good interactive development facilities provided by Prolog. This can be realised in a portable manner using the following steps, as illustrated in figure 5.2:

- Defining a Prolog syntax for classes and methods, where the executable part of the method is expressed in Prolog.
- Create the OO system's classes from this specification and define the OO system's methods as wrappers that call the Prolog implementation.

A concrete example is given in figure 5.3, which creates a derived class *my_box* from the base class *box* (a rectangular graphical). The derived class redefines the method *event*, which is called from the window in which the box appears if the mouse-pointer hovers over the box. The method receives a single argument, an instance of class *event* that holds details of the GUI event such as type, coordinates and time. In this example we ensure the box is filled solid red if the mouse is inside the box and transparent (filled with nothing: @nil) otherwise,

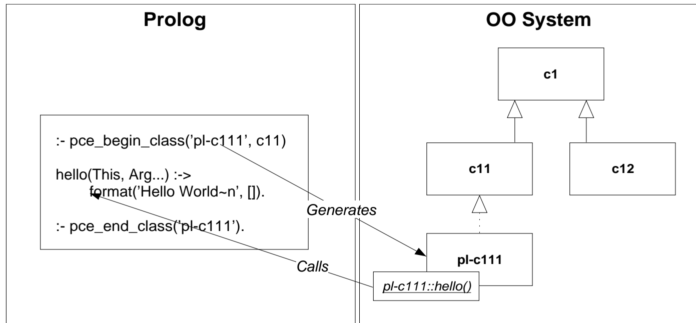


Figure 5.2: Creating sub-classes from Prolog

by redefining the method that specifies what happens if the mouse enters or leaves the box. In all other cases, the default behaviour of the super class is activated using `send_super/2`.

```
:- pce_begin_class(my_box, box).

event(Box, Event:event) :->
        (   send(Event, is_a(area_enter))
        ->  send(Box, fill_pattern(colour(red)))
        ;   send(Event, is_a(area_exit))
        ->  send(Box, fill_pattern(@nil))
        ;   send_super(Box, event(Event))
        ).

:- pce_end_class(my_box).
```

Figure 5.3: Defining a class from Prolog

XPCE is a 'soft typed' language. Method arguments may have type specifiers (`:event`) and if they do the system performs runtime checks on these. The method above requires the first argument to be an instance of the class *event*.

Implementation In XPCE, classes and methods are primary objects that can be manipulated using the API described in section 5.3. A method object consists of a name, argument type definition and a handle to the implementation. The implementation is either built into XPCE itself as a C-function, or it is provided by the XPCE/Prolog as an object (X) created by the interface. If an implementation defined by the interface is called, XPCE calls the interface

with the receiving object, type-checked arguments and provided implementation object X. The meaning of the object X is defined by the interface; XPCE just distinguishes built-in methods and methods defined by the interface.

We now describe how classes defined in Prolog are realised in XPCE. We do this in two steps. First, we describe a naive translation that creates XPCE classes and methods while the Prolog file is being compiled. Next, we describe how the actual implementation differs from this naive translation by introducing just-in-time creation of XPCE classes and methods.

The code from figure 5.3 is translated using Prolog's `term_expansion/2` based macro-facility. The `begin_class`(*my_box, box*) is expanded into a `new/2` call that creates class *my_class* as a subclass of *box*. The method definition is translated into a clause (figure 5.4) and API calls that create a method object with appropriate name, types and implementation object and associate this with the new class.

```
pce_principal:send_implementation('my_box->event', event(A), B) :-
    user:
    (   (   send(A, is_a(area_enter))
        ->  send(B, fill_pattern(colour(red)))
        ;   send(A, is_a(area_exit))
        ->  send(B, fill_pattern(@nil))
        ;   send_class(B, box, event(A))
        )
    ).
```

Figure 5.4: Clause that provides implementation for a method

Each send-method is translated into a single clause for the multifile predicate pce_principal:send_implementation/3, an example of which is shown in figure 5.4. The atom `'my_box->event'` is the unique handle to the implementation that is stored with the implementation object (X above) for the method. Prolog's first argument indexing ensures fast access, even if there are many methods.

Just-in-time creation of methods and classes Creating classes and methods while compiling the XPCE classes does not cooperate easily with Prolog's support for compiling files into object files or generating saved states. In addition, startup time is harmed by materialising all classes and methods immediately at load-time. Therefore, the actual implementation uses term expansion to compile the class declarations into the clause for pce_principal:send_implementation/3 as described and creates a number of Prolog facts that describe the classes and methods. XPCE provides a hook that is called if an undefined class or method is addressed. The Prolog implementation of this hook materialises classes and methods just-in-time. As the result of compiling an XPCE class is a set of Prolog clauses, all normal compilation and saved state infrastructure can be used without change.

Portability Our implementation is based on XPCE's ability to refine class *method*, such that the implementation can be handled by a new entity (the Prolog interface) based on a handle provided by this interface (the atom `'my_box->event'` in the example). Not all object systems have this ability, but we can achieve the same result with virtually any OO system by defining a small wrapper-method in the OO language that calls the Prolog interface. Ideally, we are able to create this method on demand through the OO system's interface. In the least attractive scenario we generate a source-file for all required wrapper classes and methods and compile the result using the target OO system's compiler. Even in this setting, we can still debug and reload the method *implementation* using the normal Prolog interactive debugging cycle, but we can only extend or modify the class and method *signatures* by running the class-compiler and restarting the application.

Experience The first version of XPCE where classes could be created from Prolog was developed around 1992. Over the years we have improved performance and the ability to generate compact and fast starting saved states. Initially the user community was reluctant, possibly due to lack of clear documentation and examples. The system proved valuable for us, making more high-level functionality available to the XPCE user using the uniform class-based framework.

We improved the usability of creating classes from Prolog by making the Prolog development environment aware of classes and methods (section 5.7). We united listing, setting spy-points, locating sources and cross-referencing. Support requests indicate that nowadays a large share of experienced XPCE users define classes. Problems are rarely related to the class definition system itself. User problems concentrate on how to find and combine classes and methods of the base system that fulfil their requirements. XPCE shares this problem with other large GUI libraries.

With the acceptance of XPCE/Prolog classes, a new style of application development emerged. In this style, classes became the dominant structuring factor for interactive applications. Persistent storage and destructive assignment make XPCE data representation a viable alternative to Prolog's poor support of these features based on assert/retract.

5.5 Transparent exchange of Prolog data

In the above scenario, control always passes through XPCE when calling a method, regardless of whether the method is built-in or implemented in Prolog. As long as the XPCE/Prolog classes only extend XPCE it is natural that data passed to the method is restricted to XPCE data. If XPCE classes are used for implementing more application-oriented code we need a way to process native Prolog data in XPCE. We distinguish two cases: *passing* Prolog terms as arguments and *storing* Prolog terms in XPCE instance variables. Below is a description of both problems, each of which is followed by a solution section, after which we present and an example in section 5.5.3.

- *Passing Prolog data to a method*

Seen from Prolog, XPCE consists of three predicates that pass arguments: `new/2`, `send/2` and `get/3`. New/2 calls the *constructor* method, called `initialise` in XPCE while `send/2` and `get/3` specify the method called. We consider the case where the method that is called is implemented in Prolog using the mechanism described in section 5.4 and the caller wishes to pass an arbitrary Prolog term to the method implementation. In this scenario the life-time of the term is limited to the execution of the method and therefore the term can be passed as a reference to the Prolog stacks.

- *Storing Prolog data in an* XPCE *instance variable*
  If methods can pass Prolog data, it also becomes useful to *store* Prolog data inside objects as the value of an instance variable. In this case the term needs to be stored on the Prolog permanent heap using a mechanism similar to `assert/1` or `recorda/3` and the instance variable needs to be filled with a handle to retrieve the stored Prolog term.

5.5.1 Passing Prolog data to a method

When a Prolog term is passed to a method, it is passed as a term handle as defined by the Prolog-to-C interface. The passed term needs not be ground and can be (further) instantiated by the implementation of the method called. Figure 5.5 provides an example, where a parse-tree is represented by a Prolog term of the form `node`(*Content, Children*) and is passed as a whole to initialise an instance of the Prolog-defined class *parse_tree*.

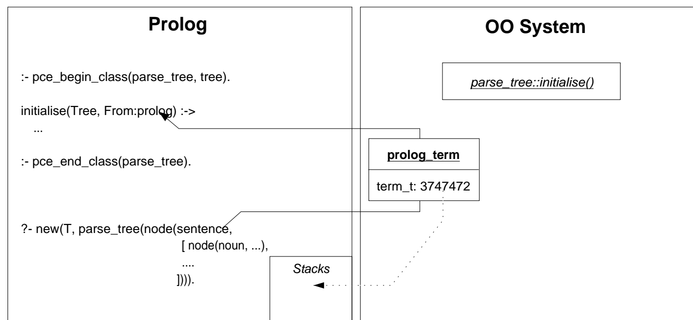


Figure 5.5: Passing Prolog terms across the OO system to other Prolog methods. The term is passed 'by reference'. It can contain unbound variables that are instantiated by the receiving method.

For the technical realisation we introduced class *host_data* providing XPCE with an opaque handle to data of the *host*, the term XPCE uses for the language(s) that are con-

nected to it. The XPCE/Prolog interface sub-classes this type to *prolog_term* (figure 5.5). Prolog data is opaque from XPCE's perspective.

When preparing an XPCE method invocation, we check whether a method argument is of type `prolog`, a type that refers to *prolog_term* as well as primitive data. If the argument is not primitive (integer, float or atom), the interface creates an instance of *prolog_term* and stores the `term_t` term reference in this object. Whenever an instance of *prolog_term* is passed from XPCE to Prolog, the interface extracts the term reference from the instance and passes it to Prolog.

5.5.2 Storing Prolog data in an XPCE instance variable

In the case where Prolog data is to be stored in instance variables, we cannot use the above described *prolog_term* with a reference to a Prolog term on the stack because the lifetime of the term needs to be linked to the object and existence of a term on the stack is not guaranteed after execution of a method completes. Instead, we need to use Prolog dynamic predicates or the recorded database and store a handle to the Prolog data in the XPCE instance variable. For the storage mechanism we opt for functions PL_record(), PL_recorded() and PL_erase() defined in the SWI-Prolog foreign interface. These functions copy terms between the stacks and permanent heap. A term on the permanent heap is represented by a handle of type `record_t`.

At the moment a method is called, the interface cannot know whether or not the term will be used to fill an instance variable and therefore a Prolog term is initially always wrapped into an instance of the XPCE class *prolog_term*. We identified two alternatives to create a persistent version of this term that is transparent to the user:

- In addtion *host_data*, Introduce another subclass of *host_data* that uses the PL_record()/PL_erase() mechanism to store the Prolog term. This can be made transparent to the Prolog user by rewriting instance variable declarations of type `prolog` to use this new class as type and define an automatic conversion between *prolog_term* and this new class.

- Introduce two internal representations for *prolog_term*: (1) the original based on `term_t` and (2) a new one based on `record_t`. Initially the object represents the Prolog term using the `term_t` reference. It is converted into the `record_t` representation by `new/2`, `send/2` or `get/3` if the *prolog_term* instance is *referenced* after completion of the method. We can detect that an object is referenced because XPCE has a reference-count-based garbage collector. Details are in the 3 steps described below.

    1. For each prolog-typed argument, create a *prolog_term* instance and keep an array of created instances.

    2. Run the method implementation.

3. Check the reference count for each created *prolog_term* instance. If zero, the instance can be destroyed. If > 0, it is referenced from some object and we transform the instance to its recorded database form. Figure 5.6 shows the situation after setting an instance variable.

We opted for the second alternative, mainly because it is easier to implement given the design of XPCE and its Prolog interface.

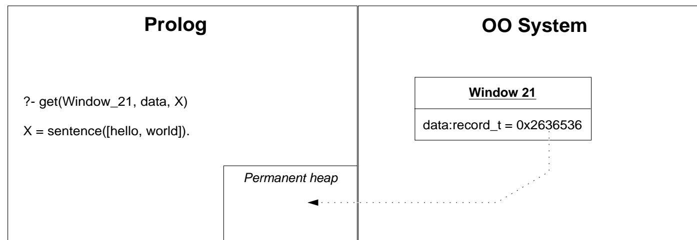


Figure 5.6: Storing Prolog data in external objects. The object contains a reference to a copy of the term maintained in the Prolog permanent heap.

5.5.3 An example: create a graphical from a Prolog tree

Figure 5.7 defines a class that creates a graphical tree where each node has a permanent payload of Prolog data associated. The hierarchy is created directly from a complex Prolog data structure. After creating a hierarchy this way, the Prolog tree as a whole is always passed *by reference* as illustrated in figure 5.5. Each node contains a permanent copy of the associated pay load as illustrated in figure 5.6.

5.5.4 Non-deterministic methods

If a Prolog-defined method is called from Prolog, it is desirable to be able to preserve possible non-determinism of the method. In the approach above, where activating a Prolog-defined method is always initiated from inside XPCE, this is not feasible. We can realise this desired feature by splitting method-invocation from Prolog into two steps. The first step resolves the methods, checks and possibly translates arguments as before. Now, instead of invoking the implementation, it returns a *continuation* (= goal). If the method is implemented in C, the implementation is called and the continuation is `true`. If the method is implemented in Prolog, the continuation is a call to the implementation for which we gave an example in figure 5.4. This implementation of `send/2` is illustrated by the code below and preserves both non-determinism and last-call optimisation.

```
:- pce_begin_class(my_node, node).

variable(data, prolog, both, "Associated data").

initialise(Node, Tree:prolog) :->
        Tree = node(Name, Data, Sons),
        send_super(Node, initialise(text(Name))),
        send(Node, data(Data)),
        forall(member(Son, Sons),
               send(Node, son(my_node(Son)))).

:- pce_end_class(my_node).
```

```
?- new(Tree, my_node(node(root, type(nounphrase),
                          [ node(dog, type(noun), []),
                            ...
                          ]))).
```

Figure 5.7: Class *my_node* creates a node and its children from a Prolog term, where each node carries a payload, also represented as a Prolog term. Bottom part shows an example invocation.

```
send(Object, Message) :-
        resolve_implementation(Object, Message, Continuation),
        call(Continuation).
```

We have not yet included this mechanism for preserving non-determinism because too much existing code relies on the implicit cut that now follows the implementation of each method. A possible solution is to add a declaration that specifies that the implementation is non-deterministic.

In this section we described passing Prolog data to methods implemented in Prolog, storing Prolog data in instance variables of the OO system and finally introduce non-determinism into methods defined in Prolog. We conclude with a brief analysis of portability and experience, where we omit non-determinism as this has not been implemented.

Portability Garbage collection in many object systems is not (only) based on reference counts and therefore the reference-count transparent change of status from term reference to a copy on the permanent heap is often not feasible. In that case one must opt for the first alternative as described in section 5.5.2. An interface as defined by PL_record()/PL_erase() is not commonly available in Prolog systems, but can be implemented easily in any system. Even without access to the source it is always possible to revert to an assert/retract-based implementation.

Experience The possibility to pass Prolog data around is used frequently, clearly simplifying and improving efficiency of methods that have to deal with application data that is already represented in Prolog, such as the parse tree generated by the SWI-Prolog SGML/XML parser (see section 7.2).

5.6 Performance evaluation

XPCE/Prolog message passing implies data conversion and foreign code invocation, slowing down execution. However, XPCE provides high-level (graphical) operations that limit the number of messages passed. Computation inside the Prolog application runs in native Prolog and is thus not harmed. For example, bottlenecks appear when manipulating bitmapped images at the pixel-level or when using Prolog-defined classes for fine-grained OO programming where good performance is a requirement.

Table 5.1 illustrates the performance on some typical method invocations through Prolog `send/2`. The first two rows call a C-defined built-in class involving no significant 'work'. We implemented class *bench* in Prolog, where the implementation of the method is a call to `true/0`. The first row represents the time to make a call to C and resolve the method implementation. The second adds checking of an integer argument while the last row adds the time to create and destroy the *prolog_term* instance. Finally, we add timing for Prolog calling Prolog and Prolog calling a C-defined built-in.

Accessing external functionality inside XPCE involves 5 times the overhead of adding C-defined predicates to Prolog. In return, we get object orientation, runtime type checking of arguments and optional arguments. Defining methods in Prolog doubles the overhead and is 10 times slower than direct Prolog-to-Prolog calls. Most XPCE methods realise significant functionality and the overhead is rarely a bottleneck.

5.7 Events and Debugging

An important advantage of the described interface is that all application-code is executed in Prolog and can therefore be debugged and developed using Prolog's native debugger and, with some restrictions described in section 5.4, Prolog's incremental compilation to update the environment while the application is running.

Goal	Class	Time (μS)
Prolog calling XPCE *a built-in method*		
`send(@426445, normalise)`	area	0.24
`send(@426445, x(1))`	area	0.31
Prolog calling XPCE *Prolog-defined method*		
`send(@426891, noarg)`	bench	0.54
`send(@426891, intarg(1))`	bench	0.69
`send(@426891, termarg(hello(world)))`	bench	0.65
Prolog calling Prolog		
`direct(@253535, hello(world))`	–	0.05
Prolog calling C		
`compound(hello(world))`	–	0.06

Table 5.1: Message passing performance, measured on an Intel X6800@2.93Ghz, SWI-Prolog 5.7.2 compiled with gcc 4.3 -O2

The Prolog debugger is faced with phenomena uncommon to the traditional Prolog world. The event-driven nature of GUI systems causes 'spontaneous' calls. Many user-interactions consist of a sequence of actions each causing their own events and Prolog call-backs. User interaction with the debugger may be difficult or impossible during such sequences. For example, call-backs resulting from dragging an object in the interface with the mouse cannot easily be debugged on the same console. The design also involves deeply nested control switches between foreign code and Prolog. The SWI-Prolog debugger is aware of the possibilities of interleaved control and provides hooks for presenting method-calls in a user-friendly fashion. Break-points in addition to the traditional spy-points make it easier to trap the debugger at interesting points during user-interaction. Figure 5.8 shows the source-level debugger in action on XPCE/Prolog code.

5.8 Related Work

To our best knowledge, there are no systems with a similar approach providing GUI to Prolog. Other approaches for accessing foreign GUI systems have been explored in section 5.2.

Started as a mechanism to provide a GUI, our approach has developed into a generic design to integrate Prolog seamlessly with an external OO programming language. The integrated system also functions as an object extension to Prolog and should therefore be compared to other approaches for representing objects in Prolog. Given the great diversity of such systems (Moura 2008), we consider this beyond the scope of this discussion.

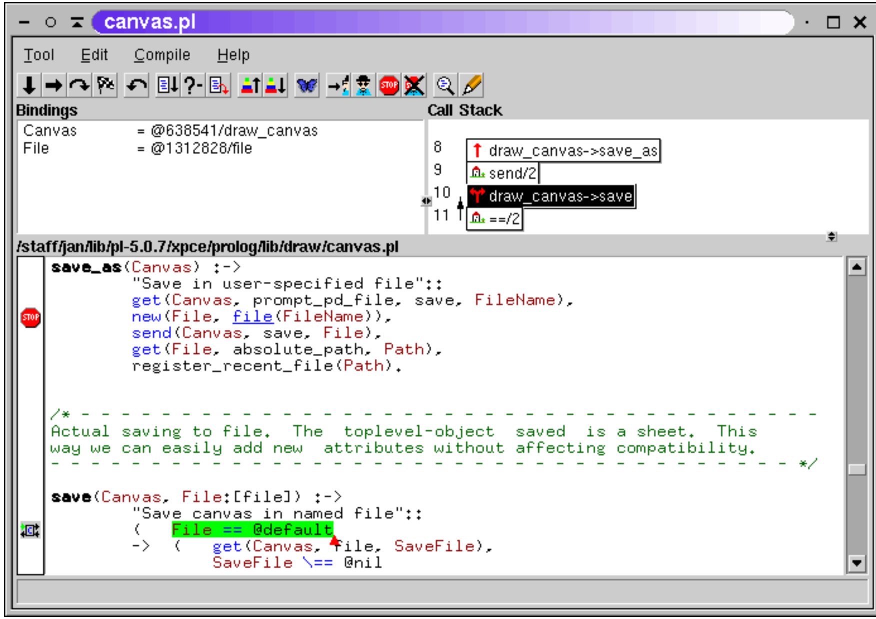


Figure 5.8: SWI-Prolog source-level debugger showing break-point and interleaved foreign/Prolog stack context.

5.9 Conclusions and discussion

We have presented an architecture for integrating an external Object Oriented system with minimal reflexive capabilities (i.e., calling a method by name) with Prolog. In most of the document we described the architecture of XPCE/SWI-Prolog and commented portability aspects when replacing XPCE with another OO system. The design allows for extending many existing OO systems naturally from Prolog. Using this interface the user can add new classes to the OO system entirely from Prolog which can be used to extend the OO system, but also for object-oriented programming in Prolog. Programming requires knowledge of the OO system's classes and methods, but requires *no* knowledge of the OO system's control primitives and syntax.

Using dynamically typed OO systems where classes and methods can be created at runtime through the interface without generating a source file, a quick and natural development cycle is achieved. If however, less of the OO system is accessible at runtime, development becomes more cumbersome because more changes to the code will require the user to restart the application.

Creating derived classes is often required to make effective use of an existing OO system, for example for refining behaviour of GUI controls. Our mechanism satisfies this requirement and allows for a tight integration between Prolog and the GUI classes. The ability to create derived classes from Prolog provides a uniform interface to the core of the OO system and extensions realised in Prolog libraries. Classes defined in Prolog form an appropriate organisation mechanism for the GUI part of applications.

The proposed solution can be less ideal for general purpose OO programming in Prolog for two reasons. First, the OO features of the OO platform dictate the OO features of our hybrid environment. For example, XPCE does not provide multiple inheritance nor polymorphism on argument types and therefore these features are lacking from XPCE/Prolog programs. Multiple inheritance can be attractive for data modelling. Second, the message passing overhead is relatively high. This is acceptable for course-grain organisation of applications and many user interface tasks, but it might be unacceptable for general purpose OO programming in Prolog. However, introducing an OO system for application programming next to interfacing to an external OO system as described in this paper is likely to confuse programmers. The discussion on supporting graphics from Prolog is continued in the overall conclusions of this thesis, section 11.3.1.

Acknowledgements

XPCE/SWI-Prolog is a Free Software project which, by nature, profits heavily from user feedback and participation. We would like to thank Mats Carlsson in particular for his contribution to designing the representation of XPCE methods in Prolog. We also acknowledge the reviewers for their extensive comments and suggestions, many of which have been used to clarify this paper.

Chapter 6

Native Preemptive Threads in SWI-Prolog

About this chapter This chapter was published at the ICLP-03 (Wielemaker 2003a). It describes adding multi-threading to Prolog, a requirement for creating scalable web services and therefore part of research question 2. Threads also play a vital role in the mediator-based MVC design that we used to implement Triple20 (chapter 2, Wielemaker et al. 2005). The SWI-Prolog implementation is the basis for the ISO standard for adding threads to Prolog (Moura et al. 2008) and has already been implemented by two major Open Source Prolog systems (YAP and XSB). Section 6.6.1 has been updated to reflect enhancements made since this paper was published.

Abstract Concurrency is an attractive property of a language to exploit multi-CPU hardware or perform multiple tasks concurrently. In recent years we see Prolog systems experimenting with multiple threads only sharing the database. Such systems are relatively easy to build and remain close to standard Prolog while providing valuable extra functionality. This article describes the introduction of multiple threads in SWI-Prolog exploiting OS-native support for threads. We discuss the extra primitives available to the Prolog programmer as well as implementation issues. We explored speedup on multi-processor hardware and speed degradation when executing a single task.

6.1 Introduction

There are two approaches to concurrency in the Prolog community, implicit fine-grained parallelism where tasks share Prolog variables and implementations (see section 6.7) in which Prolog engines only share the database (clauses) and run otherwise completely independent. We call the first class *parallel* logic programming systems and the latter *multi-threaded* systems. Writing programs for *multi-threaded* Prolog is close to normal Prolog programming, which makes multi-threading attractive for applications that benefit from course grained concurrency. Below are some typical use-cases.

- *Network servers/agents*
  Network servers must be able to pay attention to multiple clients. Threading allows multiple, generally almost independent, tasks to make progress at the same time. This can improve overall performance by exploiting multiple CPUs (SMP) or by better utilising a single CPU if (some) tasks are I/O bound. Section 6.4.1 provides an example.

- *Embedding in multi-threaded servers*
  Concurrent network-service infrastructures such as CORBA or .NET that embed a single threaded Prolog engine must serialise access to Prolog. If Prolog is responsible for a significant amount of the computation Prolog becomes a bottleneck. Using a multi-threaded Prolog engine the overall concurrent behaviour of the application can be preserved.

- *Background processing in interactive systems*
  Responsiveness and usefulness of interactive applications can be improved if background processing deals with tasks such as maintaining *mediators* (section 2.4), spell-checking and syntax-highlighting. Implementation as a foreground process either harms response-time or is complicated by interaction with the GUI event-handling.

- CPU-*intensive tasks*
  On SMP systems CPU-intensive tasks that can easily be split into independent subtasks can profit from a multi-threaded implementation. Section 6.6.2 describes an experiment.

This article is organised as follows: in section 6.2 we establish requirements for multi-threaded Prolog that satisfy the above use cases. In subsequent sections we motivate choices in the design and API. Section 6.5 provides an overview of the implementation effort needed to introduce threads in a single threaded Prolog implementation, where we pay attention to atom garbage collection. We perform two performance analysis: the first explores the performance loss when running single-threaded applications on a multi-threaded system while the second explores speedup when running a CPU-intensive job on multi-CPU hardware. Finally we present related work and draw our conclusions.

6.2 Requirements

Combining the use-cases from the introduction with the need to preserve features of interactive program development in Prolog such as aborting execution and incremental recompilation during debugging, we formulate the following requirements:

- *Smooth cooperation with (threaded) foreign code*
  Prolog applications operating in the real world often require substantial amounts of 'foreign' code for interaction with the outside world: window-system interface, interfaces to dedicated devices and networks. Prolog threads must be able to call arbitrary

foreign code without blocking the other (Prolog-)threads and foreign code must be able to create, use and destroy Prolog engines.

- *Simple for the Prolog programmer*
  We want to introduce few and easy to use primitives to the Prolog programmer.

- *Robust during development*
  We want to be as robust as feasible during interactive use and the test-edit-reload development cycle. In particular this implies the use of synchronisation elements that will not easily create deadlocks when used incorrectly.

- *Portable implementation*
  We want to be able to run our multi-threaded Prolog with minimal changes on all major hardware and operating systems.

Notably the first and last requirement suggests to base Prolog threads on the POSIX thread API (Butenhof 1997). This API offers preemptive scheduling that cooperates well with all (blocking) operating system calls. It is well supported on all modern POSIX-based systems. On MS-Windows we use a mixture of pthread-win32[1] for portability and the native Win32 thread-API where performance is critical.

6.3 What is a Prolog thread?

A Prolog thread is an OS-native thread running a Prolog *engine*, consisting of a set of stacks and the required state to accommodate the engine. After being started from a *goal* it proves this goal just like a normal Prolog implementation by running predicates from a *shared* program space. Figure 6.1 illustrates the architecture. As each engine has its own stacks, Prolog terms can only be transferred between threads by copying. Both dynamic predicates and FIFO queues of Prolog terms can be used to transfer Prolog terms between threads.

6.3.1 Predicates

By default, all predicates, both static and dynamic, are shared between all threads. Changes to static predicates only influence the test-edit-reload cycle, which is discussed in section 6.5. For dynamic predicates we kept the 'logical update semantics' as defined by the ISO standard (Deransart et al. 1996). This implies that a goal uses the predicate with the clause-set as found when the goal was started, regardless of whether clauses are asserted or retracted by the calling thread or another thread. The implementation ensures consistency of the predicate as seen from Prolog's perspective. Consistency as required by the application such as clause order and consistency with other dynamic predicates must be ensured using *synchronisation* as discussed in section 6.3.2.

[1] `http://sources.redhat.com/pthreads-win32/`

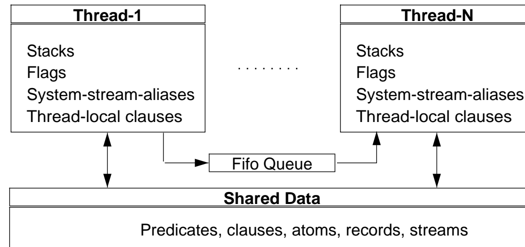


Figure 6.1: Multiple Prolog engines sharing the same database. Flags and the system-defined stream aliases such as `user_input` are copied from the creating thread. Clauses are normally shared, except for thread-local clauses discussed below in section 6.3.1.

In contrast, *Thread-local* predicates are dynamic predicates that have a different set of clauses in each thread. Modifications to such predicates using `assert/1` or `retract/1` are only visible from the thread that performs the modification. In addition, such predicates start with an empty clause set and clauses remaining when the thread dies are automatically removed. Like the related POSIX thread-specific data primitive, thread-local predicates simplifies making code designed for single-threaded use *thread-safe*.

6.3.2 Synchronisation

The most difficult aspect of multi-threaded programming is the need to *synchronise* the concurrently executing threads: ensure they use proper protocols to exchange data and maintain invariants of *shared-data* in dynamic predicates. Given the existence of the POSIX thread standard and our decision to base our thread implementation on this standard for portability reasons, we must consider modelling the Prolog API after it. POSIX threads offer two mechanisms to organise thread synchronisation:

- *A mutex*
  is a **Mut**ual **Ex**clusive device. At most one thread can 'hold' a mutex. By associating a mutex to data it can be assured only one thread has access to this data at one time, allowing it to maintain the invariants.

- *A condition variable*
  is an object that can be used to wait for a certain *condition*. For example, if data is not in a state where a thread can start using it, a thread can wait on a condition variable associated with this data. If another thread updates the data it *signals* the condition variable, telling the waiting thread something has changed and it may re-examine the condition.

As Butenhof (1997) explains in chapter 4, the commonly used thread cooperating techniques can be realised using the above two primitives. However, these primitives are not suitable for the Prolog user because great care is required to use them in the proper order and to complete all steps of the protocol. Failure to do so may lead to data corruption or to a deadlock where all threads are waiting for an event to happen that never will. Non-determinism, exceptions and the interactive development-cycle supported by Prolog complicate this further.

Examining other systems (section 6.7), we find a more promising synchronisation primitive in the form of a FIFO (first-in-first-out) queue of Prolog terms. Queues (also called *channels* or *ports*) are well understood, easy to understand by non-experts in multi-threading, can safely handle abnormal execution paths (backtracking and exceptions) and can naturally represent serialised flow of data (*pipeline*). Next to the FIFO queues we support goals guarded by a mutex by means of `with_mutex`(*Mutex, Goal*) as defined in section 6.4.2.

6.3.3 I/O and debugging

Support for multi-threaded I/O is rather primitive. I/O streams are global objects that may be created, accessed and closed from any thread knowing their handle. All I/O predicates lock a mutex associated with the stream, providing elementary consistency, but the programmer is responsible for proper closing the stream and ensuring streams are not accessed by any thread after closing them.

Stream alias names for the system streams (e.g., `user_input`) are *thread-specific*, where a new thread starts with the current bindings in its creator. Local system stream aliases allow us to re-bind the user streams and provide separate interaction consoles for each thread as implemented by `attach_console/0`. The console is realised using a clone of the normal SWI-Prolog console on Windows or an instance of the `xterm` application in Unix. The predicate `interactor/0` creates a thread, attaches a console and runs the Prolog toplevel.

Using `thread_signal/2` to execute `attach_console/0` and `trace/0` in another thread, the user can attach a console to any thread and start the debugger in any thread as illustrated in figure 6.2.

6.4 Managing threads from Prolog

An important requirement is to make threads easy for the programmer, especially for the task we are primarily targeting at, interacting with the outside world. First we start with an example, followed by a partial description of the Prolog API and the consequences for the foreign language interface.

6.4.1 A short example

Before describing the details, we present the implementation of a simple network service in figure 6.3. We will not discuss the details of all built-in and library predicates used in this

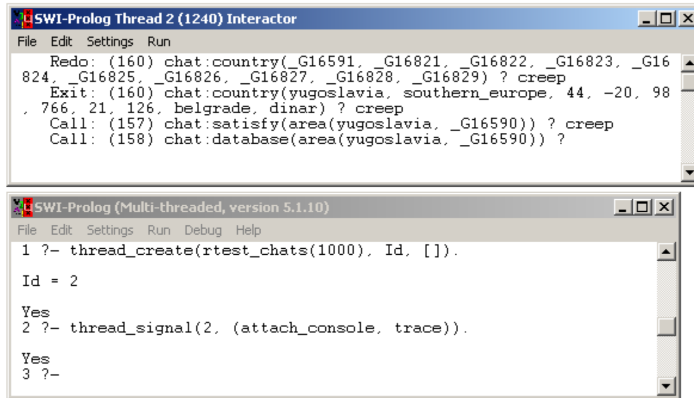


Figure 6.2: Attach a console and start the debugger in another thread.

example. The thread-related predicates are discussed in more detail in section 6.4.2 while all details can be found in the SWI-Prolog reference manual (Wielemaker 2008). Our service handles a single TCP/IP request per connection, using a specified number of 'worker threads' and a single 'accept-thread'. The accept-thread executes `acceptor/2`, accepting connection requests and adding them to the queue for the workers. The workers execute `worker/1`, getting the accepted socket from the queue, read the request and execute `process/2` to compute a reply and write this to the output stream. After this, the worker returns to the queue for the next request.

The advantages of this implementation over a traditional single-threaded Prolog implementation are evident. Our server exploits SMP hardware and will show much more predictable response times, especially if there is a large distribution in the time required by `process/1`. In addition, we can easily improve on it with more monitoring components. For example, `acceptor/2` could immediately respond with an estimated reply time, and commands can be provided to examine and control activity of the workers. Using multi-threaded code, such improvements do not affect the implementation of `process/2`, keeping this simple and reusable.

6.4.2 Prolog primitives

This section discusses the main features of built-in predicates we have added to Prolog to facilitate threads. A full description is in the SWI-Prolog reference manual (Wielemaker 2008).

thread_create(*:Goal, -Id, +Options*)

Create a thread which starts executing *Goal*. *Id* is unified with the thread-identifier. In

```
:- use_module(library(socket)).

make_server(Port, Workers) :-
    create_socket(Port, S),
    message_queue_create(Q),
    forall(between(1, Workers, _),
           thread_create(worker(Q), _, [])),
    thread_create(acceptor(S, Q), _, []).

create_socket(Port, Socket) :-
    tcp_socket(Socket),
    tcp_bind(Socket, Port),
    tcp_listen(Socket, 5).
```

```
acceptor(Socket, Q) :-
    tcp_accept(Socket, Client, _Peer),
    thread_send_message(Q, Client),
    acceptor(Socket, Q).
```

```
worker(Q) :-
    thread_get_message(Q, Client),
    tcp_open_socket(Client, In, Out),
    read(In, Command),
    close(In),
    process(Command, Out),
    close(Out),
    worker(Q).

process(hello, Out) :-
    format(Out, 'Hello world!~n', []).
```

Figure 6.3: Implementation of a multi-threaded server. Threading primitives are set in bold. The left column builds the server. The top-right runs the *acceptor* thread, while the bottom-right contains the code for a *worker* of the crew.

the calling thread, `thread_create/3` returns immediately. The new Prolog engine runs independently. Threads can be created in two modes: *attached* and *detached*. Completion of *Attached* threads must be followed by a call to `thread_join/2` to retrieve the result-status and reclaim all resources. *Detached* threads vanish automatically after completion of *Goal*. If *Goal* terminated with failure or an exception, a message is printed to the console. *Options* is an ISO option list providing the mode, a possible alias name and runtime parameters such as desired stack limits.

thread_join(+*Id, -Result***)**
Wait for the thread *Id* to finish and unify *Result* with the completion status, which is one of `true`, `false` or `exception`(*Term*).

message_queue_create(-*Queue, +Options***)**
Create a FIFO message queue (*channel*). Message queues can be read from multiple threads. Each thread has a message queue (*port*) attached as it is created. *Options* allows naming the queue and define a maximum size. If the queue is full, writers are suspended.

thread_send_message(+*QueueOrThread, +Term***)**
Add a copy of term to the given queue or default queue of the thread. Suspends the caller if the queue is full.

thread_get_message(*[+Queue], ?Term*)

Get a message from the given queue (*channel*) or default queue if *Queue* is omitted (*port*). The first message that unifies with *Term* is removed from the queue and returned. If multiple threads are waiting, only one will be given the term. If the queue has no matching terms, execution of the calling thread is suspended.

with_mutex(*+Name, :Goal*)

Execute *Goal* as `once/1` while holding the named mutex. *Name* is an atom. Explicit use of mutex objects is used to serialise access to code that is not designed for multi-threaded operation as well as coordinate access to shared dynamic predicates. The example below updates `address/2`. Without a mutex another thread may see no address for *Id* if it executes just between the `retractall/1` and `assert/1`.

```
set_address(Id, Address) :-
        with_mutex(address, (retractall(address(Id, _)),
                             assert(address(Id, Address)))).
```

thread_signal(*+Thread, :Goal*)

Make *Thread* execute *Goal* on the first opportunity, i.e., run *Goal* in *Thread* as an *interrupt*. If *Goal* raises an exception, this exception is propagated to the receiving thread. This ability to raise an exception in another thread can be used to abort threads. See below. Long running and blocking foreign code may call PL_handle_signals() to execute pending signals and return control back to Prolog if PL_handle_signals() indicates that the handler raised an exception by returning -1.

Signalling threads is used for debugging purposes (figure 6.2) and 'manager' threads to control their 'work-crew'. Figure 6.4 shows the code of both the worker and manager needed to make a worker stop processing the current job and obtain a new job from a queue. Figure 6.6 shows the work-crew design pattern to which this use case applies.

Worker	**Manager**

worker(Queue) :-
  thread_get_message(Queue, Work),
  catch(do_work(Work), **stop**, cleanup),
  worker(Queue).
	...
	thread_signal(Worker, **throw(stop)**),
	...

Figure 6.4: Stopping a worker using `thread_signal/2`. Bold fragments show the relevant parts of the code.

6.4.3 Accessing Prolog threads from C

Integration with foreign (C-)code has always been one of the main design goals of SWI-Prolog. With Prolog threads, flexible embedding in multi-threaded environments becomes feasible. We identify three use cases. Some applications in which we want to embed Prolog use few threads that live long, while others frequently created and destroy threads and finally, there are applications with large numbers of threads.

Compared to POSIX threads in C, Prolog threads use relatively much memory resources and creating and destroying a Prolog thread is relatively expensive. The first class of applications can associate a Prolog thread (engine) with every thread that requires Prolog access (1-to-1-design, see below), but for the other two scenarios this is not desirable and we developed an N-to-M-design:

- *1-to-1-design*
  The API PL_thread_attach_engine() creates a Prolog engine and makes it available to the thread for running queries. The engine may be destroyed explicitely using PL_thread_destroy_engine() or it will be destroyed automatically when the underlying POSIX thread terminates.

- *N-to-M-design*
  The API PL_create_engine() creates a Prolog engine that is not associated to any thread. Multiple calls can be used to create a pool of M *engines*. Threads that require access to Prolog claim and release an engine using PL_set_engine(). Claiming and releasing an engine is a fast operation and the system can realise a suitable pool of engines to balance concurrency and memory requirements. A demo implementation is available.[2]

6.5 Implementation issues

We tried to minimise the changes required to turn the single-engine and single-threaded SWI-Prolog system into a multi-threaded version. For the first implementation we split all global data into three sets: data that is initialised when Prolog is initialised and never changes afterwards, data that is used for shared data-structures, such as atoms, predicates, modules, etc. and finally data that is only used by a single engine such as the stacks and virtual machine registers. Each set is stored in a single C-structure, using thread-specific data (section 6.3.2) to access the engine data in the multi-threaded version. Update to shared data was serialised using mutexes.

A prototype using this straight-forward transition was realised in only two weeks, but it ran slowly due to too heavy use of pthread_getspecific() and too many mutex synchronisation points. In the second phase, fetching the current engine using pthread_getspecific() was reduced by caching this information inside functions that use it multiple times and passing

[2] `http://gollem.science.uva.nl/twiki/pl/bin/view/Development/MultiThreadEmbed`

it as an extra variable to commonly used small functions as identified using the gprof (Graham et al. 1982) profiling tool. Mutex contention was analysed and reduced from some critical places:[3]

- All predicates used reference counting to clean up deleted clauses after retract/1 for dynamic or (re-)consult/1 for static code. Dynamic clauses require synchronisation to make changes visible and cleanup erased clauses, but static code can do without this. Reclaiming dead clauses from static code as a result of the test-edit-recompile cycle is left to a garbage collector that operates similarly to the atom garbage collection described in section 6.5.1.

- Permanent heap allocation uses a pool of free memory chunks associated with the thread's engine. This allows threads to allocate and free permanent memory without synchronisation.

6.5.1 Garbage collection

Stack garbage collection is not affected by threading and continues concurrently. This allows for threads under real-time constraints by writing them such that they do not perform garbage collections, while other threads can use garbage collection.

Atom garbage collection is more complicated because atoms are shared global resources. Atoms referenced from global data such as clauses and records use reference counting, while atoms reachable from the stacks are marked during the marking phase of the atom garbage collector. With multiple threads this implies that all threads have to mark their atoms before the collector can reclaim unused atoms. The pseudo code below illustrates the signal-based implementation used on Unix systems.

```
atom_gc()
{ mark_atoms_on_stacks();                   // mark my own atoms
  foreach(thread except self)               // ask the other threads
  { pthread_kill(thread, SIG_ATOM_GC);
    signalled++;
  }
  while(signalled-- > 0)                    // wait until all is done
    sem_wait(atom_semaphore);
  collect_unmarked_atoms();
}
```

A thread receiving SIG_ATOM_GC calls mark_atoms_on_stacks() and signals the atom_semaphore semaphore when done. The mark_atoms_on_stacks() function is designed such that it is safe to call it asynchronously. Uninitialised variables on the Prolog stacks may be interpreted incorrectly as an atom, but such mistakes are infrequent and can be corrected in a later run of the garbage collector. The atom garbage collector holds the

[3] See *Update* at the end of section 6.6 for additional places were we reduced contention.

atom mutex, preventing threads to create atoms or change the reference count. The marking phase is executed in parallel.

Windows does not provides asynchronous signals and synchronous (cooperative) marking of referenced atoms is not acceptable because the invoking thread as well as any thread that wishes to create an atom must block until atom GC has completed. Therefore the thread that runs the atom garbage collector uses SuspendThread() and ResumeThread() to stop and restart each thread in turn while it marks the atoms of the suspended thread.

Atom-GC and GC interaction SWI-Prolog uses a sliding garbage collector (Appleby et al. 1988). During the execution of GC, it is hard to mark atoms. Therefore during atom-GC, GC cannot start. Because atom-GC is such a harmful activity, we should avoid it being blocked by a normal GC. Therefore the system keeps track of the number of threads executing GC. If GC is running in some thread, atom-GC is delayed until no thread executes GC.

6.5.2 Message queues

Message queues are implemented using POSIX condition variables using the recorded database for storing Prolog terms in the queue. Initially the implementation used the pthreads-win32 emulation on Windows. In the current implementation this emulation is replaced by a native Windows alternative (Schmidt and Pyarali 2008) which does not comply fully to the POSIX semantics for condition variables, but provides an approximately 250 times better throughput of messages. The incorrectness has no implications for our purposes.

The SWI-Prolog recorded database cooperates with the atom garbage collector using atom reference counts: recording a term increments the reference count of each atom that appears in it and erasing the record decrements the reference counts. In both message queues and `findall/3`, recorded terms fulfil the role of temporary storage and the need to synchronise twice for every atom in a term has proven to be a major performance bottleneck. In the current implementation, atoms in temporary recorded terms are no longer registered and unregistered. Instead, atom garbage collection marks atoms that appear in queued records and the solution store of `findall/3`. See also the *update* paragraph at the end of section 6.6.

6.6 Performance evaluation

Our aim was to use the multi-threaded version as default release version, something which is only acceptable if its performance running a normal non-threaded program is close to the performance of the single-threaded version, which is investigated in section 6.6.1. In section 6.6.2 we studied the speedup on SMP systems by splitting a large task into subtasks that are distributed over a pool of threads.

6.6.1 Comparing multi-threaded to single threaded version

We used the benchmark suite by Fernando Pereira[4] for comparing the single threaded to the multi threaded version. The Pereira benchmark set consists of 34 tests, each of which tests a specific aspect of a Prolog implementation. The tests themselves are small and iterated many times to run for a measurable time. We normalised the iterations of each test to make it run for approximately one second on our base case: single-threaded SWI-Prolog on Linux. We ran the benchmarks in five scenarios on the same hardware, a machine with an 550Mhz Crusoe CPU running SuSE Linux 7.3 and Windows 2000 in dual-boot mode.

Bar	Threading	OS	Comments
1	Single	Linux	Our *base-case*.
2	Single	Linux	With extra variable. See description.
3	Multi	Linux	Normal release version.
4	Single	Windows	Compiled for these tests.
5	Multi	Windows	Normal release version.

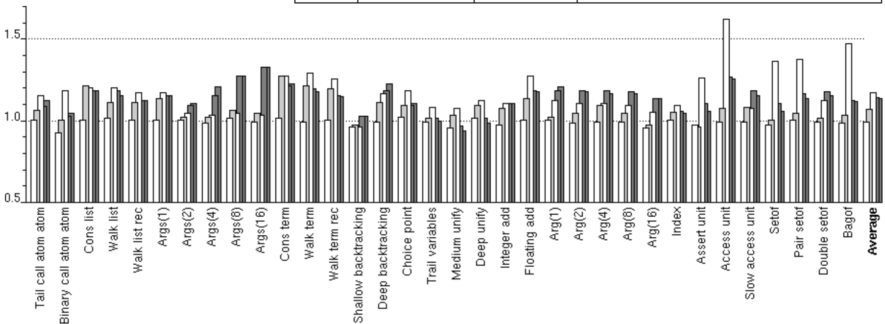


Figure 6.5: Performance comparison between single and multi-threaded versions. The Y-axis shows the time to complete the benchmark in seconds. The legend above summarises the test setting represented by each bar. The rightmost 5 bars show the average.

Figure 6.5 shows the results. First, we note there is no significant difference on any of the tests between the single- and multi-threaded version on Windows 2000 (bars 4&5). The figure does show a significant difference for Linux running on the same hardware (bars 1&3). Poor performance of Linux on 'assert unit', 'access unit' and the setof/bagof tests indicates a poor implementation of mutexes that are used for synchronising access to dynamic predicates and protecting atoms in records used for storing the setof/bagof results.

The slow-down on the other tests cannot be explained by synchronisation primitives as they need no synchronisation. The state of the virtual machine in the single threaded version is stored in a global structure, while it is accessible through a pointer passed between functions in the multi threaded version. To explain the differences on Linux we first compiled a

[4] http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/pereira.txt

version that passes a pointer to the virtual machine state but is otherwise identical to the single threaded version. This version (2nd bar) exhibits behaviour similar to the multi-threaded (3th bar) version on many of the tests, except for the tests that require heavy synchronisation. We conclude that the extra variable and argument in many functions is responsible for the difference. This difference does not show up in the Windows version. We see two possible explanations for that. First, Unix shared object management relies on *position independent* code, for which it uses a base register (EBX on IA32 CPUs), while Windows uses *relocation* of DLLs. This implies that Unix systems have one register less for application code, a significant price on a CPU with few registers. Second, because overall performance of the single threaded version is better on Linux, it is possible that overall optimisation of gcc 2.95 is better than MSVC 5 or to put in differently, Windows could have been faster in single threaded mode if the optimisation of MSVC 5 would have been better.

Finally, we give the average result (last column of figure 6.5) comparing the single-threaded with the multi-threaded version (cases 1 and 3 in figure 6.5) for a few other platforms and compilers. Dual AMD-Athlon, SuSE 8.1, gcc 3.1: -19%; Single UtraSPARC, Solaris 5.7, gcc 2.95: -7%; Single Intel PIII, SuSE 8.2, gcc 3.2: -19%. Solaris performs better on the mutex-intensive tests.

Update We re-ran the Linux comparison on modern hardware: AMD Athlon X2 5400+ dual core CPU running Linux 2.6.22, glibc 2.6.1 and gcc 4.2.1. Both single and multi-threaded versions where compiled for the AMD64 (x86_64) architecture, which provides 16 instead of 8 general purpose registers. Since publication of this paper we changed the implementation of `findall/3` to use the variant of recorded terms described in section 6.5.2 to avoid the need for synchronisation in the all-solution predicates.

The average performance loss over the 34 tests is now only 4%. This is for a large part caused by the dynamic predicate tests (assert_unit and access_unit; -20% and -29%). Because `findall/3` and friends no longer require significant synchronisation, the multi-threaded version performs practically equal to the single-threaded version on the last four tests of figure 6.5 (setof . . . bagof).

6.6.2 A case study: Speedup on SMP systems

This section describes the results of multi-threading the Inductive Logic Programming system Aleph (Srinivasan 2003), developed by Ashwin Srinivasan at the Oxford University Computing Laboratory. Inductive Logic Programming (ILP) is a branch of machine learning that synthesises logic programs using other logic programs as input.

The main algorithm in Aleph relies on searching a space of possible general clauses for the one that scores best with respect to the input logic programs. Given any one example from the input, a lattice of plausible single-clauses ordered by generality is bound from above by the clause with `true` as the body ($\top$), and bound from below by a long (up to hundreds of literals) clause known as the most-specific-clause (or bottom, $\bot$) (Muggleton 1995).

Many strategies are possible for searching this often huge lattice. Randomised local search (Železný et al. 2003) is one form implemented in Aleph. Here a node in the lattice is selected at random as a starting location to *(re)-start* the search. A finite number of *moves* (e.g., radially from the starting node) are made from the start node. The best scoring node is recorded, and another node is selected at random to restart the search. The best scoring node from all restarts is returned.

As each restart in a randomised local search of the lattice is independent, the search can be multi-threaded in a straight forward manner using the worker-crew model, with each worker handling moves from a random start point and returning the best clauses as depicted in figure 6.6. We exploited the thread-local predicates described section 6.3.1 to make the working memory of the search kept in dynamic predicates local to each worker.

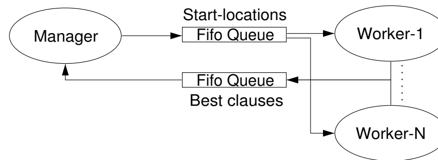


Figure 6.6: Concurrent Aleph. A manager schedules start points for a crew of workers. Each worker computes the best clause from the neighbourhood of the start point, delivers it to the manager and continues with the next start-point.

6.6.2.1 Experimental results and discussion

An exploratory study was performed to study the speedup resulting from using multiple threads on an SMP machine. We realised a work-crew model implementation for randomised local search in Aleph version 4. As the task is completely CPU bound we expected optimal results if the number of threads equals the number of utilised processors.[5] The task consisted of 16 random restarts, each making 10 *moves* using the *carcinogenesis* (King and Srinivasan 1996) data set.[6] This task was carried out using a work-crew of 1, 2, 4, 8 and 16 workers scheduled on an equal number of CPUs. Figure 6.7 shows that speedup is nearly optimal upto about 8 CPUs. Above 8, synchronisation overhead prevents further speedup. Note that later enhancements to messages queues as described in section 6.5.2 were not available for these tests.

The above uses one thread per CPU, the optimal scenario for purely CPU bound tasks. We also assessed the performance when using many threads per CPU using Aleph. These results indicate the penalty of converting a single-threaded design into a multi-threaded one.

[5] We forgot to reserve a CPU for the manager. As it has little work to do we do not expect results with an additional CPU for the manager to differ significantly from our results.

[6] ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Datasets/carcinogenesis/progol/carcinogenesis.ta

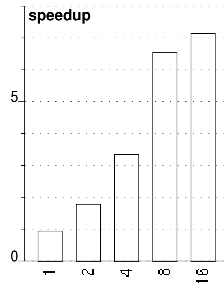


Figure 6.7: Speedup with an increasing number of CPUs defined as elapsed time using one CPU divided by elapsed time using N CPUs. The task consisted of 16 *restarts* of 10 *moves*. The values are averaged over 30 runs. The study was performed on a Sun Fire 6800 with 24 UltraSPARC III 900 MHz Processors, 48 GB of shared memory, utilising up to 16 processors.

Figure 6.8 shows the results. The X-axis show the number of used threads with the same meaning as used in the above experiment. The Y-axis shows both the (user) CPU time and the elapsed time. The top-two graphs show that on single CPU hardware there is no handicap upto 8 threads. With 16 and 32 threads, overhead starts to grow quickly. On dual-CPU hardware (bottom-two graphs), the situation is slightly different. The point for 1 thread (left) illustrate the difference in CPU speed between our single-CPU platform and SMP platform. With two threads, CPU time remains almost the same and elapsed time is reduced to almost 50%. As more threads are used, both CPU and elapsed time increase much faster than in the single-CPU scenario.

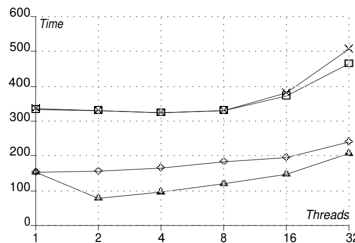


Figure 6.8: CPU- and elapsed time running Aleph concurrent on two architectures. The top two graphs are executed on a single CPU Intel PIII/733 Mhz, SuSE 8.2. The bottom two graphs are executed on a dual Athlon 1600+, SuSE 8.1. The X and triangle marked graps represent elapsed time.

6.7 Related Work

This section provides an incomplete overview of other Prolog implementations providing multi-threading where threads only share the database. Many implementations use message queues (called *port* if the queue is an integral part of the thread or *channel* if they can be used by multiple threads).

SICStus-MT(Eskilson and Carlsson 1998) describes a prototype implementation of a multi-threaded version of SICStus Prolog based on the idea to have multiple Prolog engines only sharing the database. They used a proprietary preemptive scheduler for the prototype and therefore cannot support SMP hardware and have trouble with clean handling of blocking system-calls. The programmer interface is similar to ours, but they do not provide queues (channels) with multiple readers, nor additional synchronisation primitives.

CIAO Prolog[7] (Carro and Hermenegildo 1999) provides preemptive threading based on POSIX threads. The referenced article also gives a good overview of concurrency approaches in Prolog and related languages. Their design objectives are similar, though they stress the ability to backtrack between threads as well as Linda-like (Carriero and Gelernter 1989) blackboard architectures. Threads that succeed non-deterministically can be restarted to produce an alternative solution and instead of queues they use 'concurrent' predicates where execution suspends if there is no alternative clause and is resumed after another thread asserts a new clause.

Qu-Prolog[8] provides threads using its own scheduler. Thread creation is similar in nature to the interface described in this article. Thread communication is, like ours, based on exchanging terms through a queue attached to each thread. For atomic operations it provides `thread_atomic_goal/1` which freezes all threads. This operation is nearly impossible to realise on POSIX threads. Qu-Prolog supports `thread_signal/2` under the name `thread_push_goal/2`. For synchronisation it provides `thread_wait/1` to wait for arbitrary changes to the database.

Multi-Prolog(de Bosschere and Jacquet 1993) is logic programming instantiation of the Linda blackboard architecture. It adds primitives to 'put' and 'get' both passive Prolog literals and active Prolog atoms (threads) to the blackboard. It is beyond the scope of this article to discuss the merits of message queues vs. a blackboard.

6.8 Discussion and conclusions

We have demonstrated the feasibility of supporting preemptive multi-threading using portable POSIX thread primitives in an existing Prolog system developed for single-threading. Built on the POSIX thread API, the system has been confirmed to run unmodified on six Unix dialects. The MacOS X version requires special attention due to incomplete support for POSIX semaphores. The Windows version requires a fundamentally different implementation of atom garbage collection due to the lack of asynchronous signals. The

[7] `http://clip.dia.fi.upm.es/Software/Ciao/`
[8] `http://www.svrc.uq.edu.au/Software/QuPrologHome.html`

pthreads-win32 emulation needed replacements using the native Windows API at various places to achieve good performance.

Concurrently running static Prolog code performs comparable to the single-threaded version and scales well on SMP hardware, provided that the threads need little synchronisation. Synchronisation issues appear in:

- *Dynamic predicates*
  Dynamic predicates require mutex synchronisation on assert, retract, entry and exit. Heavy use of dynamic code can harm efficiency significantly. The current system associates a unique mutex to each dynamic predicate. Thread-local code only synchronises on entry to retrieve the clause list that belongs to the calling thread. In the future, we will consider using lock-free primitives (Gidenstam and Papatriantafilou 2007) for synchronising dynamic code, in particular thread-local code.

- *Atoms*
  Creating an atom, creating a reference to an atom from `assert/1` or `recorda/1` as well as erasing records and clauses referencing atoms require locking the atom table. Worse, atom garbage collection affects all running threads, harming threads under tight real-time constraints. Multi-threaded applications may consider using SWI-Prolog's non-standard packed strings to represent text concisely without locking.

- *Meta-calling*
  Meta-calling requires synchronised mapping from module and functor to predicate. The current system uses one mutex per module.

- *Passing messages over a queue*
  Despite the enhancements described in section 6.5.2, passing messages between threads requires copying them twice and locking the queue by both the sender and receiver.

POSIX mutexes are stand-alone entities and thus not related to the data they protect through any formal mechanism. This also holds for our Prolog-level mutexes. Alternatively a lock could be attached to the object it protects (e.g., a dynamic predicate). We have not adopted this model as we regard the use of explicit mutex objects restricted to rare cases and the current model using stand-alone mutexes is more flexible.

The interface presented here does, except for the introduction of message queues, not abstract much from the POSIX primitives. Since the original publication of this paper we added higher level abstractions implemented as library predicates. The predicate `concurrent`(*N,Goals,Options*) runs M goals using N threads and stops if all work is done or a thread signalled failure or an error, while `first_solution`(*X,Goals,Options*) tries alternative strategies to find a value for *X*, stopping after the first solution. The latter is useful if there are different strategies to find an answer (e.g., literal ordering or breath vs. depth first search) and no way of knowing the best. We integrated threads into the development tools, providing source-level debugging for threads and a graphical monitor that displays status and resource utilisation of threads.

Manipulation of Prolog engines from foreign code (section 6.4.3) is used by the bundled Java interface (JPL) contributed by Paul Singleton as well as the C#[9] interface contributed by Uwe Lesta.

Multi-thread support has proved essential in the development of Triple20 (chapter 2, Wielemaker et al. 2005) where it provides a responsive GUI application and ClioPatria (chapter 10, Wielemaker et al. 2008), where it provides scalability to the web server. Recently, we see a slowdown in the paste with which individual cores[10] become faster. Instead, recent processors quickly accommodate more cores on one chip. This development demands support for multiple cores. The approach we presented allows for exploiting such hardware without much change to common practice in Prolog programming for the presented use-cases. In particular, Prolog hosted web-services profit maximally with almost no consequences to the programmer. Exploiting multiple cores for CPU-intensive tasks may require significant redesign of the algorithm. This situation is unfortunate, but shared with most today's programming languages.

Acknowledgements

SWI-Prolog is a Free Software project which, by nature, profits heavily from user feedback and participation. We would like to express our gratitude to Sergey Tikhonov for his courage to test and help debug early versions of the implementation. The work reported in section 6.6.2 was performed jointly with Ashwin Srinivasan and Steve Moyle at the Oxford University Computing Laboratory. We gratefully acknowledge the Oxford Supercomputing Centre for the use of their system, and in particular Fred Youhanaie for his patient guidance and support. Anjo Anjewierden has provided extensive comments on earlier drafts of this article.

[9] http://gollem.science.uva.nl/twiki/pl/bin/view/Foreign/CSharpInterface35

[10] The terminology that describes processing units has become confusing. For a long time, we had one chip that contained one processing unit, called CPU or processor. Now, there can be multiple 'cores' on a single physical chip that act as (nearly) independent classical CPUs. The term CPU has become ambiguous and can refer either to the chip or to a single core on a chip.

Chapter 7

SWI-Prolog and the Web

About this chapter This chapter is published in Theory and Practice of Logic Programming (Wielemaker et al. 2008). Written as a journal paper that provides an overview of using SWI-Prolog for web related tasks, it has some overlap with the previous chapters. The original paper has been extended with material from (Wielemaker et al. 2007): dispatching and session management in section 7.4.2 and the entire section 7.5.

Section 7.8 was contributed by the coauthor Zhisheng Huang and section 7.9 by Lourens van der Meij, both from the VU, Amsterdam.

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.

Where 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 an external Web server, development and deployment become much easier. To support this architecture, in addition to the HTTP transfer protocol, we must support parsing, representing and generating the key Web document types such as HTML, XML and RDF.

This paper 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 benefits of using Prolog for Web related tasks is illustrated using three case studies.

7.1 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 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 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 is in our vision 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 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 paper 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 paper 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.

This paper illustrates the benefits of using Prolog for Web related tasks in three case studies: 1) SeRQL, an RDF query language for meta data management, retrieval and reasoning; 2) XDIG, an eXtended Description Logic interface, which provides ontology management and reasoning by processing DIG XML documents and communicating to external DL reasoners; and 3) A faceted browser on Semantic Web databases integrating meta-data from multiple

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

collections of art-works. This case study serves as a complete Semantic Web application serving the end-user. Part II of this thesis contains two additional examples of applying the infrastructure described in this paper: PlDoc (chapter 8), an environment to support literate programming in Prolog and ClioPatria (chapter 10), a web-server for thesaurus-based annotation and search.

This paper is organised as follows. Section 7.2 to section 7.3 describe reading, writing and representation of Web related documents. Section 7.4 describes our HTTP client and server libraries. Supporting AJAX and CSS, which form the basis of modern interactive web pages, is the subject of section 7.5. Section 7.6 describes extensions to the Prolog language that facilitate use in Web applications. Section 7.7 to section 7.9 describe the case studies.

7.2 XML and HTML documents

The core of the Web is formed by document standards and exchange protocols. This section discusses the processing of tree-structured documents transferred as SGML or XML. 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 *content*, a sequence of sub-elements 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 by 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.

In this section we discuss parsing, representing and generating SGML/XML documents. Finally (section 7.2.3) we compare our work with PiLLoW (Gras and Hermenegildo 2001).

7.2.1 Parsing and representing XML and HTML documents

The first SGML parser for SWI-Prolog was created by Anjo Anjewierden based on 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 7.1.

Below, we motivate some of the key aspects of the representation of figure 7.1.

- 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 7.6.2. At the same time SWI-Prolog stacks are limited to 128MB each on 32-bit machines. 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

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

⟨*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*⟩

Figure 7.1: SGML/XML tree representation in Prolog. The notation list-of ⟨*x*⟩ describes a Prolog list of terms of type ⟨*x*⟩.

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 stack 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 represented by a single handle to a shared atom.

- The value of an attribute that is declared in the DTD as multi-valued (e.g., `NAMES`) is 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. Attribute-values are always returned as a single atom if no type information is provided by the DTD.

- Optionally, attribute-values of type `NUMBER` or `NUMBERS` are mapped to Prolog a prolog number or list of 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. Prolog systems with bounded arithmetic may also 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.

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 call-backs on Prolog for the SGML *events*. The call-back mode can process 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 large files with a repetitive record structure in limited memory. Section 7.3.1 describes how this is used to process RDF documents.

When we decided to implement a lightweight parser the only alternative available was the SP[3] system. Lack of flexibility of the API and the size of SP (15× larger than ours) caused us to implement our own parser. Currently, there are other lightweight XML and HTML libraries available, some of which may satisfy our requirements.

Full documentation is available from the SWI-Prolog website.[4] The SWI-Prolog SGML parser has been adopted by XSB Prolog.

7.2.2 Generating Web documents

There are many approaches to generating Web pages from programs in general and Prolog in particular. Below we present some use-cases, requirements and alternatives that must be considered.

- 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 that consist 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 (Prolog), for example `format('<b>bold</b>')` or `print_html(b(bold))`. The second approach can guarantee well-formed output, but requires the programmer to learn the mapping between Prolog syntax and HTML syntax. Direct printing requires hardly any knowledge beyond the HTML syntax.

- Documents that contain a significant static part are best represented in the markup language where special constructs insert program-generated parts. A popular approach implemented by PHP[5] and ASP[6] is to add a reserved element such as ⟨*script*⟩ or use the SGML/XML *programming instruction* written as `<?...?>`. The obvious name PSP (Prolog Server Pages) is in use by various projects taking this approach.[7] An-

[3] http://www.jclark.com/sp/
[4] http://www.swi-prolog.org/packages/sgml2pl.html
[5] www.php.net
[6] www.microsoft.com
[7] 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

other approach is PWP[8] (Prolog Well-formed Pages). It is based on the principle that the input is well-formed XML that interacts with Prolog through additional attributes. Output is guaranteed to be well-formed XML. Because we did not yet encounter a real need for any of these approaches in projects, our current infrastructure does not include any of them.

- 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 because due to alternative character encodings, entity usage and different use of SGML abbreviations there are many different source-texts that represent the same tree. The `load_structure/3` predicate described in section 7.2 together with output primitives from the library `sgml_write.pl` provide this functionality. The XDIG case study described in section 7.8 follows this approach.

7.2.2.1 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 7.2 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. In our opinion, the result is not optimal due to the unnatural order of statements and the introduction of potentially many extra variables as illustrated in figure 7.2. In this figure we first generate a small table using `mkthumbnail/3`, which is then inserted at the right location into the skeleton page using the variable *ThumbNail*. As the number of partial DOM structures that must be created grows, this style of programming quickly becomes unreadable.

We decided to design a two-step process. The first step is formed by the DCG rule `html//1`,[9] which translates a Prolog term 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. Generated from a Prolog term with the same nesting as the target HTML document, consistent opening and closing of elements is guaranteed. In addition to variable substitution which is provided by Prolog we allow calling rules. Rules are invoked by a term *Rule* embedded in the argument of `html//1`. Figure 7.3 illustrates our approach, producing the same document as figure 7.2 in a more

[8] `http://www.cs.otago.ac.nz/staffpriv/ok/pwp.pl`

[9] The notation ⟨*name*⟩//⟨*arity*⟩ refers to the grammar rule ⟨*name*⟩ with the given ⟨*arity*⟩, and consequently the predicate ⟨*name*⟩ with arity ⟨*arity*⟩+2.

```
    ...,
    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))
                 ])
```

Figure 7.2: Building a complex DOM tree from smaller components by using Prolog variables (*ThumbNail* in this figure).

readable fashion. Any reusable part of the page generation can be translated into a DCG rule. Using the *Rule* syntax it is clear which parts of the argument of `html//1` is directly translated into HTML elements and which part is expanded in a rule.

```
    ...,
    html([ h1('Photo gallery'),
           \thumbnail(URL, Caption)
         ]).
```

```
thumbnail(URL, Caption) -->
    html(table([ tr(td([halign=center], img([src=URL],[]))),
                 tr(td([halign=center], Caption))
               ]).
```

Figure 7.3: Defining reusable fragments (thumbnail//2) using library `html_write.pl`

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 7.4.

⟨*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*⟩

Figure 7.4: The `html//1` argument specification

7.2.3 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, a technique we described in the second paragraph of section 7.2.2.1. Frequently used HTML constructs are supported using reserved terms using dedicated processing. This realises a form of macro expansion using a predefined and fixed set of macros. We use DCGs and the *Rule* construct, which makes it evident 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, etcetera. 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 7.5. Having a canonical DOM representation greatly simplifies processing parsed HTML documents.

7.3 RDF documents

Where the datamodel of both HTML and XML is a tree-structure with attributes, the datamodel of the Semantic Web (SW) language RDF[10] consists of {*Subject*, *Predicate*, *Object*} *triples*. Both *Subject* and *Predicate* are URIs.[11] *Object* is either a URI or a *Literal*. As the

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

[11] URI: *Uniform Resource Identifier* is like a URL, but need not refer to an existing resource on the Web.

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

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

Figure 7.5: 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

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 out-going edges. Figure 7.6 illustrates this.

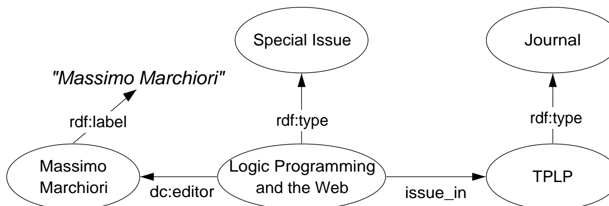


Figure 7.6: Sample RDF graph. Ellipses are vertices representing URIs. Quoted text is a literal. Edges are labelled with URIs.

A number of languages are layered on top of the RDF triple model. RDFS provides a frame-based representation. The OWL-dialects[12] provide three increasingly complex Web ontology languages. SWRL[13] 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 7.2. A triple-

[12] http://www.w3.org/2004/OWL/
[13] http://www.w3.org/Submission/SWRL/

or graph-based structure is the most natural choice for representating an RDF document in Prolog. The nodes of this graph are formed by URIs and literals. Because a URI is a string and the only operation defined on URIs by SW languages is testing equivalence, a Prolog atom is the most obvious candidate to represent a URI. One may 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. Writing the often long URIs as a quoted atom in Prolog source code harms readability and complicates changing namespaces. Therefore, the RDF library comes with a compile-time rewrite mechanism based on `goal_expansion/2` that allows for writing resources in Prolog source as ⟨*ns*⟩:⟨*local*⟩. Literals are expressed as `literal`(*Value*), where *Value* represents the value of the literal. The full Prolog representation of RDF elements is given in figure 7.7.

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

Figure 7.7: RDF types in Prolog.

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.

7.3.1 Input and output of RDF documents

The RDF/XML parser is realised as a Prolog library on top of the XML parser described in section 7.2. Similar to the XML parser it has two interfaces. 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 call-back/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 7.8 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`.

```
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).
```

Figure 7.8: Loading triples using `process_rdf/3`

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 7.9 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 the bnode is unique.

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

Figure 7.9: Blank node to express the compound dimension property

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 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 chapter 3 (Wielemaker et al. 2003b).

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 7.3.2 and writes a database directly to a file or 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 is also considerably slower than a direct write.

7.3.2 Storing and indexing RDF triples

If we collect RDF triples in a database we must provided an API to query the RDF graph. Obviously, the natural primary API to query an RDF graph is a pure non-deterministic predicate `rdf`(*?S,?P,?O*). Considering that RDF graphs tend to be large (see below), indexing the RDF database is crucial for good performance. Table 7.1 illustrates the calling pattern from a real-world application counting 4 million triples. When designing the implementation `rdf`(*?S,?P,?O*) we can exploit the restricted set of Prolog datatypes used in RDF graphs as described by figure 7.7. The RDF store was developed in the context of projects (see section 9.3) which formulated the following requirements.

- Upto at least 10 million triples on 32-bit hardware.[14]
- Fast graph traversal using any instantiation pattern.
- Case-insensitive search on literals.
- Prefix search on literals for completion in the User Interface.
- Searching for words that appear in literals.
- Multi-threaded access allowing for concurrent readers.
- 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.

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. Each cloud of connected properties is equipped with a reachability matrix. See section 3.4.1.1 for details.

[14] our current aim is 300 million on 64-bit with 64 Gb memory

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

Table 7.1: Call-statistics on a real-world system

- Literals are kept in an AVL tree, sorted case-insensitive and case-preserving (e.g., AaBb...). Numeric literals preceed all non-numeric and are kept sorted on their numeric value. Storing literals in a separate sorted table avoids the need to store duplicates and 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 metaphone (Philips 2000) based on `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 avoids both duplication of data and table-lookup. We consider this 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 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 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. 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 excep-

tion 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 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 chapter 3 (Wielemaker et al. 2003b) and loads approximately 20 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*.

7.3.3 Reasoning with RDF documents

We have identified two approaches for defining an API that supports reasoning based on Semantic Web languages such as RDFS and OWL. Both languages define an *abstract syntax* that defines their semantics in terms of conceptual entities on which the language is based. Both languages are also defined by triples that can be deduced from the set of explicitely provided triples, the *deductive closure* of the explicitely provided triples under the language. The extra triples that can be derived are called *entailed* triples. Each of these views on the language can be used to define an API:

- *Abstract syntax based* API
  The abstract syntax introduces concepts that form the basis of the language, such as *class*, *individual* or *restriction*. If this approach is applied to RDFS, the API provides predicates such as rdfs_individual_of(*?Resource, ?Class*) and rdfs_subclass_of(*?Sub, ?Super*). The SWI-Prolog library rdfs.pl and the ClioPatria (chapter 10) module owl.pl provide an API based on the abstract syntax.

- *Entailment reasoning based* API
  Semantic web query languages such as SeRQL provide access to the deductions by querying the full deductive closure instead of only the explicitely provided RDF state-

ments. Because the deductive closure is also a set of triples, this technique requires no additional API.

We use this technique in our implementation of the SeRQL (and SPARQL) query languages, as illustrated in figure 7.19. In this scenario the module `rdfs` exports an alternative definition of `rdf/3` that is true for any triple in the deductive closure. The implementation uses backward reasoning.

Figure 7.10 illustrates the two approaches. *Triples* lists the explicit triples. Both APIs return the explicit fact that *mary* is a *woman* as well as the derived (using the semantics of RDFS) fact that *mary* is a *human*. Both interfaces provide the same semantics. Both interfaces can be implemented using backward reasoning or forward reasoning. If the entailment interface is implemented as a backward reasoner, it needs to use different rulesets depending on the RDF predicate (second argument of the triple). Implemented as a forward reasoner, the derived triples are added to the database. This simplifies querying, but makes it harder to have multiple reasoners available to the application programmer at the same time. An API based on the abstract syntax and using backward reasoning can easily allow for multiple reasoners in the same application and makes it explicit what type of reasoning is used. This adds to the conceptual clarity of programs that use this API. An API based on entailment reasoning can switch between reasoners for the whole application without any change to the application code.

Triples	*Entailment* API	*Abstract syntax* API
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 .	false.	false.

Figure 7.10: Different APIs for RDFS

7.4 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 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.

Adding 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-using an existing application or

library by providing an interface for it. We compare our work with PiLLoW (Gras and Hermenegildo 2001) and the ECLiPSe HTTP services (Leth et al. 1996).

Given a basic TCP/IP socket library, writing an HTTP client is trivial (our client counts just 505 lines of code). Both PiLLoW and ECLiPSe include a client written in Prolog. The choice between embedding Prolog in an existing server framework and providing a pure Prolog-based server implementation is more complicated:

- The server is much more complex, which implies there is more to gain by re-using external code. Initially, the core server library counted 1,784 lines; the current server architecture counts over 9,000 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]

- Servers almost 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 because it blocks (some) malformed requests.

Despite these observations, we consider, like the ECLiPSe team, a pure Prolog-based server worthwhile. As argued in section 7.6.1, many Prolog Web applications profit from using state stored in the server. Large resources such as WordNet (Miller 1995) cause long startup times. In such cases the use of CGI (Common Gateway Interface) is not appropriate as a new copy of the application is started 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 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 7.11. The modules in this figure are described in the subsequent sections.

7.4.1 HTTP client libraries

We support two clients. The first (`http_open.pl`) is a lightweight client that only supports the HTTP GET method by means of `http_open`(+URL, *-Stream*, *+Options*). *Options*

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

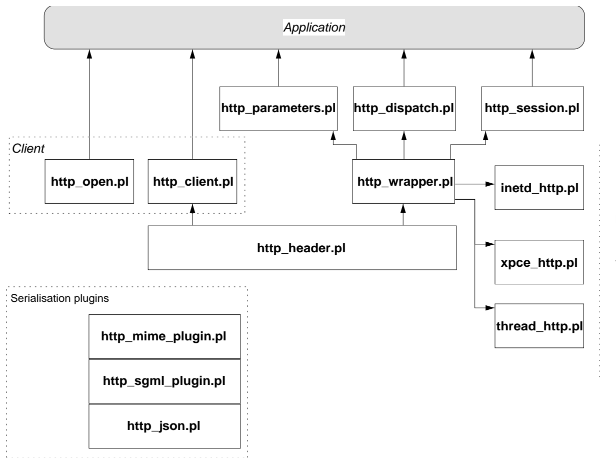


Figure 7.11: Module dependencies of the HTTP library. The modules are described in section 7.4.1 and section 7.4.2.

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 not-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 plugin 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 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 7.12 shows the code for fetching a URL and parsing the returned HTML document it into a Prolog term as described in section 7.2.

Both the PiLLoW and ECLiPSe approach return the document's content as a string. Using an intermediate string is often a waste of memory resources and limits the maximum size of documents that can be processed. In contrast, our interface is stream-based (http_open/3). The `http_get/3` and `http_post/4` interfaces allows for plugin-based processing of the input stream. Stream-based processing avoids potentially large intermediate data structures 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']), ...
```

Figure 7.12: Fetching an HTML document

7.4.2 The HTTP server library

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-strategy of the CGI protocol, where the handler writes a page consisting of an HTTP header followed by the document content. Figure 7.13 provides a simple example that returns 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`.

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

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

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

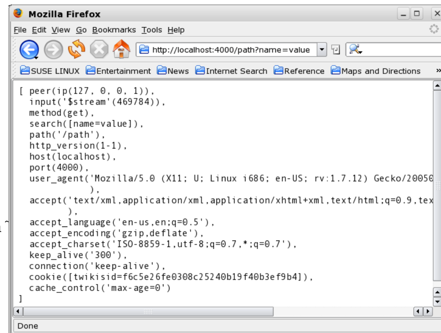


Figure 7.13: A simple HTTP server. The right window shows the client and the format of the parsed request.

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.

7.4.2.1 The HTTP dispatching code

The core HTTP library handles all requests through a single predicate specified by `http_server/2`. Normally this predicate is defined 'multifile' to split the source of the server over multiple files. This approach proved inadequate for a larger server with multiple developers for the following reasons:

- There is no way to distinguish between non-existence of an HTTP location and failure of the predicate due to a programming error. This is an omission in itself, but with a larger project and multiple developers it becomes more serious.
- There is no easy way to tell where the specific clause is that handles an HTTP location.
- As the order of clauses in a multi-file predicate that come from different files is ill defined, it is not easy to reliably redefine the sevice behind a given HTTP location. Redefinition is desirable for re-use as well as for experiments during development.

To overcome these limitations we introduced a new library `http_dispatch.pl` that defines the directive :- `http_handler`(*Location, Predicate, Options*). The directive is handled by `term_expansion/2` and is mapped to a multi-file predicate. This predicate in turn is used to build a Prolog term stored in a global variable that provides fast search for locations. Modifications to the multi-file predicate cause recomputation of the Prolog term on the next HTTP request. *Options* can be used to specify access rights, a priority to allow overruling existing definitions and assignment of the request to a new thread (*spawn*). Typically, each location is handled by a dedicated predicate. Based on the handler definitions, we can easily distinguish failure from non-existence as well as find, edit and debug the predicate implementing an HTTP location.

7.4.2.2 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 7.14 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.

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

Figure 7.14: Fetching HTTP form data

7.4.2.3 Session management

The library `http_session.pl` provides session management over the otherwise 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 7.15 illustrates the idea.

7.4.2.4 Evaluation

The presented server infrastructure is currently used by many internal and external projects. Coding a server is 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 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 7.2.

7.5 Supporting AJAX: JSON and CSS

Recent web-technology is moving towards extensive use of CSS (Cascading Style Sheets) and JavaScript. The use of JavaScript has evolved from short code snippets adding visual

```
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).
```

Figure 7.15: 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. We omitted error handling for readability of the example.

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

Table 7.2: HTTP performance executing a trivial query 10,000 times. Times are in seconds. Localhost, dual AMD 1600+ running SuSE Linux 10.0

effects through small libraries for presenting menus, etc. to extensive *widget* libraries such as YUI[16]. Considering the classical three-tier web application model (storage, application logic and presentation), the presentation tier is gradually migrated from the server towards the client. Typical examples are Google Maps and Google Calendar. Modern web *applications* consist of JavaScript libraries with some glue code that accesses one or more web-servers through an API that provides the content represented using the JSON serialisation format.[17].

This is an attractive development for deploying Prolog as a web-server. Although we

[16] `http://developer.yahoo.com/yui/`
[17] `http://www.json.org/`

have seen that Prolog can be deployed in any of the three tiers, its support for the presentation layer is weak due to the lack of ready-to-use resources. If the presentation layer is moved to the client we can reuse the presentation resources from the JavaScript community.

An AJAX-based web application is an HTML page that links in JavaScript widget libraries, other JavaScript web applications, application specific libraries and glue code as well as a number of CSS files, some of which belong to the loaded libraries while others provide the customisation. This design requires special attention in two areas. First, using a naive approach with declarations that load the required resources is hard to maintain and breaks our DCG-based reusability model because the requirements must be specified in the HTML ⟨*head*⟩ element. Second, most AJAX libraries use JSON (JavaScript Object Notation) as serialisation format for data. The subsequent two sections describe how we deal with each of these.

7.5.1 Producing HTML head material

Modern HTML+CSS and AJAX-based web-pages require links to scripts and style files in the HTML ⟨*head*⟩ element. XHTML documents that use RDFa (Adida and Birbeck 2007) or other embedded XML that requires additional namespaces, require XML namespace declarations, preferably also in the HTML ⟨*head*⟩. One of the advantages of HTML generation as described is that pages can be composed in a modular way by calling the HTML-generating DCG rules. This modularity can only be maintained if the DCG rule produces both the HTML and causes the HTML ⟨*head*⟩ to be expanded with the CSS, JavaScript and XML namespaces on which the generated HTML depends.

We resolved this issue by introducing a *mail* system into the HTML generator. The grammar rule `html_receive`(+*MailBox, :Goal*) opens a mailbox at the place in the HTML token sequence where it appears. The grammar rule `html_post`(+*MailBox, +Content*) posts HTML content embedded with calls to other grammar rules to *MailBox*. A post processing phase collects all posts to a mailbox into a list and runs *Goal* on list to produce the final set of HTML tokens. This can be used for a variety of tasks where rules need to emit HTML at certain places in the document. Figure 7.16 gives an example dealing with footnotes at the bottom of a page.

The mail system is also used by the library `html_head.pl` to deal with JavaScript and CSS resources that are needed by an HTML page. The library provides a mailbox called `head`. The directive `html_resource`(+*Resource, +Attributes*) allows for making declarations on dependencies between resources such as JavaScript and CSS files. The rule `html_requires`(+*Resource*) specifies that we need a particular CSS or JavaScript file, as well as all other resource files that are needed by that file according to the `html_resource/2` declarations. The `html_receive//2` emit wrapper that is inserted by `html_head.pl` performs the following steps:

1. Remove all duplicates from the requested resources.
2. Add implied requirements that are not yet requested.

```
:- use_module(library(http/html_write)).

footnote_demo :-
        reply_html_page(
                title('Footnote demo'),
                body([ \generate_body,
                       \html_receive(footnotes, emit_footnotes)
                     ])).

:- dynamic current_fn/1.

generate_body -->
        { assert(current_fn(1)) },
        html([ h1('Footnote demo'),
               p([ 'We use JavaScript',
                   \footnote('Despite the name, JavaScript is \
                             essentially unrelated to the Java \
                             programming language.'), ' for ...'
                 ])
             ]).


footnote(Content) -->
        { retract(current_fn(I)),
          I2 is I + 1,
          assert(current_fn(I2))
        },
        html(sup(class(footnote_ref), a(href('#fn'+I2), I))),
        html_post(footnotes, \emit_footnote(I, Content)).

%%      emit_footnotes(+List)// is det.
%
%       Emit the footnotes,  Called delayed from html_receive/2.

emit_footnotes([]) -->
        [].                             % no footnotes
emit_footnotes(List) -->
        html(div(class(footnotes),
                 List)).

emit_footnote(I, Content) -->
        html(div(class(footnote),
                 [ span(class(footnote_index),
                        a(name(fn+I), I)),
                   \html(Content)
                 ])).
```

Figure 7.16: Using the HTML mail system to generate footnotes

3. Try to apply *aggregate* scripts (see below).
4. Do a topological sort based on the dependencies, placing all requirements before the scripts that require them.
5. Emit HTML code for the sorted final set of resources.

Step *3* was inspired by the YUI framework which, in addition to the core JavaScript files, provides a number of *aggregate* files that combine a number of associated script files into a single file without redundant white space and comments to reduce HTTP traffic and enhance loading performance. If more than half of the scripts in a defined aggregate are required, the scripts are replaced by the aggregate. The 50% proportion is suggested in the YUI documentation, but not validated. The current implementation uses manually created and declared aggregates. The library has all information to create and maintain aggregates automatically based on usage patterns. This would provide maximal freedom distributing JavaScript and CSS over files for maximal modularity and conceptual clarity with optimal performance.

Figure 7.17 shows a rule from ClioPatria (chapter 10, Wielemaker et al. 2008). The rule is a reusable definition for displaying a single result found by the ClioPatria search engine. Visualisation of a single result is defined in CSS. Calling this rule generates the required HTML, and at the same time ensures that the CSS declarations (`path.css`) are loaded from the HTML ⟨*head*⟩. This mechanism provides simple reusability of rules that produce HTML fragments together with the CSS and JavaScript resources that provide the styling and interactive behaviour.

```
result_with_path(Result, Path) -->
        html_requires(css('path.css')),
        html(span(class(result),
                  [ Result,
                    span(class(path), \html_path(Path))
                  ])).
```

Figure 7.17: A single rules produces HTML and ensures that the required CSS is loaded.

7.5.2 Representing and converting between JSON and Prolog

JSON is a simple format that defines *objects* as a list of name/value pairs, arrays as an unbounded sequence of values and a few primitive datatypes: strings, numbers, boolean (`true` and `false`) and the constant `null`. For example, a point in a two-dimensional plane can be represented in JSON as `{"x":10, "y":20}`. If explicit typing is desired, one might use `{"type":"point", "x":10, "y":20}`. The JSON syntax is a subset of the SWI-Prolog syntax,[18] but the resulting term is very unnatural from Prolog's perspective: attribute

[18] Not of the ISO Prolog syntax because JSON is UNICODE (UTF-8) based and allows for the `\uXXXX` syntax to express UNICODE code points.

names map to strings instead of atoms, wasting space and making lookup unnecessarily slow. The `{...}` notation is for Prolog's grammar rules and unnatural to process. Instead, the natural Prolog representation of a point object is a term `point(10,20)`.

This section describes how JSON can be integrated into Prolog. To facilitate JSON handling from Prolog we created two libraries. The first, `json.pl` reads and writes JSON terms from/to a Prolog stream in an unambiguous format that covers the full JSON specification. The generic format is defined as in figure 7.18. For example, the point object above is represented as `json([x=10,y=20])`.

Object	::=	json([]) | json([Pair, ...])
Pair	::=	Name = Value
Name	::=	⟨*atom*⟩
Array	::=	[] | [Value, ...]
Value	::=	⟨*atom*⟩ | ⟨*number*⟩ | Object | Array
	|	@(true) | @(false) | @(null)

Figure 7.18: Unambiguous representation of a JSON term, using the same BNF structure as the JSON specification.

The second, `json_convert.pl` supports type checking and conversion to a 'natural' Prolog term. Complex types are defined by the name/arity of a Prolog compound term. Primitive types are `atom`, `integer`, etc. JSON interface types are declared using the directive `json_object`(+*Specification*), which we introduce through the example below. The example defines two types. The first specifies the type `person/2` with arguments *name* and *address*. In turn, *address* is defined by the street, house-number and city, all of which are covered by Prolog primitive types.

```
:- json_object
        person(name:atom, address:address),
        address(street:atom, number:integer, city:atom).
```

With these declarations, `prolog_to_json`(+*Prolog*, -JSON) converts a term such as `person`(*mary, address(kerkstraat, 42, amsterdam)*) into the unambiguous JSON representation `json`(*[mary, json([kerkstraat, 42, amsterdam]*). The inverse operation is performed by `json_to_prolog`(+JSON, -*Prolog*). The library enforces the type declarations.

7.6 Enabling extensions to the Prolog language

SWI-Prolog has been developed in the context of projects, many of which focused 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.

7.6.1 Multi-threading

Concurrency is necessary for applications for the following reasons:

- Network delays may cause communication of a single transaction to take long. 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.
- 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 (chapter 6, Wielemaker 2003a) because it resolves delays and exploiting multi-CPU hardware 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 state 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 7.3.2.

7.6.2 Atoms and UNICODE support

UNICODE[19] 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 in different scripts (e.g., Latin and Hebrew) as well as documents that contain text from multiple scripts. This feature greatly simplifies applications that must be able to deal with multiple scripts, such as web applications serving a world-wide audience. Traditional HTML applications commonly insert special symbols through entities such as the copyright (©) sign, Greek and mathematical symbols, etcetera. Using UNICODE we can represent all entity values uniformely as plain text. UTF-8, an encoding of UNICODE as a sequence of bytes, is at the heart of XML and the Semantic Web.

HTML documents can be represented using Prolog strings because Prolog integers can represent all UNICODE code-points. As we have claimed in section 7.2 however, using Prolog strings is not the most obvious choice. XML attribute names and values can contain arbitrary UNICODE characters, which requires the unnatural use of strings for these as well. If we consider RDF, IRIs can have arbitrary UNICODE characters[20] and we want to represent IRIs 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 properties below for Prolog atoms.

[19] `http://www.Unicode.org/`

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

- 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 maximum number of atoms is 2^{25} (32 million) on 32-bit hardware.
- Continuously running servers cannot allow memory leaks and therefore processing dynamic data using atoms requires atom garbage collection.

7.7 Case study — A Semantic Web Query Language

In this case-study we describe the SWI-Prolog SeRQL implementation, currently distributed as an integral part of ClioPatria chapter 10 (Wielemaker et al. 2008). SeRQL is an RDF query language developed as part of the Sesame project[21] (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 building it has contributed significantly to the development of the infrastructure. Figure 7.19 lists the main components of the server.

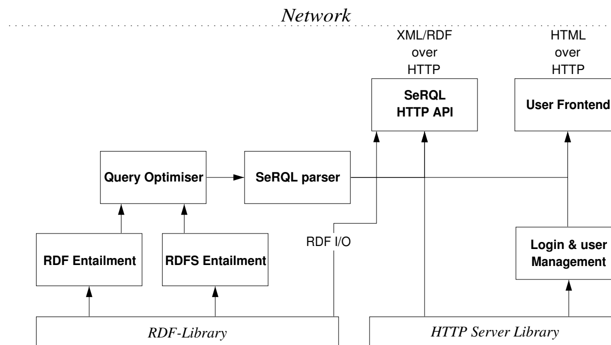


Figure 7.19: Module dependencies of the SeRQL system. Arrows denote 'imports from' relations.

The *entailment* modules are plugins that implement the entailment approach to RDF reasoning described in section 7.3.3. They implement `rdf/3` as a pure predicate, adding implicit triples to the raw triples loaded from RDF/XML documents. Figure 7.20 shows the somewhat simplified entailment module for RDF. The negations avoid duplicate results. This

[21] `http://www.openrdf.org`

is not strictly necessary, but if the test is cheap we expect that the extra cost will generally be returned in subsequent processing steps. The multifile rule serql:entailment/2 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.

```
:- 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).
```

Figure 7.20: RDF entailment module

The SeRQL parser is a DCG-based parser that translates a SeRQL query text into a compound goal calling `rdf/3` and predicates from the SeRQL runtime library which provide comparison and functions built into the SeRQL language. The resulting control-structure is passed to the query optimiser chapter 4 (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 7.21 illustrates this in a simple example.

HTTP access consists of two parts. The human-centred portal consists of HTML pages with forms to manage 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 7.2.2.1. 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 produce the results in the

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

Figure 7.21: 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).

requested output format. The HTML output format uses `html_write.pl`. The RDF/XML format uses `rdf_write_xml/2` described in section 7.3.1. 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 7.2. Using direct writing instead of techniques described in section 7.2.2.1 avoids potentially large intermediate datastructures and is not complicated given the often simple structure of the documents.

Evaluation

Development of the SeRQL server and the SWI-Prolog web libraries 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 differently. Sesame's forward reasoning makes it handle some RDFS queries much faster. Sesame does not contain a query optimiser which causes order-dependent and sometimes very long response times on conjunctions.

The power of LP where programs can be handled as data is exploited by parsing the SeRQL query into a program and optimising this 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 only depends on the standard SWI-Prolog distribution and therefore runs unmodified on all systems supporting SWI-Prolog. It 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` from the library `sgml_write.pl`.

7.8 Case study — XDIG

In section 7.7 we have discussed the case study how SWI-Prolog is used for a RDF query system, i.e., a meta-data 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 a standard for accessing DL reasoning engines called the DIG description logic interface[22] (Bechhofer et al. 2003), DIG interface for short. Many DL reasoners like 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, an eXtended DIG Description Logic interface, which has been implemented on top of the SWI-Prolog Web libraries. XDIG is a platform that provides a DIG *proxy* in which application specific transformations and extensions can be realised. 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 which corresponds to the DL concept language. Where OWL is based on the triple model described in section 7.3, DIG statements are grounded directly in XML. Figure 7.22 shows a DIG statement which defines the concept *MadCow* as a cow which eats brains, part of sheep.

```
<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>
```

Figure 7.22: a DIG statement on MadCow

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

7.8.1 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.

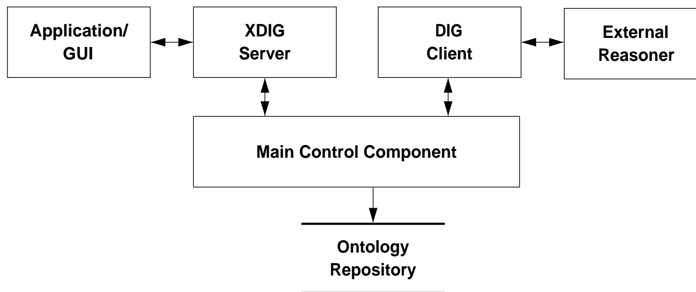


Figure 7.23: Architecture of XDIG

The general architecture of XDIG is shown in figure 7.23. It consists of the following components:

XDIG Server The XDIG server deals with requests from ontology applications. It supports our extended DIG interface: in addition to the standard DIG/DL requests, like 'tell' and 'ask' it supports requests such as changing system settings. The library `dig_server.pl` implements the XDIG protocol on top of the Prolog HTTP server described in section 7.4.2. The predicate `dig_server`(+*Request*) is called from the HTTP server to process a client's *Request* as illustrated in figure 7.13. 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 on *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 7.2. The tree model of XDIG data has been proved to be convenient for DIG data management.

Figure 7.24 shows a piece of code from the XDIG defining the predicate `direct_concept_relevant`(+DOM, *?Concept*) which 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.

```
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)
    ).
```

Figure 7.24: direct_concept_relevant checks that a concept is referenced by a DIG statement

7.8.2 Application

XDIG has been used to develop several DL reasoning services. PION is a reasoning system that deals with inconsistent ontologies[23] (Huang and Visser 2004; Huang et al. 2005), MORE is a reasoner[24] (Huang and Stuckenschmidt 2005) that supports multiple versions of the same ontology and DION is a debugger of inconsistent ontologies[25](Schlobach and Huang 2005).

[23] http://wasp.cs.vu.nl/sekt/pion
[24] http://wasp.cs.vu.nl/sekt/more
[25] http://wasp.cs.vu.nl/sekt/dion

With the support of an external DL reasoner like Racer (Haarslev and Möller 2001), DION can serve as an inconsistent ontology debugger using a bottom-up approach.

7.9 Case study — Faceted browser on Semantic Web database integrating multiple collections

In this case study we describe a pilot for the STITCH-project[26] that aims at 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 user interface for browsing the merged collections. One requirement within the pilot is to use "standard Semantic Web 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 could be split into three main tasks:

- Gathering data, i.e., collecting records of the collections and controlled vocabularies they use and transforming these into RDF.
- Establishing semantic links between the vocabularies using off-the-shelf ontology mapping tools.
- Building a prototype User Interface (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 to realise all three tasks. To illustrate our use of the SWI-Prolog Web libraries we focus on their application in the prototype user interface because it is the largest subsystem using these libraries.

7.9.1 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 user interface 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 is stored in an RDF database, which can be either an external SeRQL repository or an in-memory SWI-Prolog RDF database. The system consists 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.

[26] `http://stitch.cs.vu.nl`

7.9.1.1 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 1000 from the Koninklijke Bibliotheek.
- An RDF representation of hierarchically structured *facets*. Hierarchical structuring is defined using SKOS[27], an RDF schema to represent controlled vocabularies.
- Mappings between SKOS Concept Schemes used in the different collections.
- Portal-specific information as "*Site Configuration Objects*" (SCO), identified by URIs with properties defining what collections are part of the setup, what facets are shown, and also values for the constant text in the Web page presentation and other User Interface configuration properties. Multiple SCOs 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 7.7).

RDF database access Querying the RDF store for compound results such as a list of artworks where each artwork is enriched with information on how it must be displayed (e.g., title, thumbnail, collection) based on HTTP query arguments consists of three steps: 1) building SeRQL queries from the HTTP query arguments, 2) passing them on to the SeRQL-engine, gathering the result rows and 3) finally post-processing the output, e.g., counting elements and sorting them. Figure 7.25 shows an example of a generated SeRQL query. Finding matching records involves finding records annotated by the facet 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 7.25.

The SeRQL-query interface contains timing and debugging facilities for single queries; for flexibility it provides access to an external SeRQL server[28] for which we used Sesame[29], but also to the in-memory store of the SWI-Prolog SeRQL implementation described in section 7.7.

---

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

[28] We used the `sesame_client.pl` library that provides an interface to external SeRQL servers, packaged with the SWI-Prolog SeRQL library

[29] `http://www.openrdf.org`

```
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>
```

Figure 7.25: An example of a SeRQL query, which returns details of records matching two facet values (AnimalPieces and Paintings)

7.9.1.2 HTML-code generation

We used the SWI-Prolog `html_write.pl` library described in section 7.2.2.1 for our HTML-code generation. There are three distinct kinds of Web pages 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 (1200 lines of Prolog code of which 800 lines are DCG rules), part of which are simple list-traversing rules such as the example of Figure 7.26.

7.9.1.3 Web Server

The Web server is implemented using the HTTP server library described in section 7.4.2. The Web server component itself is small. It follows the skeleton code described in Figure 7.13. 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 7.9.1.1, is returned by the RDF-interaction component based on the URL root. It is passed on to the HTML-code generation component which generates Web content.

7.9.2 Evaluation

This case study shows that SWI-Prolog is effective for building applications in the context of the Semantic Web. 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 software has been tested to install and run transparently both on Linux and on Microsoft Windows.

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

Figure 7.26: Part of the html code generation for displaying all the images of a query result in an HTML table

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 Semantic Web 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 setups made us decide 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).

7.10 Conclusion

We have presented an overview of the libraries and Prolog language extensions we have implemented and which we provide to the Prolog community as Open Source resources. The following components have been identified as vital for using Prolog in (semantic) web related tasks:

- HTTP *client and server support*
  Instead of using proprietary protocols between Prolog and other components in a web server architecture, we propose the use of the web HTTP protocol. Based on HTTP, Prolog can be deployed anywhere in the server architecture.

- *Web document support*
  In addition to the transfer protocol, standardisation of document formats is what makes the web work. We propose a representation of XML/SGML, HTML, RDF and JSON documents as Prolog terms. Standardisation of these Prolog terms is a first requirement to enable portable libraries processing web documents. We presented a modular and extensible mechanism to generate HTML documents. The infrastructure automatically includes CSS and JavaScript on which the document depends.

- RDF *storage and querying*
  RDF is naturally represented by a Prolog predicate `rdf`(*Subject, Predicate, Object*). We implemented `rdf/3` as a foreign extension to Prolog because this allows us to optimise the representation by exploiting the restrictions and use patterns provided by the Semantic Web languages.

Proper support of the above components is not possible without extending Prolog with at least the following features:

- *Threading*
  Availability of multi-CPU hardware and requirements for responsive and scalable web services demands concurrency. This is particularly true when representing large amounts of knowledge as Prolog rules. Multiple threads provide scalable reasoning with this knowledge without duplication.

- *Atom handling*
  Atoms are the obvious representation for IRIs in RDF as well as element and attribute names in XML. This requires UNICODE atoms as well and large numbers of atoms. In addition, atoms can be used to represent attribute values and (textual) content in XML, which requires atoms of unbounded length and atom garbage collection.

Considering the three tier model for implementing web services, the middleware, dealing with the application logic, is the most obvious tier for exploiting Prolog. Flexibility provided by using HTTP for communication does not limit Prolog to the middleware. In the case studies presented in this paper we have seen Prolog active as storage component (section 7.7), middleware (section 7.8) and in the presentation tier (section 7.9). In chapter 10, the Prolog server (ClioPatria) initially handled all three tiers. Later, part of the presentation was moved to JavaScript running in the web-browser to improve dynamic behaviour and exploit reusable JavaScript libraries. ClioPatria uses all infrastructure described in this chapter.

Figure 7.27 shows the ‘Semantic Web layer cake’ that illustrates the architecture of the Semantic Web and which parts of the cake are covered by the infrastructure described in this chapter.

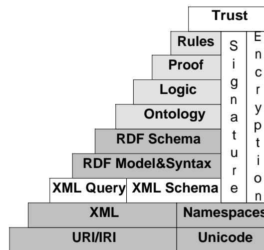


Figure 7.27: The Semantic Web layer cake by Tim Burners Lee. The dark grey boxes are fully covered. For light grey areas, ontologies are covered by standard languages but our coverage of the standard is partial while the other layers are not covered by clear standards, but Prolog is a promising language to explore them.

Development in the near future is expected to concentrate on Semantic Web reasoning, such as the translation of SWRL rules to logic programs. Such translations will benefit from tabling to realise 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 or PWP-like (see section 7.2.2) page-generation facilities.

Acknowledgements

Preparing this paper as well as maturing RDF and HTTP support was supported by the MultimediaN[30] project funded through the BSIK programme of the Dutch Government. Other projects that provided a major contribution to this research include the European projects IMAT (ESPRIT 29175) and GRASP (AD 1008) and the ICES-KIS project “Multimedia Information Analysis” funded by the Dutch government. 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 Organisation 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.

[30] `www.multimedian.nl`

Part II

Knowledge-intensive applications

Chapter 8

PlDoc: Wiki style literate Programming for Prolog

About this chapter This chapter has been presented at 17th Workshop on Logic-based methods in Programming Environments, Porto (WLPE 2007, Wielemaker and Anjewierden 2007). It is a concrete example of using the infrastructure described in part I for realising a web application. At the same time it is part of the SWI-Prolog development tools (Wielemaker 2003b). The web front end can be merged into web applications, creating a self documenting web server. This feature has been used in ClioPatria, described in chapter 10.

Notes on a LaTeX backend in the future work section of the original paper have been replaced by a new section 8.7. Section 8.5 was contributed by the co-author Anjo Anjewierden, University of Twente.

Abstract This document introduces PlDoc, a literate programming system for Prolog. Starting point for PlDoc was minimal distraction from the programming task and maximal immediate reward, attempting to seduce the programmer to use the system. Minimal distraction is achieved using structured comments that are as closely as possible related to common Prolog documentation practices. Immediate reward is provided by a web interface powered from the Prolog development environment that integrates searching and browsing application and system documentation. When accessed from *localhost*, it is possible to go from documentation shown in a browser to the source code displayed in the user's editor of choice.

8.1 Introduction

Combining source and documentation in the same file, generally named *literate programming*, is an old idea. Classical examples are the TeX source (Knuth 1984) and the self documenting editor GNU-Emacs (Stallman 1981). Where the aim of the TeX source is first

of all documenting the program, for GNU-Emacs the aim is to support primarily the end user.

There is an overwhelming amount of articles on literate programming, most of which describe an implementation or qualitative experience using a literate programming system (Ramsey and Marceau 1991). Shum and Cook (1994) describe a controlled experiment on the effect of literate programming in education. Using literate programming produces more comments in general. More convincingly, it produced 'how documentation' and examples where, without literate programming, no examples were produced at all. Nevertheless, the subjects participating in the above experiment considered literate programming using the AOPS (Shum and Cook 1993) system confusing and harming debugging. This could have been caused by AOPS which, like TEX's `weave` and `tangle`, uses an additional preprocessing step to generate the documentation and a valid program for the compiler.

Recent developments in programming environments and methodologies make a case for re-introducing literate programming (Pieterse et al. 2004). The success of systems such as JavaDoc[1] and Doxygen (van Heesch 2007) is evident. Both systems are based on *structured comments* in the source code. Structured comments use the comment syntax of the programming language (e.g., `%...\n` or `/*...*/` in Prolog) and define additional syntax that make the comment recognisable as being 'structured' (e.g., start `/**` in JavaDoc) and provides layout directives (e.g., HTML in JavaDoc). This approach makes the literate programming document a valid document for the programming language. Using a source document that is valid for the programming language ensures smooth integration with any tool designed for the language.

Note that these developments are different from what Knuth intended: "The literate programmer can be regarded as an essayist that explains the solution to a human by crisply defining the components and delicately weaving them together into a complete artistic creation" (Knuth 1984). Embedding documentation in source code comments merely produces an API *Reference Manual*.

In the Prolog world we see lpdoc (Hermenegildo 2000), documentation support in the Logtalk (Moura 2003) language and the ECLiPSe Document Generation Tools[2] system. All these approaches use Prolog *directives* making additional statements about the code that feed the documentation system. In 2006 a commercial user in the UK whose products are developed using a range of technologies (including C++ using Doxygen for documentation) approached us to come up with an alternative literate programming system for Prolog, aiming at a documentation system as non-intrusive as possible to their programmers' current practice in the hope this will improve the documentation standards for their Prolog-based work.

This document is structured as follows. First we outline the different options available to a literate programming environment and motivate our choices. Next we introduce PlDoc using and example, followed by a more detailed overview of the system. Section 8.5 tells the story of introducing PlDoc in a large open source program, after which we compare our

[1] `http://java.sun.com/j2se/javadoc/`
[2] `http://eclipse.crosscoreop.com/doc/userman/umsroot088.html`

work to related projects in section 8.6.

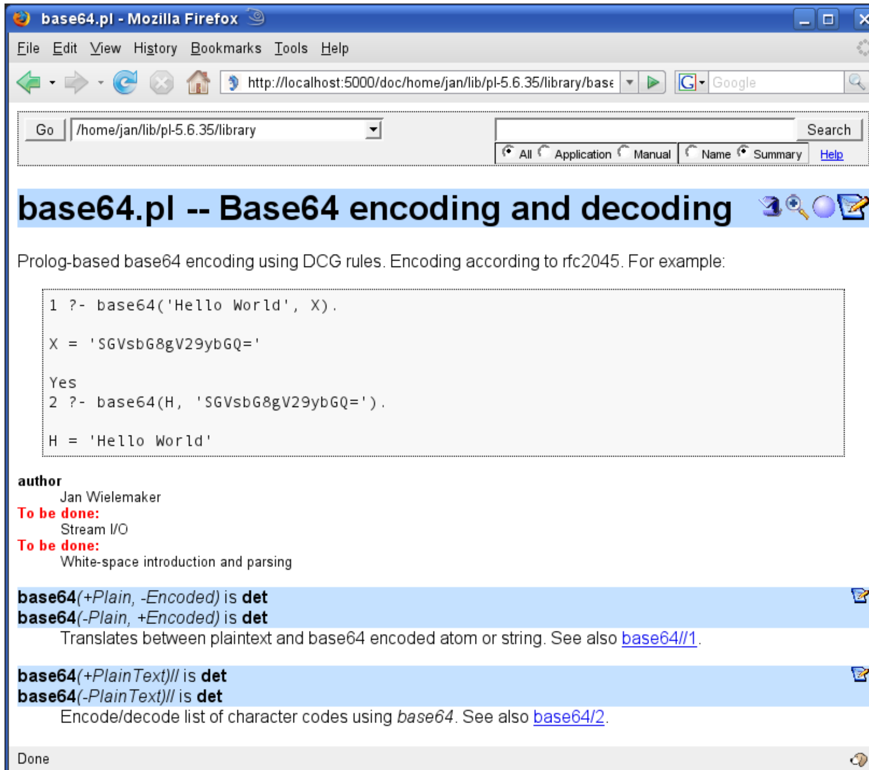


Figure 8.1: Documentation of library base64.pl. The example is described in section 8.3. The syntax ⟨*name*⟩(⟨*arg*⟩...)// and ⟨*name*⟩//⟨*arity*⟩ define and reference grammar rules (DCG). Accessed from 'localhost', PlDoc provides edit (rightmost) and and reload (leftmost) buttons that integrate it with the development environment.

8.2 An attractive literate programming environment

Most programmers do not like documenting code and Prolog programmers are definitely no exception to this rule. Most can only be 'persuaded' by the organisation they work for, using a grading system in education that values documentation (Shum and Cook 1994) or by the desire to produce code that is accepted in the Open Source community. In our view, we

must seduce the programmer to produce API documentation and internal documentation by creating a rewarding environment. In this section we present the available dimensions and motivate our primary choices in this design-space based on our aim for minimal impact and reward.

For the design of a literate programming system we must make decisions on the input: the language in which we write the documentation and how this language is merged with the programming language (Prolog) into a single source file. Traditionally the documentation language was TEX-based (including Texinfo). Recent systems (e.g., JavaDoc) also use HTML. In Prolog, we have two options for merging documentation in the Prolog text such that the combined text is a valid Prolog document. The first is using Prolog comments and the second is to write the documentation in *directives* and define (possibly dummy) predicates that handle these directives.

In addition we have to make a decision on the output format. In backend systems we see a shift from TEX (paper) and plain-text (online) formats towards HTML, XML+XSLT and (X)HTML+CSS which are widely supported in todays development environments. Web documents provide both comfortable online browsing and reasonable quality printing.

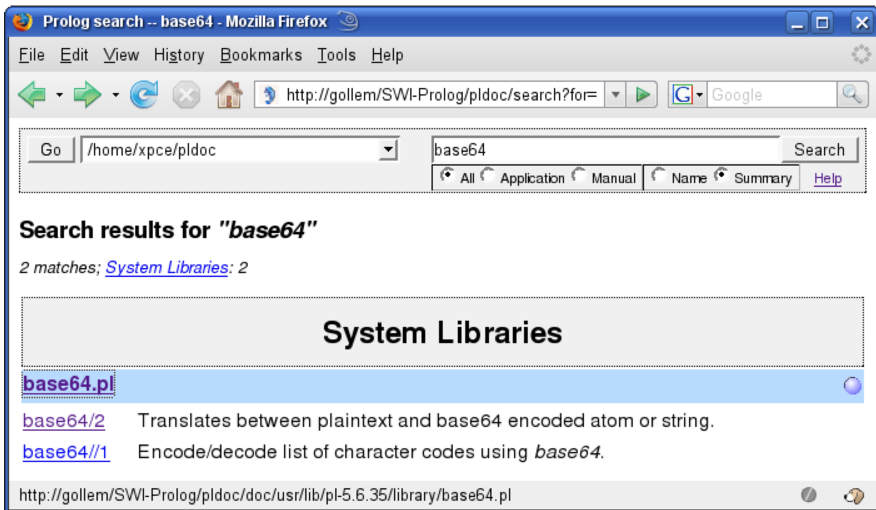


Figure 8.2: Searching for “base64”

In this design space we aim at a system with little overhead for the programmer and a short learning curve that immediately rewards the programmer with a better overview and integrated web-based search over both the application documentation and the Prolog manual.

Minimal impact Minimising the impact on the task of the programmer is important. Programming itself is a demanding task and it is important to reduce the mental load to the minimum, only keeping that what is essential for the result. Whereas object oriented languages can extract some basics from the class hierarchy and type system, there is little API information that can be extracted automatically from a Prolog program, especially if it does not use modules. Most information for an API reference must be provided explicitly and additionally to the program.

Minimising impact as well as maximising portability made us decide against the approach of lpdoc, ECLiPSe and Logtalk (see section 8.6) which provide the documentation in language extensions by means of directives and in favour of using structured comments based on layout and structuring conventions around in the Prolog community. Structured comments start with `%%` (similar to Postscript document structuring comments) or `/**` and use simple plain text markup that has been in use for a long time in email, usenet as well as for commenting source code. Wikis (Leuf and Cunningham 2001) have introduced similar markup for editing web-pages online. The popularity of wikis as well as the success in translating simple text markup into proper layout suggest this as a promising approach.

Immediate and maximal reward to the programmer A documentation system is likely to be more appreciated by the programmer if it provides immediate benefits during development instead of merely satisfying long term documentation needs for the organisation. If we can provide fully up-to-date searchable documentation that is linked directly to the underlying source code, the programmer is likely to appreciate the documentation system for browsing the system under development. This has been achieved by adding the documentation system as an optional library to the Prolog development environment. With PlDoc loaded into Prolog, the compiler processes the structured comments and maintains a Prolog documentation database as described in section 8.8.1. This database is made available to the developer through a web server running in a separate thread (section 8.4.2). The SWI-Prolog `make/0` command updates the running Prolog system to the latest version of the loaded sources and updates the web site at the same time.

In addition, we merge the documentation of the loaded Prolog code with the Prolog manuals in a consistent view presented from the embedded web server. This relieves the programmer from making separate searches in the manuals and other parts of system under development.

8.3 An example

Before going into detail we show the documentation process and access for the SWI-Prolog library `base64.pl`. `base64.pl` provides a DCG rule for base64 encoding and decoding as well as a conversion predicate for atoms. Part of the library code relevant for the documentation is in figure 8.3 (described below). Figure 8.1 shows the documentation in a browser and figure 8.2 shows the search interface.

In figure 8.3, we see a number of documentation constructs in the comments:

```
/** <module> Base64 encoding and decoding

Prolog-based base64 encoding using DCG rules. Encoding according to
rfc2045. For example:

==
1 ?- base64('Hello World', X).
X = 'SGVsbG8gV29ybGQ='

2 ?- base64(H, 'SGVsbG8gV29ybGQ=').
H = 'Hello World'
==

@tbd    Stream I/O
@tbd    White-space introduction and parsing
@author Jan Wielemaker
*/

%%      base64(+Plain, -Encoded) is det.
%%      base64(-Plain, +Encoded) is det.
%
%       Translates between plaintext and base64  encoded atom
%       or string. See also base64//1.

base64(Plain, Encoded) :- ...

%%      base64(+PlainText)// is det.
%%      base64(-PlainText)// is det.
%
%       Encode/decode list of character codes using _base64_.
%       See also base64/2.

base64(Input) --> ...
```

Figure 8.3: Commented source code of library base64.pl

- The `/** <module> Title` comment introduces overall documentation of the module. Inside, the == delimited lines mark a source code block. The `@`*keyword* section provides JavaDoc inspired keywords from a fixed and well defined set described at the end of section 8.4.1.

- The `%%` comments start with one or more `%%` lines that contain the predicate name, argument names with optional mode, type and determinism information. Multiple modes and predicates can be covered by the same comment block. The predicates declarations must obey a formal syntax that is defined in figure 8.4. The formal part is followed by plain text using wiki structuring, processed using the same rules that apply to the module comment. Like JavaDoc, the first sentence of the comment body

is considered a *summary*. Keyword search processes both the formal description and the summary. This type of search has a long history, for example in the Unix `man` command.

8.4 Description of PlDoc

8.4.1 The PlDoc syntax

PlDoc processes structured comments. Structured comments are Prolog comments starting with `%%` or `/**`. The former is more in line with the Prolog tradition for commenting predicates while the second is typically used for commenting the overall module structure. The system does not enforce this. Java programmers may prefer using the second form for predicate comments as well.

Comments consist of a formal header, a wiki body and JavaDoc inspired keywords. When using `%%` style comments, the formal header ends with the first line that starts with a single `%`. Using `/**` style comments, the formal header is ended by a blank line. There are two types of formal headers. The formal header for a predicate consists of one or more *type* and *mode* declarations. The formal header that comments a module is a line holding "⟨*module*⟩ *Title*". The ⟨*tag*⟩-style syntax can be extended in future versions of PlDoc to accommodate other documentation elements (e.g., sections).

The type and mode declaration header consists of one or more Prolog terms. Each term describes a mode of a predicate. The syntax is described in figure 8.4.

<table>
<tr><td>⟨modedef⟩</td><td>::=</td><td>⟨head⟩['//'] ['is' ⟨determinism⟩]</td></tr>
<tr><td>⟨determinism⟩</td><td>::=</td><td>'det'</td></tr>
<tr><td></td><td>|</td><td>'semidet'</td></tr>
<tr><td></td><td>|</td><td>'nondet'</td></tr>
<tr><td></td><td>|</td><td>'multi'</td></tr>
<tr><td>⟨head⟩</td><td>::=</td><td>⟨functor⟩'('⟨argspec⟩ {',' ⟨argspec⟩}')'</td></tr>
<tr><td></td><td>|</td><td>⟨atom⟩</td></tr>
<tr><td>⟨argspec⟩</td><td>::=</td><td>[⟨mode⟩]⟨argname⟩[':'⟨type⟩]</td></tr>
<tr><td>⟨mode⟩</td><td>::=</td><td>'+' | '-' | '?' | ':' | '@' | '!'</td></tr>
<tr><td>⟨type⟩</td><td>::=</td><td>⟨term⟩</td></tr>
</table>

Figure 8.4: BNF for predicate header

The optional `//`-postfix indicates that ⟨*head*⟩ is a grammar rule (DCG). The *determinism* values originate from Mercury (Jeffery et al. 2000). Predicates marked as `det` must succeed exactly once and leave no choice points. The `semidet` indicator is used for predicates that either fail or succeed deterministically. The `nondet` indicator is the most general one and implies there are no constraints on the number of times the predicate succeeds and whether or not it leaves choice points on the last success. Finally, `multi` is as `nondet`, but

demands the predicate to succeed at least one time. Informally, `det` is used for deterministic transformations (e.g., arithmetic), `semidet` for tests, `nondet` and `multi` for *generators*.

The mode patterns are given in figure 8.5. Originating from DEC-10 Prolog were the *mode* indicators (+,−,?) had a formal meaning. The ISO standard (Deransart et al. 1996) adds '@', meaning "the argument shall remain unaltered". Quintus added ':', meaning the argument is module sensitive. Richard O'Keefe proposes[3] '=' for "remains unaltered" and adds '*' (ground) and '>' "thought of as output but might be nonvar".

+	Argument must be fully instantiated to a term that satisfies the type.
-	Argument must be unbound on entry.
?	Argument must be bound to a *partial term* of the indicated type. Note that a variable is a partial term for any type.
:	Argument is a meta argument. Implies +.
@	Argument is not further instantiated.
!	Argument contains a mutable structure that may be modified using `setarg/3` or `nb_setarg/3`.

Figure 8.5: Defined modes

The body of a description is given to a Prolog defined wiki parser based on Twiki[4] using extensions from the Prolog community. In addition we made the following changes.

- List indentation is not fixed, the only requirement is that all items are indented to the same column.
- Font changing commands such as `*bold*` only work if the content is a single word. In other cases we demand `*|bold text|*`. This proved necessary due to frequent use of punctuation characters in comments that make single font-switching punctuation characters too ambiguous.
- We added == around code blocks (see figure 8.3) as code blocks are frequent and not easily supported by the core Twiki syntax.
- We added automatic links for ⟨*name*⟩/⟨*arity*⟩, ⟨*name*⟩//⟨*arity*⟩, ⟨*file*⟩.pl, ⟨*file*⟩.txt (interpreted as wiki text) and image files using image extensions. If a file reference or predicate indicator is enclosed in double square brackets (e.g., **[[file.png]]**, the contents is included at this place. This mechanism can be used to include images in files or include the documentation of a predicate into a `README` file without duplicating the description (see section 8.7).
- Capitalised words appearing in the running text that match exactly one of the arguments are typeset in *italics*.

[3] `http://gollem.science.uva.nl/SWI-Prolog/mailinglist/archive/2006/q1/0267.html`
[4] `http://www.twiki.org`

- We do not process embedded HTML. One of the reasons is that we want the option for other target languages (see section 8.7). Opening up the path to unlimited use of HTML complicates this. In addition, passing <, > and & unmodified to the target HTML easily produces invalid HTML.

The '@' keyword section of a comment block is heavily based on JavaDoc. We give a summary of the changes and additions below.

- @return is dropped for obvious reasons.
- @error *Error* is added as a shorthand for @throws `error`(*Error, Context*)
- @since and @serial are not (yet) supported
- @compat is added to describe compatibility of libraries
- @copyright and @license are added
- @bug and @tbd are added for issue tracking

A full definition of the Wiki notation and keywords is in the PlDoc manual.[5].

8.4.2 Publishing the documentation

PlDoc realises a web application using the SWI-Prolog HTTP infrastructure (chapter 7, Wielemaker et al. 2008). Running in a separate thread, the normal interactive Prolog toplevel is not affected. The documentation server can also be used from an embedded Prolog system. By default access is granted to 'localhost' only and the web interface provides links to the development environment as shown in figure 8.1.

Using additional options to `doc_server`(+*Port*, +*Options*), access can be granted to a wider public. Since September 15 2006, we host a public server running the latest SWI-Prolog release with all standard libraries and packages loaded from `http://gollem.science.uva.nl/SWI-Prolog/pldoc/`. Currently (June 2008), the server handles approximately 1,000 search requests (15,000 page views) per day. This setup can also be used for an intranet that supports a development team. Running a Prolog server with all libraries that are available to the team loaded, it provides an up-to-date and searchable central documentation source.

8.4.3 IDE integration and documentation maintenance cycle

When accessed from 'localhost', PlDoc by default provides an option to edit a documented predicate. Clicking this option activates an HTTP request through JavaScript similar to AJAX (Paulson 2005), calling `edit`(+*PredicateIndicator*) on the development system. This hookable predicate locates the predicate and runs the user's editor of choice on the given location.

[5] `http://www.swi-prolog.org/packages/pldoc.html`

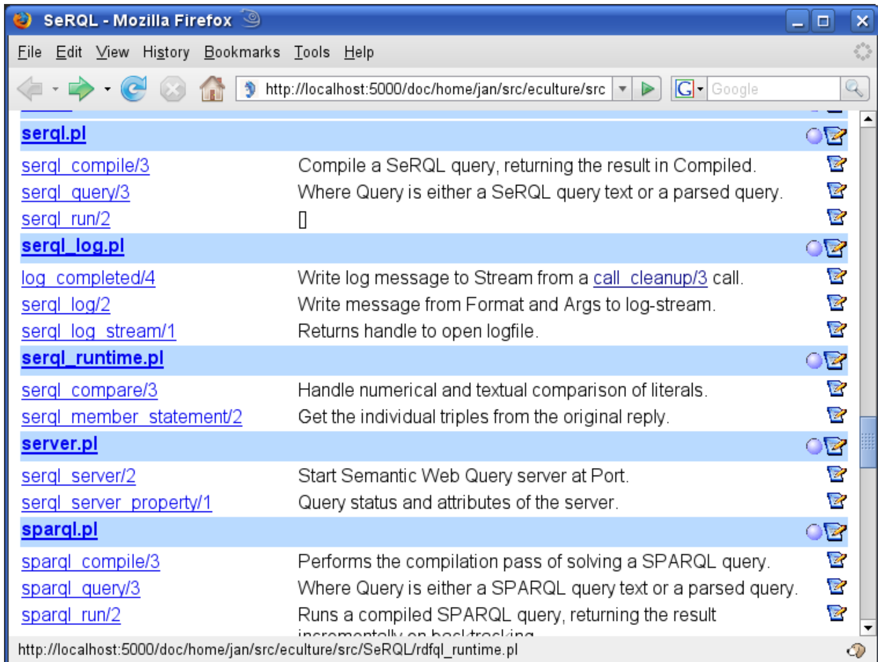


Figure 8.6: PlDoc displaying a directory index with files and their public predicates accessed from 'localhost'. Each predicate has an 'edit' button and each file a pretty print button (blue circle, see section 8.4.4)

In addition the browser interface shows a 'reload' button to run `make/0` and refreshes the current page, reflecting the edited documentation.

Initially, PlDoc is targeted to the working directory. In the directory view it displays the content of the `README` file (if present) and all Prolog files with a summary listing of the public predicates as illustrated in figure 8.6.

As a simple quality control measure PlDoc lists predicates that are exported from a module but not documented in red at the bottom of the page. See figure 8.7.

We used the above to provide elementary support through PlDoc for most of the SWI-Prolog library and package sources (approx. 80,000 lines). First we used a simple `sed` script to change the first line of a `%` comment that comments a predicate to use the `%%` notation. Next we fixed syntax errors in the formal part of the documentation header. Some of these where caused by comments that should not have been turned into structured comments. PlDoc's enforcement that argument names are proper variable names and types are proper

Prolog terms formed the biggest source of errors. Finally, directory indices and part of the individual files were reviewed, documentation was completed and fixed at some points. The enterprise is certainly not complete, but an effort of three days made a big difference in the accessibility of the libraries.

8.4.4 Presentation options

By default, PlDoc only shows public predicates when displaying a file or directory index. This can be changed using the 'zoom' button displayed with every page. Showing documentation on internal predicates proves helpful for better understanding of a module and helps finding opportunities for code reuse. Searching shows hits from both public and private predicates, where private predicates are presented in grey using a yellowish background.

Every file entry has a 'source' button that shows the source file. Structured comments are converted into HTML using the Wiki parser. The actual code is coloured based on information from the SWI-Prolog cross referencer using code shared with PceEmacs.[6] The colouring engine uses `read_term/3` with options 'subterm_positions' to get the term layout compatible to Quintus Prolog (SICS 1997) and 'comments' to get comments and their positions in the source file.

8.5 User experiences

tOKo (Anjewierden et al. 2004) is an open source tool for text analysis, ontology development and social science research (e.g., analysis of Web 2.0 documents). tOKo is written in SWI-Prolog. The user base is very diverse and ranges from Semantic Web researchers who need direct access to the underlying code for their experiments, system developers who use an HTTP interface to integrate a specific set of tOKo functionality into their systems, to social scientists who only use the interactive user interface.

The source code of tOKo, 135,000 lines (excluding dictionaries) distributed over 321 modules provides access to dictionaries, the internal representation of the text corpus, natural language processing and statistical NLP algorithms, (pattern) search algorithms, conversion predicates and the XPCE[7] code for the user interface.

Before the introduction of the PlDoc package only part of the user interface was documented on the tOKo homepage. Researchers and system developers who needed access to the predicates had to rely on the source code proper which, given the sheer size, is far from trivial. In practice, most researchers simply contacted the development team to get a handle on "where to start". This example shows that when open source software has non-trivial or very large interfaces it is necessary to complement the source code with proper documentation of at least the primary API predicates.

After the introduction of PlDoc all new tOKo functionality is being documented using the PlDoc style of literate programming. The main advantages have already been mentioned,

[6] `http://www.swi-prolog.org/emacs.html`
[7] `http://www.swi-prolog.org/packages/xpce/`

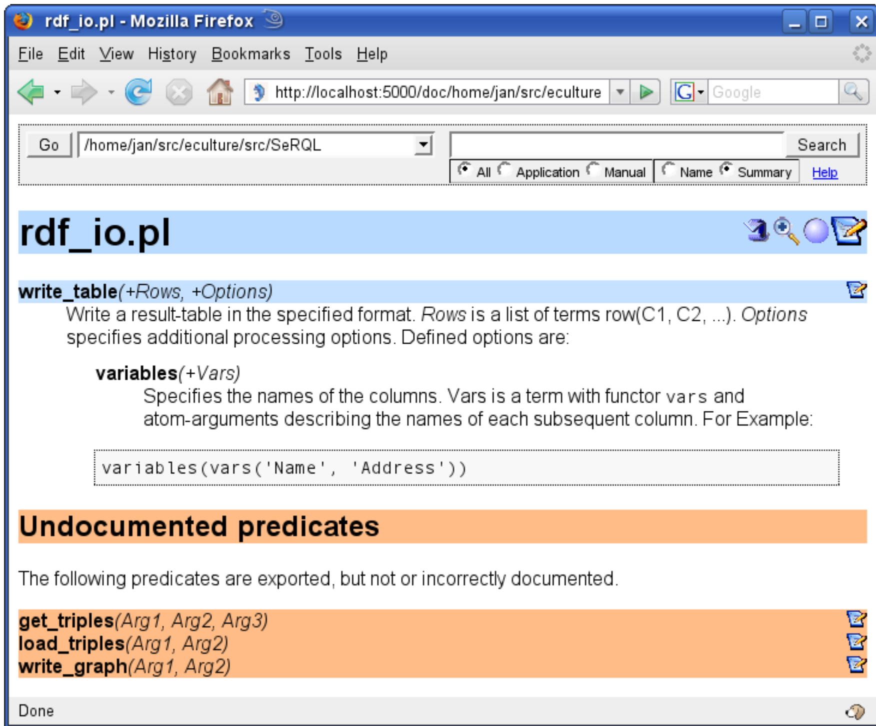


Figure 8.7: Undocumented public predicates are added at the bottom. When accessed from 'localhost', the developer can click the *edit* icon, add or fix documentation and click the *reload* icon at the top of the page to view the updated documentation.

in particular the immediate reward for the programmer. The intuitive notation of PlDoc also makes it relatively easy to add the documentation. The Emacs Prolog mode developed for SWI-Prolog[8] automatically reformats the documentation, such that mixing code and documentation becomes natural after a short learning curve.

One of the biggest advantages of writing documentation at all is that it reinforces a programmer to think about the names and arguments of predicates. For many of the predicates in tOKo the form is `operation`(*Output, Options*) or `operation`(*Input, Output, Options*). Using an option list, also common in the ISO standard predicates and the SWI-Prolog libraries, avoids an explosion of predicates. For example, `misspellings_corpus/2`,

[8] `http://turing.ubishops.ca/home/bruda/emacs-prolog`

which finds misspellings in a corpus of documents, has options for the algorithm to use, the minimum word length and so forth: misspellings_corpus(*Output, [minimum_length(5), edit_distance(1), dictionary(known)]*). Without documentation, once the right predicate is found, the programmer still has to check and understand the source code to determine which options are to be used. Writing documentation forces the developer to think about determining a consistent set of names of predicates and names of option type arguments.

A problem that the PlDoc approach only solves indirectly is when complex data types are used. In tOKo this for example happens for the representation of the corpus as a list of tokens. In a predicate one can state that its first argument is a list of tokens, but a list of tokens itself has no predicate and the documentation of what a token list looks like is non-trivial to create a link to. Partial solutions are to point to a predicate where the type is defined, possibly from a @see keyword or point to a .txt file where the type is defined.

Completing the PlDoc style documentation for tOKo is still a daunting task. The benefits for the developer are, however, too attractive not to do it.

8.6 Related work

The lpdoc system (Hermenegildo 2000) is the most widely known literate programming system in the Logic Programming world. It uses a rich annotation format represented as Prolog directives and converts these into Texinfo (Chassell and Stallman 1999). Texinfo has a long history, but in our view it is less equipped for supporting interactive literate programming for Logic Programming in a portable environment. The language lacks the primitives and notational conventions used in the Logic Programming domain and is not easily extended. The required TEX-based infrastructure and way of thinking is not readily available to any Prolog programmer.

In Logtalk (Moura 2003), documentation supporting declarations are part of the language. The intermediate format is XML, relying on XML translation tools and style sheets for rendering in browsers and on paper. At the same time the structure information embedded in the XML can be used by other tools to reason about Logtalk programs.

The ECLiPSe[9] documentation tools use a single comment/1 directive containing an attribute-value list of information for the documentation system. The Prolog-based tools render this as HTML or plain text.

PrologDoc[10] is a Prolog version of JavaDoc. It stays close to JavaDoc, heavily relying on '@'-keywords and using HTML for additional markup. Figure 8.8 gives an example.

Outside the Logic Programming domain there is a large set of literate programming tools. A good example is Doxygen (van Heesch 2007). The Doxygen website[11] provides an overview of related systems. Most of the referenced systems use structured comments.

[9] http://eclipse.crosscoreop.com/doc/userman/umsroot088.html
[10] http://prologdoc.sourceforge.net/
[11] http://www.doxygen.org

```
/**
    @form member(Value,List)
    @constraints
    @ground Value
    @unrestricted List
    @constraints
        @unrestricted Value
        @ground List
    @descr True if Value is a member of List
*/
```

Figure 8.8: An example using PrologDoc

8.7 The PlDoc LaTeX backend

Although the embedded HTTP server backend is our primary target, PlDoc is capable of producing LaTeX files from both Prolog files and plain text files using Wiki style markup as described in section 8.4.1. The LaTeX backend can be used to create a standalone LaTeX document from a Prolog file. Currently however, it is primarily used to generate (sub-)sections of the documentation, where the main structure of the documentation is managed directly in LaTeX and the generated fragments are included using LaTeX `\input` or `\include` statements. This approach provides flexibility by distributing documentation source at will over pure LaTeX, PlDoc Wiki and literate programming in the Prolog source while producing uniform output. Notably the **[[. . .]]** syntax (section 8.4.1) can be used to make Wiki files include each other and import predicate descriptions at appropriate locations in the text.

We describe a simple scenario in which we produce a user guide section on Prolog list processing. Because it concerns a user guide, we do not simply want to include the full API of the `lists.pl` library, but we do want to include descriptions of predicates from this library. We decide to write the section using the plain text conventions and create a file `lists.txt` using the content below.

```
---+ Prolog classical list processing predicates

List membership and list concatenation are the two
classical Prolog list processing predicates.

    * [[member/2]]
    * [[append/3]]
```

This text is processed into a LaTeX file `lists.tex` where the outermost section level is

`\subsection` using the sequence of Prolog commands below. The LaTeX output is shown in figure 8.9.

```
?- doc_collect(true).    % start collecting comments
?- [library(lists)].     % load the predicates with comments
?- doc_latex('lists.txt', 'lists.tex',
             [ stand_alone(false),
               section_level(subsection)
             ]).
```

```
% This LaTeX document was generated using the LaTeX backend of
% PlDoc, The SWI-Prolog documentation system

\subsection{Prolog classical list processing predicates}

List membership and list concatenation are the two classical Prolog
list processing predicates.

\begin{description}
    \predicate{member}{2}{?Elem, ?List}
True if \arg{Elem} is a member of \arg{List}

    \predicate{append}{3}{?List1, ?List2, ?List1AndList2}
\arg{List1AndList2} is the concatination of \arg{List1} and
\arg{List2}
\end{description}
```

Figure 8.9: LaTeX output from `lists.tex`

8.8 Implementation issues

In section 8.2, we claimed that a tight integration into the environment that makes the documentation immediately available to the programmer is an important asset. SWI-Prolog aims at providing a modular *Integrated Development Environment* (IDE) that allows the user to replace modules (in particular the editor) with a program of choice. We also try to minimise installation dependencies. Combined, these two constraints ask for a modular Prolog-based implementation. This implementation has the following main components:

- Collect documentation
- Parse documentation

- Rendering comments as HTML or LaTeX
- Publish using a web-server

8.8.1 Collecting documentation

Collecting the documentation must be an integral part of the development cycle to ensure that the running program is consistent with the presented documentation. An obvious choice is to make the compiler collect comments. This is achieved using a hook, which is called by the compiler called as:

`prolog:comment_hook`(+*Comments*, +*TermPos*, +*Term*).

Here, *Comments* is a list of *Pos-Comment* terms representing comments encountered from where `read_term/3` started reading upto the end of *Term* that started at *TermPos*. The calling pattern allows for processing any comment and distinguishes comments outside Prolog terms from comments inside the term.

The hook installed by the documentation server extracts structured comments by checking for `%%` or `/**`. For structured comments, it extracts the formal comment header and the first line of the comment body which serves, like JavaDoc, as a *summary*. The formal part is processed and the entire structured comment is stored unparsed, but indexed using the parsed formal header and summary that link the comment to a predicate. The stored information is available through the public Prolog API of PlDoc and can be used, together with the cross referencer Prolog API, as the basis for additional development tools.

8.8.2 Parsing and rendering comments

If a structured comment has to be presented through the web-interface or converted into LaTeX, it is first handed to a Prolog grammar that identifies the overall structure of the text as paragraphs, lists and tables. Then, the text is enriched by recognising hyperlinks (e.g., `print/2`) and font changes (e.g., `_italic_`). These two phases produce a Prolog term that represents the structure in the comment. The final step uses the HTML output infrastructure described in section 7.2.2.1 or a similarly designed layer to produce LaTeX.

8.8.3 Porting PlDoc

PlDoc is Open Source and may be used as the basis for other Prolog implementations. The required comment processing hooks can be implemented easily in any Prolog system. The comment gathering and processing code requires a Quintus style module system. The current implementation uses SWI-Prolog's nonstandard (but not uncommon) packed string datatype for representing comments. Avoiding packed strings is possible, but increases memory usage on most systems.

The web server relies on the SWI-Prolog HTTP package, which in turn relies on the socket library and multi-threading support. Given the standardisation effort on thread support in

Prolog (Moura et al. 2008), portability may become feasible. In many situations it may be acceptable and feasible to use the SWI-Prolog hosted PlDoc system while actual development is done in another Prolog implementation.

8.9 Conclusions

In literate programming systems there are choices on the integration between documentation and language, the language used for the documentation and the backend format(s). Getting programmers to document their code is already hard enough, which provided us with the motivation to go for minimal work and maximal and immediate reward for the programmer. PlDoc uses structured comments using Wiki-style documentation syntax extended with plain-text conventions from the Prolog community. The primary backend is HTML+CSS, served from an HTTP server embedded in Prolog. The web application provides a unified search and view for the application code, Prolog libraries and Prolog reference manual. PlDoc can be integrated in web applications, creating a self-documenting web server. We use this in ClioPatria (chapter 10). The secondaty backend is LaTeX, which can be used to include documentation extracted from Prolog source files into LaTeX documents.

PlDoc exploits some key features of the infrastructure of part I and the Prolog language to arrive at an interactive documentation system that is tightly integrated with the development cycle and easily deployed. Prominent is of course the web-server infrastructure described in chapter 7 which provides the HTTP server and HTML generation library. Threading (chapter 6) ensures that the documentation server does not interfere with the interactive toplevel. Reflexiveness and incremental compilation are properties of the Prolog language that allow for a simple to maintain link between source and documentation and access to up-to-date documentation during interactive development.

Currently, PlDoc cannot be called *knowledge-intensive*. In part, this was a deliberate choice to keep the footprint of PlDoc small. In the future we may consider more advanced search features that could be based on vocabularies from computer science such as FOLDOC[12] to facilitate finding material without knowledge of the particular jargon used for describing it. It is also possible to exploit relations that can be retrieved from the program, such as the modular organisation and dependency graphs. This thesis describes all components needed to extend PlDoc with support for knowledge-based search in the system and application documentation.

Acknowledgements

Although the development of PlDoc was on our radar for a long time, financial help from a commercial user in the UK finally made it happen. Comments from the SWI-Prolog user community have helped fixing bugs and identifying omissions in the functionality.

[12] `http://foldoc.org`

Chapter 9

Semantic annotation and search

About this chapter The material presented in this chapter is composed from two articles about the ICES-KIS project "Multimedia Information Analysis" (MIA). Schreiber, Dubbeldam, Wielemaker, and Wielinga (2001) (section 9.1) appeared in *IEEE Intelligent systems* and Wielemaker, Schreiber, and Wielinga (2003a) appeared as a book chapter in the series *Frontiers in Artificial Intelligence and Applications*.

Both papers study the use of ontologies for image annotation. The annotation and search prototypes were built in SWI-Prolog, using XPCE (chapter 5, Wielemaker and Anjewierden 2002) for the GUI. The knowledge representation is based on RDF, using a pure Prolog-based RDF store which provided the experience for chapter 3 (Wielemaker et al. 2003b). This chapter describes early work in applying emerging technology from the Semantic Web community to annotating multi media objects. It has been included in this thesis as motivation and background for chapters 2 to 4 and 10. The lessons learned are described in section 9.3 at the end of this chapter. The search and user evaluation sections of section 9.1 have been removed because they are considered irrelevant to this thesis.

9.1 Ontology-Based Photo Annotation

Guus Schreiber, Barbara Dubbeldam, Jan Wielemaker and Bob Wielinga
IEEE intelligent systems, 2001

9.1.1 Introduction

For magazine editors and others, finding suitable photographs for a particular purpose is increasingly problematic. Advances in storage media along with the Web enable us to store and distribute photographic images worldwide. While large databases containing photographic

images exist, the tools and methods for searching and selecting an image are limited. Typically, the databases have a semistructured indexing scheme that allows a keyword search but not much more to help the user find the desired photograph.

Currently, researchers promote the use of explicit background knowledge as a way out of the search problems encountered on the Internet and in multimedia databases. The *semantic Web* (Berners-Lee 1999) and emerging standards (such as the resource description framework (RDF, Brickley and Guha 2000) make creating a syntactic format specifying background knowledge for information resources possible.

In this article, we explore the use of background knowledge contained in ontologies to index and search collections of photographs. We developed an annotation strategy and tool to help formulate annotations and search for specific images. The article concludes with observations regarding the standards and tools we used in this annotation study.

9.1.2 Our approach

Companies offering photographic images for sale often provide CDs containing samples of the images in reduced jpeg format. Magazine editors and others typically search these CDs to find an illustration for an article. To simulate this process, we obtained three CDs with collections of animal photo samples with about 3,000 photos.

Figure 9.1 shows the general architecture of our annotation tool. We specified all ontologies in RDF Schema (RDFS, Brickley and Guha 2000) using the Protégé-2000 (Fridman Noy et al. 2000) ontology editor (version 1.4). This editor supports the construction of ontologies in a frame-like fashion with classes and slots. Protégé can save the ontology definitions in RDFS. The SWI-Prolog RDF parser (chapter 3, Wielemaker et al. 2003b) reads the resulting RDFS file into the annotation tool, which subsequently generates an annotation interface based on the RDFS specification. The tool supports reading in photographs, creating annotations, and storing annotations in an RDF file. A query tool with a similar interface can read RDF files and search for suitable photographs in terms of the ontology.

The architecture shown in figure 9.1 is in the same spirit as the one described in Lafon and Bos 2000. However, we place more emphasis on the nature of the ontologies, the subject matter description, and the explicit link to a domain ontology.

9.1.3 Developing ontologies

To define semantic annotations for ape photographs, we needed at least two groups of definitions:

- *Structure of a photo annotation.* We defined a *photo annotation ontology* that specifies an annotation's structure independent of the particular subject matter domain (in our case, apes). This ontology provides the description template for annotation construction.

- *Subject matter vocabulary.* We also constructed a *domain-specific ontology* for the animal domain that provides the vocabulary and background knowledge for describing

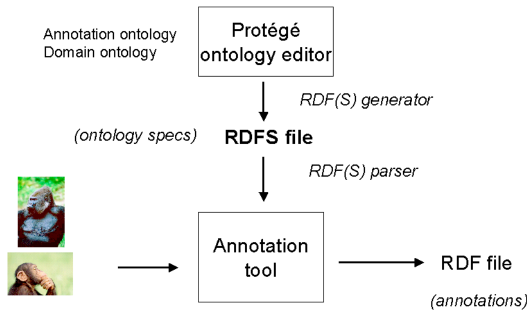


Figure 9.1: Overview of the approach used in this study. We used the Protégé ontology editor to construct ontologies and store them in RDFS format.

features of the photo's subject matter. In this case, the ontology consisted of definitions of the phylum hierarchy of ape species with the corresponding species' attributes and constraints.

9.1.3.1 Photo annotation ontology

The first decision was whether we could use an existing annotation template as a starting point. After evaluating metadata standards such as Dublin Core, (Dublin Core Metadata Initiative 1999) it was clear that they were developed for another purpose and weren't well suited for extensive content-oriented annotations. Because ontology-based annotation is relatively new, we decided to set the existing annotation standards aside and define the annotation ontology based on our own analysis.

When looking at a photo, what kind of things do we want to state about it? We distinguished three viewpoints:

- *What does the photo depict?* We call this the photo's *subject matter feature*. For example, a photo depicts a gorilla eating a banana. This part of the photo annotation ontology links to the domain ontology.
- *How, when, and why was the photo made?* We call this the *photograph feature*. Here, we specify metadata about the circumstances related to the photo such as the photographer or the vantage point (for example, a close-up or an aerial view).
- *How is the photo stored?* We call such photo characteristics the *medium feature*. This represents metadata such as the storage format (such as jpeg) or photo resolution.

In this study, we focused mainly on the subject matter description. In Dublin Core, the single element *subject* represents this aspect.

Figure 9.2 gives an overview of the annotation ontology represented as a UML class diagram. A *photo annotation* contains at least one subject matter description and an arbitrary number of photograph features and medium features. The subject matter description has an internal structure. The actual photograph and medium features are subclasses of the abstract feature concepts. The subclasses, shown in gray, represent just a sample collection of features. The annotation tool only makes a number of minimal assumptions about the annotation ontology. This lets us add new features to the ontology.

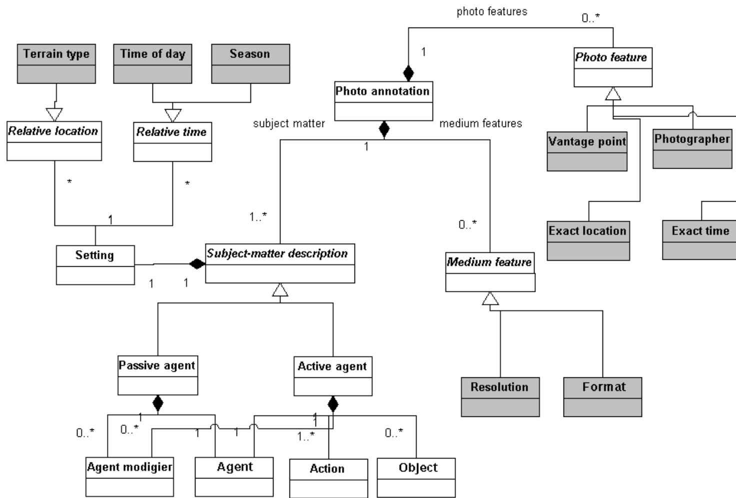


Figure 9.2: Structure of the photo annotation ontology represented as a UML class diagram.

When constructing an ontology, we often needed to incorporate definitions already available in other corpora. For example, to define a *colour* feature, we don't want to type in all the possible values for "colour." A resource such as WordNet (Miller 1995) already contains this information. In this study, we used the WordNet plug-in for Protégé. It provides a cut-and-paste method for importing sections of WordNet into an ontology.

9.1.3.2 Structure of the subject matter description

From the content perspective, the subject matter description is the most interesting part of the ontology. For this, we used the notion of *structured annotation* described in Tam and Leung 2001. They propose a description template consisting of four elements:

1. An *agent*, for example, "an ape." An agent can have modifiers such as "colour =

orange."

2. An *action*, for example, "eating."
3. An *object*, for example, "a banana." Objects can also have modifiers (colour = "green").
4. A *setting*, for example, "in a forest at dawn."

We used this general scheme to define two description templates that we found useful in our example domain:

1. A *passive agent* is a restricted form of the scheme with a single agent, any number of agent modifiers, and a setting.
2. An *active agent* is the complete template with a single agent, agent modifiers, an action, optionally an object, and a setting.

The setting is typically restricted to two context dimensions, namely relative time (for example, time of day or season) and relative location (for example, terrain type). Here we copied parts of the WordNet vocabulary.

9.1.3.3 The subject matter ontology

The subject matter ontology describes the vocabulary and background knowledge of the photo's subject domain. For this study, we developed a domain ontology based on the phylum hierarchy of animal species. Phylum is a metaclass with a number of properties (*slots* in the Protégé terminology). Table 9.1 shows the properties we currently use.

A particular species represents a class that is an instance of metaclass phylum. This organisation of the ontology lets us define instance-type features of species—for example, that an orangutan has an "orange" colour and has as geographical range "Indonesia," while still being able to treat a species as a class. This sloppy class-instance distinction is a feature of Protégé-2000 that makes it well suited for complex metamodeling.

We specified the phylum hierarchy through subclass relations between species classes. For example, an orangutan is a "great ape," a subclass of "ape," a subclass of "primate," and so forth. Features that are characteristic of apes in general are specified at the hierarchy's appropriate level and are inherited by the species subclasses.

Figure 9.3 shows a snapshot of the Protégé-2000 ontology editor with the species hierarchy (left) and some characteristics defined for an "orangutan" (right). Sometimes, characteristics are inherited from classes higher up in the hierarchy (for example, the life-stage terms).

9.1.3.4 General terminology

Both the annotation ontology and the domain ontology use general terminology. Instead of defining this ourselves, we used parts of WordNet (Miller 1995) and IconClass (van der

Species feature	Description
Geographical range	The geographical area where the animal typically lives; for example, “Africa,” “Indonesia.”
Typical habitats	The terrain where the animal usually lives; for example, “rain forest,” “savanna.”
Life-stage terminology	Terms used to talk about life stages of animals; for example, “lamb,” “cub.”
Gender terminology	Terms used to talk about male and female animals; for example, “lioness.”
Group terminology	Terms used to talk about a group of these animals; for example, “troop,” “herd.”
Colour features	Description of general colours (“orange”) or colour patterns (“striped”).
Other characteristics	A catch-all category for characteristics such as “captive animal” and “domestic animal.”

Table 9.1: Features of animal species defined in the domain ontology.

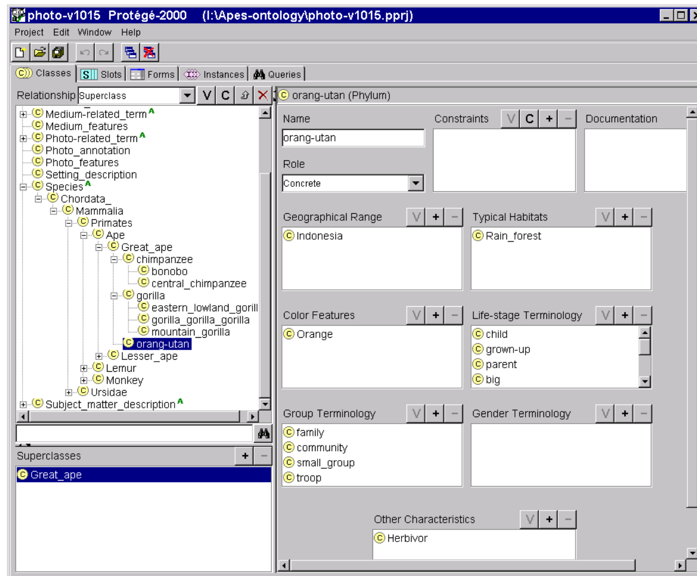


Figure 9.3: Snapshot of the Protégé-2000 ontology editor showing part of the domain ontology.

Waal 1985) —for example, WordNet includes a collection of *vantage points* (a photograph feature). In other cases, WordNet provides a partial value set for a feature value—for example, when we want to describe an ape's colour aspects, we want to use both general colours ("orange") as well as animal-specific colour terms ("striped"). Therefore, we can expect that in general we might want to include definitions from many different sources in the ontologies required for annotations. To take another domain, if we want to annotate pictures of art objects, we would like to use the Art and Architecture Thesaurus (AAT) thesaurus, IconClass, and possibly many other sources. This means we need a structured approach for linking domain vocabularies.

Figure 9.4 shows a graphical overview of the ontologies and vocabularies using the UML package notation. The links represent UML dependencies: "⟨*source*⟩ depends on ⟨*destination*⟩." Due to technical limitations we were forced to realise 'depends on' by physically importing ontologies due to two problems. First, the Protégé tool does not support ontology modularisation—we can import an ontology by copying, but cannot create a separate module for it. Second, RDFS versions of most vocabularies do not exist. It seems reasonable to expect that such a version of WordNet will become available in the near future, but it is not known whether domain-specific vocabularies (such as AAT) will. The alternative is to write dedicated access routines for vocabularies. In the European GRASP project ("Global Retrieval, Access and information System for Property items"), a CORBA-based ontology server directly links descriptions of art objects to elements of the AAT.

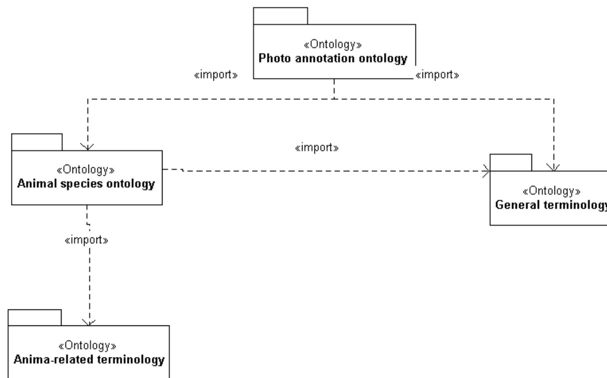


Figure 9.4: Import relations between the ontologies for animal photograph annotations using the UML package notation.

9.1.3.5 Linking the annotation ontology with the domain ontology

To keep the annotation ontology and the subject matter ontology separate, we defined an explicit mapping between the subject matter description in the former ontology to the phylum hierarchy in the later ontology. Figure 9.5 shows part of this mapping. This figure contains a snapshot of the RDFS browser part of the tool we developed. In the figure, we see the description of the RDFS class *passive agent*, a subclass of subject matter description. The class has three properties: the *setting* property links to a resource of type *setting description* (which in turn consists of *relative time* and *relative location* properties); the property *agent modifier* links a passive agent description to an *animal characteristic*.; and the property *agent* indicates that the agent should be some species. The classes *species* and *animal characteristic* both belong to the domain ontology.

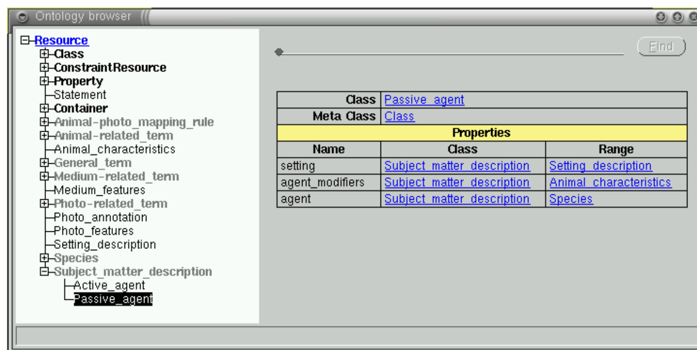


Figure 9.5: Snapshot of the tool's RDFS browser. Two properties of the subject matter description link to the animal domain ontology (agent → species; agent modifier → animal characteristic).

Although our mappings are simple, we expect other mappings will be more complex, especially in cases where there is no simple one-to-one mapping. Research on ontology mapping and merging (McGuinness et al. 2000) is required.

9.1.4 Annotating photographs using our multimedia information analysis tool

The tool we developed reads an RDFS file containing ontology specifications. The RDFS produced by Protégé conforms to the W3C standard, (Brickley and Guha 2000) except for the range definition of properties. RDFS only allows a single type for a range constraint; this is too limited for Protégé. We handled this inconsistency by simply allowing multiple range constraints. The RDFS specification document indicates that we should specify a superclass

for multiple range classes, but this syntactic solution is not desirable from an ontological-engineering perspective because it breaks modular design: the new class is conceptually not part of the original concept hierarchy or hierarchies but belongs to the RDFS annotation template.

From the RDFS specifications, the tool generates a user interface for annotating photos. Figure 9.6 shows a snapshot of the annotation interface. There are three tabs for the three groups of features: subject matter, photograph, and medium. The figure shows the passive agent template for a subject matter description. In the example, the annotation says the agent is a chimpanzee with two modifiers, namely "life stage = young" and "posture = scratching-the-head."

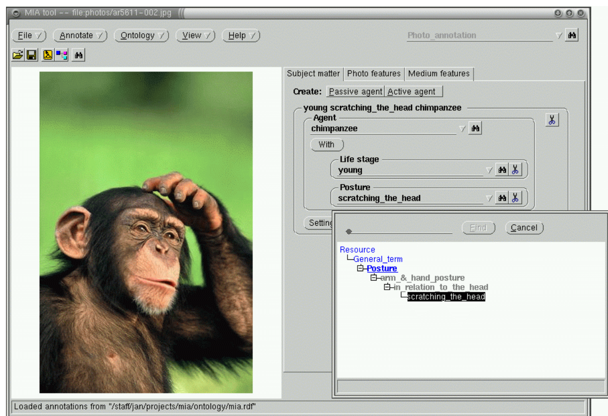


Figure 9.6: A snapshot of the annotation interface. The user has selected "chimpanzee" as the animal species. Two agent modifiers are defined: "life stage" and "posture." At the lower right, the part of the domain ontology is shown containing fillers for the "posture" feature.

The user can enter terms in two ways. He or she can type in a term and use a completion mechanism to see whether it matches a term in the ontology (the typed text becomes bold when matched). Alternatively, the user can click on the magnifier icon to browse the relevant part of the ontology to select a term. The pop-up window at the lower right shows this for the agent modifier *posture*. The window lets the user select a term from the hierarchy under the class *posture*. The hierarchy of terms comes from WordNet and IconClass. The tool also supports administrative tasks such as reading in new photos, storing photo annotations, and loading existing photo annotations.

The user interface generator can be defined almost independently of the ontology. The generator reads the RDFS representing an annotation schema. For each property of this

schema that represents another compound schema, it generates a *tab* or sentence item. If a property refers to an ontological term, it generates an item providing completion, search, and hierarchical browsing. Finally, for properties defined as RDF-literal, it generates a simple text item. Schemas entered as a "sentence" require an additional declaration to tell the generator the order in which the properties appear in the sentence.

Such a generic interface also has some drawbacks. The RDFS's structure maps directly to the interface, but grouping based on ontological motivations is not necessarily the best choice for the UI, and properties in RDF provide no ordering information. Also, the interface uses abstract terms such as *agent* that might not be intuitive for users. For practical applications, the user must provide additional grouping, ordering and label information to the UI generator.

9.1.5 Discussion

This study has only scratched the surface of the problems encountered when trying out a content-oriented approach to annotate and search for photos.

9.1.5.1 What do ontologies offer over keywords?

In photo collections indexed with keywords, a small subset of the controlled keyword set is associated with an image. The keywords themselves are unrelated atoms. If we consider the terms of the ontology to be our controlled keyword list, using an ontology and a structured description based on this ontology changes the annotation and querying process in a number of ways:

- It guides the annotation process using restrictions and default information.
- It makes the relation between property values and agents explicit, telling which property value is connected using which property to which element of the subject matter or the photo itself. Consider "chimpanzee under large tree." Reduced to keywords, "large" can refer to the chimpanzee, the tree, or even the photo.
- The ontology provides relations between the terms; in our case, default information ("orangutans live in Indonesia") and inheritance. Inheritance provides a controlled means to widen or constrain a query.

In our view, there is enough evidence to warrant further research, but there is a long way to go to actually prove that ontology-based search is better (in some respects) than keyword search.

9.1.5.2 Architecture feasibility

The architecture presented in this article provides a Semantic Web standard-conforming framework for representing ontologies and photo annotations. The tool makes two limiting assumptions. First, it assumes the annotated object is an image. Second, it assumes an

RDF Schema defining a set of classes with properties that represent an annotation. Each class either has atomic attributes or has attributes for which a sentence schema is defined.

Currently, the tools represent the RDF data as triples in the Prolog database. The triple database includes three modules: one for the ontological data, one for the annotations, and one for the queries. Neither this representation nor the used RDF query algorithm will scale to large databases (greater than 100,000 triples). Scalability of RDF storage and query systems is an active research topic (see the RDF mailing list, `rdf-interest@w3c.org`).

9.1.5.3 Guidelines for using Web standards

During this study, we used RDFS, RDF, XML, and XML-DTD. We started using the latter to define the photo annotation structure until we realised we were using a language intended for defining syntactical structure for the specification of semantical structure. At that point, we decided to treat a photo annotation as a semantic unit described in RDF and to define its structure as an RDFS.

Likewise, an ontology can be expressed in RDF (the OpenDirectory project[1]), but this approach loses the frame semantics of RDFS. Ontological class definitions typically require constrained properties and inheritance. This means that RDFS is a much more suitable formalism than plain RDF. If one limits the formalism to pure RDF, the ontology itself is machine-readable, but not machine-understandable.

We extended RDFS by refining *class* to add additional knowledge to the model, such as default values for properties. OIL (Fensel et al. 2000) uses the same mechanism to extend the semantics of RDFS. Unfortunately, these extensions are not generally machine-understandable. RDFS can only grow by the acceptance of OIL and OIL-like extensions as additional standards.[2]

9.1.5.4 The link with Dublin Core

In this article, we focused on annotations about the content of a photograph. In terms of the Dublin Core element set, (Dublin Core Metadata Initiative 1999) our structured subject matter description is an elaborate refinement of the *subject* element. For some of the photograph features and medium features, the link with Dublin Core is more straightforward. Table 9.2 shows the mapping between features in our annotation ontology and Dublin Core elements. Assuming an official RDFS specification of Dublin Core becomes available, we can redefine these features as subproperties of the corresponding Dublin Core elements. (Currently, there is only an unofficial one in Appendix B of the RDFS document mainly intended as an example of the use of RDFS.) In this case study, the Dublin Core *type* element will always have the value *image* (following the DCMI type vocabulary). In this way, we can ensure that the resulting photo annotations comply with Dublin Core's *dumb-down principle*, which states

[1] `http://www.dmoz.org/`

[2] This role is played by OWL (Dean et al. 2004), the Semantic Web language that is in part based on DAML and OIL.

that refinements of the element set are allowed provided it is still possible to access the annotation through the basic element set.

Annotation ontology	Feature type	Dublin Core element (qualifier)
Copyright holder	Photo feature	Rights
Photographer	Photo feature	Creator
Exact time	Photo feature	Coverage (temporal)
Exact location	Photo feature	Coverage (spatial)
Format	Medium feature	Format
Resolution	Medium feature	Format
Size	Medium feature	Format (extent)
Photograph colour	medium feature	Format

Table 9.2: Correspondence between features of the annotation ontology and Dublin Core element.

9.1.5.5 Support tools

Support tools are crucial for making the architecture sketched in figure 9.1 work. The Protégé-2000 tool proved useful for our study. The RDFS generated by Protégé conformed to the W3C standard, with the exception of range constraints. The main problems we had with Protégé was that it does not support multiple ontologies with import relations and it was incapable of loading large ontologies such as WordNet. Lack of modularity clutters the ontology definitions. Once multiple ontologies can be handled, it is likely that tool requirements will come up with respect to ontology-mapping mechanisms (for example, defining the link between the subject matter description and the domain ontology).

9.1.5.6 Preprocessing existing annotations

Most photos in existing collections are annotated. This was also true for the photos we used in our study. The nature of the annotation varied considerably—one CD contained free-text annotations and another used keywords. Depending on the amount of useful information in the existing annotation, it might be worthwhile to consider the construction of a preprocessor to generate a (partial) semantic annotation from the existing annotation. Natural language analysis techniques are likely to be required for such preprocessing.

Acknowledgments

The Dutch government's ICES-KIS project "Multimedia Information Analysis" supported this work. We also thank Audrey Tam for the discussion about combining structured annotations with ontologies and Hans Akkermans, Michel Klein, Lloyd Rutledge, and Marcel Worring for other beneficial discussions. We are also grateful to the anonymous reviewers for their detailed and helpful comments.

9.2 Supporting Semantic Image Annotation and Search

Jan Wielemaker, Guus Schreiber and Bob Wielinga
Frontiers in Artificial Intelligence and Applications, 2003

Abstract In this article we discuss an application scenario for semantic annotation and search in a collection of art images. This application shows that background knowledge in the form of ontologies can be used to support indexing and search in image collections. The underlying ontologies are represented in RDF Schema and are based on existing data standards and knowledge corpora, such as the VRA Core Categories 3.0, the Art and Architecture Thesaurus and WordNet. This work is intended to provide a "proof of concept" for indexing and search in a Semantic Web.

9.2.1 Introduction

In this article we are exploring the possibilities of using background knowledge for indexing and querying an image collection. Many of such collections currently exist and users are increasingly faced with problems of finding a suitable (set of) image(s) for a particular purpose. Each collection usually has its own (semi-)structured indexing scheme that typically supports a keyword-type search. However, finding the right image is often still problematic.

In this work we investigate an alternative approach. We explore the question whether ontologies can support the indexing and querying process in any significant way. This work should be seen as a "proof of concept". At the moment, the use of explicit background knowledge is often-mentioned as a way out of the search problems such as currently arise on the Internet and in multimedia databases. This new research area is part of the "Semantic Web" (Berners-Lee 1999). Emerging web standards such as RDF (Lassila and Swick 1999) are providing a syntactic format for explicating background knowledge of information resources. However, the added value of such semantic annotations still has to be proven. Within the scope of this work we have not done any detailed evaluation studies of the approach presented. Our sole aim is to see whether we can find sufficient evidence that this is a plausible approach worth further investigation.

9.2.2 Approach

Figure 9.7 shows the general architecture we used within this study. All ontologies were represented in RDFS Schema (Brickley and Guha 2000). The resulting RDF Schema files are read into the tool with help of the SWI-Prolog RDF parser (chapter 3, Wielemaker et al. 2003b, Parsia 2001). The annotation tool subsequently generates a user interface for annotation and search based on the RDF Schema specification. The tool supports loading images and image collections, creating annotations, storing annotations in a RDF file, and two types of image search facilities.

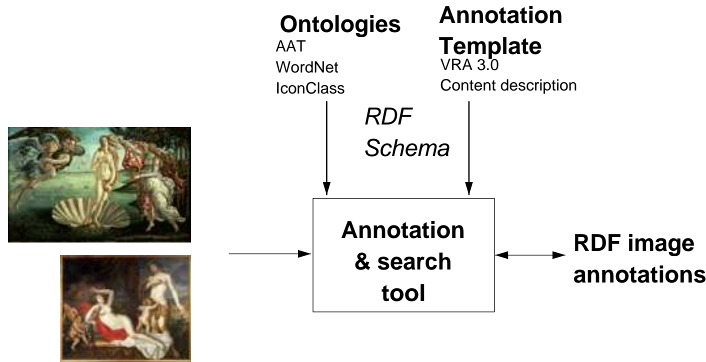


Figure 9.7: Overview of the approach in this study. The RDF Schema specifications of the ontologies and of the annotation template are parsed by the RDF/RDFS parser into the tool. The tool generates an annotation and search interface from these specifications. This interface is used to annotate and query images. The annotations are stored in an RDF file.

The architecture shown in figure 9.7 is in the same spirit as the one described by Lafon and Bos 2000. The main difference lies in the fact that we place more emphasis on the nature of the ontologies. In the rest of this paper the architecture depicted in figure 9.7 is described in more detail. The background knowledge is discussed in section 9.2.3. The annotation and query aspects of the tool are discussed in, section 9.2.4 in the form of an application scenario. section 9.2.5 discusses the experiences gained.

This work is a sequel to earlier work on semantic annotation and search of a collection of photographs of apes (section 9.1, Schreiber et al. 2001). As the sample domain for the present work we used 200 images of paintings.

9.2.3 Background Knowledge

9.2.3.1 Ontologies

For this study we used three thesauri, which are relevant for the art-image domain:

1. The Art and Architecture Thesaurus (AAT, Peterson 1994) is a large thesaurus containing some 120,000 terms relevant for the art domain.

2. WordNet (Miller 1995) is a large lexical database. WordNet concepts ("synsets") are typically used to describe the content of the image, e.g., "woman sitting on bed"

3. IconClass (van der Waal 1985; van den Berg 1995) is an iconographic classification system, providing a hierarchically organised set of concepts for describing the content of visual resources.

AAT, WordNet and IconClass were translated into the RDF Schema notation. In a prior publication (Wielinga et al. 2001) one can find a discussion on issues arising when representing AAT in RDF Schema.

These sources share a significant number of terms. This could be used to design a new ontology by *merging* them. In the context of the Semantic Web however, it is important to *re-use* existing work rather than modifying it. Therefore we have added equivalence relations between terms appearing in multiple ontologies that refer to the same entity in the world. For example, the IconClass concept **king** is linked to the AAT concept **kings (people)**, making all IconClass subterms of **king** valid subterms of AAT **kings (people)** and visa-versa.

9.2.3.2 Annotation template

For annotation purposes the tool provides the user with an annotation template derived from the VRA 3.0 Core Categories (Visual Resources Association Standards Committee 2000). The VRA template provides a specialisation of the Dublin Core set of metadata elements, tailored to the needs of art images. The VRA Core Categories follow the "dumb-down" principle (i.e., a tool can interpret the VRA data elements as Dublin Core data elements).

The 17 VRA data elements (i.e., properties of an art image) were for visualisation purposes grouped into three sets:

1. *Production-related descriptors* such as creator ("maker"), style/period, technique and, culture
2. *Physical descriptors* such as measurements and colour
3. *Administrative descriptors* such as collection ID, rights and current location

In figure 9.8 one can see a tab with the set of production-related descriptors. The other VRA descriptors can be found on two other tabs.

In addition, we needed to provide ways for describing the subject matter of the painting. VRA provides a subject element, but we were interested in providing more structured content descriptions. For this purpose we used a simple "sentence structure" which was developed for a previous experiment (section 9.1, Schreiber et al. 2001):

```
<agent> <action> <object> <setting>
```

Each content-description sentence should contain at least one agent or object. Agents and objects may have attributes ("modifiers"), such as colour, cardinality and position. The "setting" can be described with relative time (e.g., '"sunset") and relative place (e.g., "a forest"). In addition, the scene as a whole can be characterised. The application scenario in the next section gives an example of the use of this template. Multiple sentences can be used to describe a single scene.

9.2.3.3 Linking the annotation template to the ontologies

Where possible, a slot in the annotation template is bound to one or more subtrees of the ontologies. For example, the VRA slot style/period is bound to two subtrees in AAT containing the appropriate style and period concepts (see also figure 9.9). Four VRA data elements are currently linked to parts of AAT: technique, style/period, culture and material. Most parts of the subject-matter description are also linked to subtrees of the ontologies. Here, heavy use is made of WordNet and IconClass. The latter is in particular useful for describing scenes as a whole.

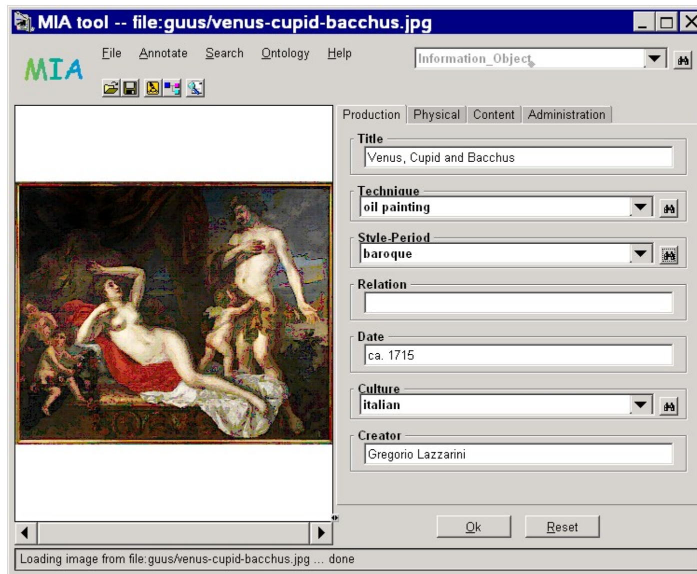


Figure 9.8: Snapshot of a semantic annotation and search tool for art images. The figure shows a fragment of the annotation window showing one tab with VRA data elements for describing the image, here the production-related descriptors. The slots associated with a "binoculars" button are linked to one or more subparts of the underlying ontologies, which provide the concepts for this part of the annotation.

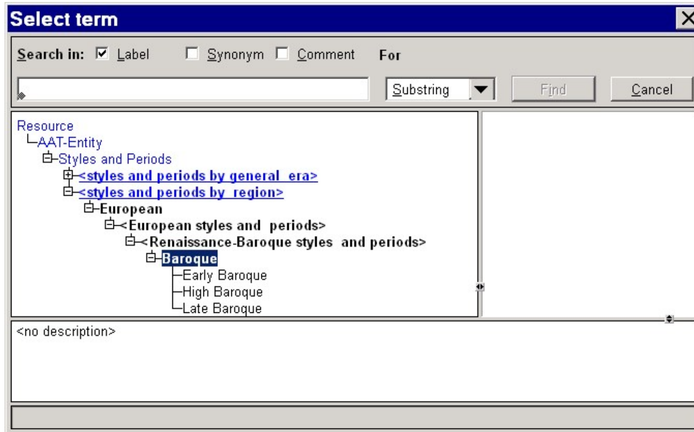


Figure 9.9: Browser window for values of style/period. The concept **baroque** has been selected as a value for this slot. The top-level concepts of the AAT subtrees from which we can select a value for style/period are shown with an underlined bold font (i.e., <**styles and periods by general era**> and <**styles and periods by region**>). The browser shows that there are three more specific concepts that could have been used (e.g., high baroque)

9.2.4 An Application Scenario

9.2.4.1 Annotating an image

Figure 9.8 shows a screen shot of the annotation interface. In this scenario the user is annotating an image representing a baroque Italian painting, depicting a mythological scene featuring Venus, Cupid and Bacchus. The figure shows the tab for production-related data elements. The three elements with a "binoculars" icon are linked to subtrees in the ontologies, in this case AAT. For example, if we would click on the "binoculars" for style/period the window shown in figure 9.9 would pop up, showing the place in the hierarchy of the concept **baroque**. We see that it is a concept from AAT. The top-level concepts of the AAT subtrees from which we can select a value for style/period are shown with an underlined bold font (i.e., <**styles and periods by general era**> and <**styles and periods by region**>). The ontology makes it easier for people to select the correct concept. For example, seeing that **baroque** contains three specialisations the user might want to use one of these terms, e.g., **high baroque**.

The user interface provides some support for finding the right concept. For example, the user can type in a few characters of a term and then invoke a completion mechanism (by typing a space). This will provide a popup list of concepts matching the input string.

In the browser window, more advanced concept search options can be selected, including substrings and use of synonyms.

The domain knowledge can be extended to cover more slots. For example, the creator slot could take values from the Getty thesaurus ULAN (Union List of Artist Names, Getty 2000).[3]

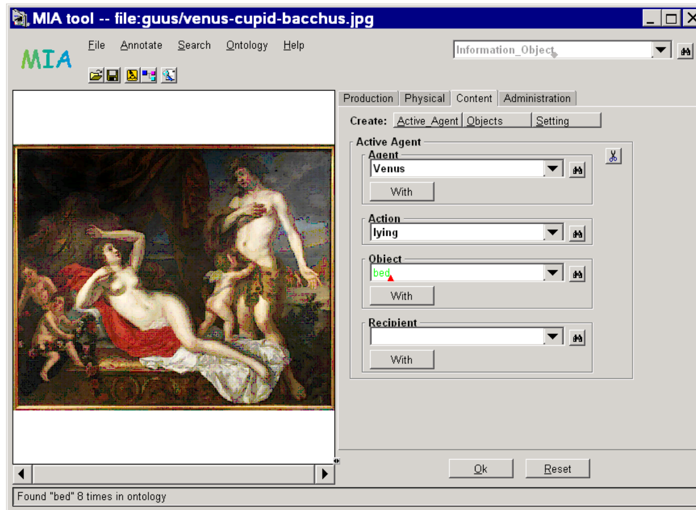


Figure 9.10: Description of the subject matter. Here, the content is described with the sentence "Venus (agent) lying (action) bed (object)". Multiple content descriptions can be entered for an image

For annotation purposes the ontologies serve two purposes. Firstly, the user is immediately provided with the right context for finding an adequate index term. This ensures quicker and more precise indexing. Also, the hierarchical presentation helps to disambiguate terms. An example of this is shown in figure 9.10. Here, the user is describing the subject matter of the painting. Suppose she wants to say that "Venus is lying on a bed". The template on the right-hand side implements the content template as described in section 9.2.3.2, in this case:

```
Agent:   Venus
Action:  lying
Object:  bed
```

[3]Including ULAN exceeded the scalability of our prototype. ClioPatria, as described in chapter 10 can easily deal with this scale of background information.

When the user types in the term "bed" as the object in a content-description template, the tool will indicate that this an ambiguous term. In the user interface the term itself gets a green colour to indicate this and the status bar near the bottom shows the number of hits in the ontologies. If one clicks on the binocular button, the tool will provide the user with a choice of concepts in AAT and WordNet that are associated with this term (piece of furniture, land depression, etc.). Figure 9.11 shows two of the concepts associated with "bed". From the placement of the terms in the respective hierarchies, it is usually immediately clear to the indexer which meaning of the term is the intended one.

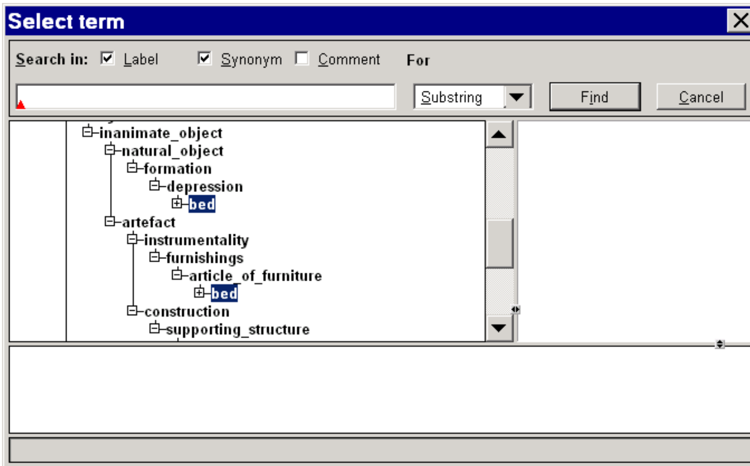


Figure 9.11: Disambiguating the term "bed". This window shows two of the concepts associated with bed. From the placement of the terms in the respective hierarchies, it is usually immediately clear to the indexer which meaning of the term is the intended one.

Term disambiguation is a frequent occurrence in this type of application. For example, the term "Venus" may be associated with the Roman deity or with the planet.[4]

9.2.4.2 Searching for an image

The tool provides two types of semantic search. With the first search option the user can search for concepts at a random place in the image annotation. Figure 9.12 shows an example of this. Suppose the user wants to search for images associated with the concept **Aphrodite**. Because the ontologies contain an equivalence relation between Venus (as a Roman deity,

[4]In the "Venus lying bed" example, this disambiguation of "Venus" is not required, because the planet is not considered a legal value for an "agent" (this is debatable, by the way). However, general search for "Venus" (see the next subsection) will require disambiguation

not the planet nor the tennis player) and Aphrodite, the search tool is able to retrieve images for which there is no syntactic match. For example, if we would look at the annotation of the first hit in the right-hand part of figure 9.12, we would find “Venus” in the title (“Birth of Venus” by Botticelli) and in the subject-matter description (**Venus (a Roman deity) standing seashell**). The word “Venus” in the title can only be used for syntactic marches (we do not have an ontology for titles), but the concept in the content description can be used for semantic matches, thus satisfying the “Aphrodite” query.

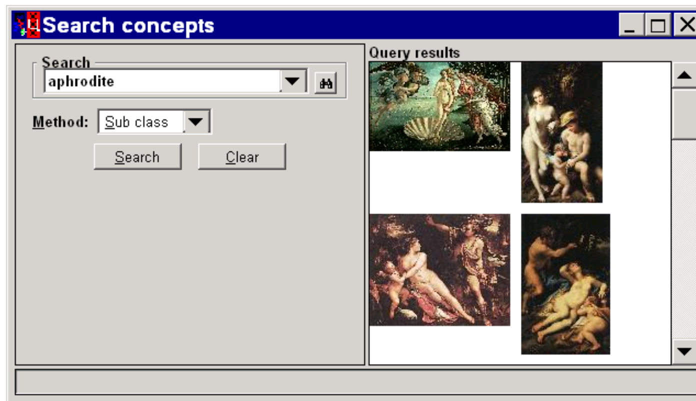


Figure 9.12: Example of concept search. The query “Aphrodite” will retrieve all images for which we can derive a semantic match with the concept **Aphrodite**. This includes all images annotated with the concept **Venus** (as a Roman deity). Only a small fragment of the search results is depicted

General concept search retrieves images which match the query in some part of the annotation. The second search option allows the user to exploit the annotation template for search proposes. An example of this is shown in figure 9.13. Here, the user is searching for images in which the slot culture matches **Netherlandish**. This query retrieves all images with a semantic match for this slot. This includes images of **Dutch** and **Flemish** paintings, as these are sub concepts of **Netherlandish**.

9.2.5 Discussion

This article gives some indication on how a Semantic Web for images might work. Semantic annotation allows us to make use of concept search instead of pure syntactic search. It paves also the way for more advanced search strategies. For example, users may be specialising or generalising a query with the help of the concept hierarchy when too many or too few hits are found.

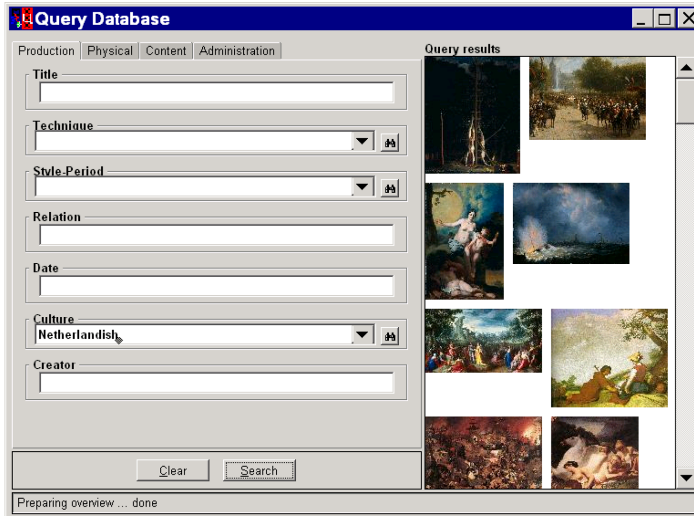


Figure 9.13: Search using the annotation template. The query "Netherlandish" for the slot culture retrieve all images with a semantic match for this slot. This includes images of **Dutch** and **Flemish** paintings, as these are sub concepts of **Netherlandish**

In a previous study on a collection of ape photographs (section 9, Schreiber et al. 2001) we did some qualitative analysis on the added value with respect to keyword search. The provisional conclusion was that for some queries (e.g., "ape") keyword search does reasonably well, but for other sightly different queries (e.g., "great ape") the results are suddenly poor. This is exactly where semantic annotation could help.

Although our approach relies to some extent on manual annotation (as we did in this study), it should be possible to generate partial semantic annotations from existing annotations (which vary from free text to structured database entries). This would speed up the annotation process considerably. We are currently starting a follow-up study with a collection of 4,000 images of paintings for which we want to extract data such as title, creator, year and location from the existing annotation.

Our experiences with RDF Schema were generally positive. We made heavy use of the metamodeling facilities of RDF Schema (which allows one to treat classes as instances of other classes) for defining and manipulating the meta models of the different thesauri. In our experience this feature is in particular needed in cases where one has to work with existing representations of large ontologies. This is a typical feature of a Semantic Web: you have to work with existing ontologies to get anywhere (we're not going to rebuild AAT), even if one disagrees with some of the design principles of the ontology.

For our purposes RDF Schema has some limitations in expressivity. We especially needed a notion of property cardinality and of equivalence between resources (classes, instances, properties). For this reason we plan to move at some near point in the future to OWL, the Web Ontology Language currently under development at W3C (W3C Web Ontology Working Group 2001).

A major challenge from a knowledge-engineering perspective is to provide principled methods for adding ontology "glue". The equivalence relations mentioned in section 9.2.3.1 are one example of this. In many cases existing ontologies can be augmented with additional background knowledge. For example, if we know the creator is André Derain (an artist in ULAN), we could suggest that the style of the painting is "Fauve" (a style in AAT). But for this to work, we need to add knowledge linking ULAN and AAT. In this area much more work needs to be done to see what is feasible and what is not,

9.3 Lessons learned

This chapter describes an application developed before the material described in part I of this thesis (except for chapter 5) was available. The overall aim of this research was to test the hypothesis that using background knowledge in the form of ontologies improves the annotation and search process for multi-media objects (images) compared to using simple keywords. We used software prototype development to explore architectural opportunities provided by the emerging Semantic Web. In particular, based on our experience from Shelley and the CommonKADS Workbench (see chapter 1), using the RDF triple model as the core storage formalism for applications was expected to be a good design. Exploiting the dynamic nature of XPCE/Prolog and in particular its automatic layout capabilities, we generated the user interface as much as possible from the RDF-Schema for an annotation. The schema provides the attributes, their hierarchical organisation (modelled as `rdfs:subPropertyOf` and mapped to *tabs*) and a value set (`rdfs:range` of a property) that can be used for menus (small sets) or autocompletion (large sets). The annotation ontology thus provided all information for the GUI, except for ordering of tabs and properties. Intuitive ordering was defined with additional (Prolog) rules.

In our first experiment (section 9.1) we used small hand crafted ontologies and could afford to store the RDF simply as a dynamic predicate `rdf`(*Subject, Predicate, Object*). The second experiment (section 9.2) used much larger triple sets from existing background knowledge which also introduced the problems of integrating these ontologies into a coherent metadata vocabulary, which was resolved using `rdfs:subPropertyOf` mapping rules. The scalability was improved by introducing a formal API for modifying the triple set that managed additional indexing tables, still providing the core query API as `rdf/3`. (Parsia 2001).

Based on this experience and considering our future research plans which involved annotation using terms from large existing ontologies, we decided to invest in a new generation infrastructure to fulfil our requirements:

- *Scalability and expressivity*
  We anticipated on the need to store up to 10 million triples in main memory on commodity hardware in the near future. As we intended to map different annotation vocabularies using `rdfs:subPropertyOf`, we required efficient support reasoning with these mapping relations. These two requirements triggered the development of RDF-DB (chapter 3, Wielemaker et al. 2003b).

- *Modular handling of ontologies*
  When using externally provided large ontologies, it is of course desirable to keep these separated. Still, the developer of mapping relations and (small) additional ontologies wants an integrated overview of all ontologies. Protégé was not capable to deal with modular ontologies, nor with the scale of data we anticipated. This triggered the design of Triple20 (chapter 2, Wielemaker et al. 2005).

- *Reusable GUI framework*
  Still planning for a local GUI implementation for the next generation of search a annotation tools, we assembled Triple20 from reusable components (see section 2.4.1).

Acknowledgments This work was supported by the ICES-KIS project "Multimedia Information Analysis" funded by the Dutch government. We gratefully acknowledge the work of Maurice de Mare on the annotation of the images.

Chapter 10

Thesaurus-based search in large heterogeneous collections

About this chapter This chapter is published at ISWC 2008 (Wielemaker et al. 2008), were it received an honorary mention. We claim that standard Semantic Web reasoning such as Description Logics (OWL-DL) and graph patterns (SPARQL) are insufficient for an important category of Semantic Web applications and that the infrastructure described in part I of this thesis is an adequate platform for prototyping novel techniques to explore RDF graphs. Because web infrastructure support for interactive applications is improving quickly and to enable wider deployment we have chosen to use a web-based interface for this annotation and search prototype rather than the local GUI interface as described in chapter 9.

Our toolset ClioPatria requires all infrastructure of part I, although GUI programming is only used for the development environment features. Michiel Hildebrand, Jacco van Ossenbruggen and Guus Schreiber contributed to this papers as co-authors.

Abstract In cultural heritage, large virtual collections are coming into existence. Such collections contain heterogeneous sets of metadata and vocabulary concepts, originating from multiple sources. In the context of the E-Culture demonstrator we have shown earlier that such virtual collections can be effectively explored with keyword search and semantic clustering. In this paper we describe the design rationale of ClioPatria, the E-Culture open-source software which provides APIs for scalable semantic graph search. The use of ClioPatria's search strategies is illustrated with a realistic use case: searching for "Picasso". We discuss details of scalable graph search, the required OWL reasoning functionalities and show why SPARQL queries are insufficient for solving the search problem.

10.1 Introduction

Traditionally, cultural heritage, image and video collections use proprietary database systems and often their own thesauri and controlled vocabularies to index their collection. Many institutions have made or are making (parts of) their collections available online. Once on the web, each institution, typically, provides access to their own collection. The cultural heritage community now has the ambition to integrate these isolated collections and create a potential source for many new inter-collection relationships. New relations may emerge between objects from different collections, through shared metadata or through relations between the thesauri.

The MultimediaN E-culture project[1] explores the usability of Semantic Web technology to integrate and access museum data in a way that is similar in spirit to the MuseumFinland project (Hyvönen et al. 2005). We focus on providing two types of end-user functionality on top of heterogeneous data with weak domain semantics. First, keyword-based search, as it has become the de-facto standard to access data on the web. Secondly, thesaurus-based annotation for professionals as well as amateurs.

In this paper we formulate the requirements for an infrastructure to provide search and annotation facilities on heterogenous data. We developed ClioPatria[2] as an infrastructure to prototype thesaurus-based search and annotation facilities. We provide the lessons learned in the development of this infrastructure and the construction of end-user prototypes.

This document is organised as follows. In section 10.2 we first take a closer look at our data and describe our requirements by means of a use case. In section section 10.3 we take a closer look at search and what components are required to realise keyword search in a large RDF graph. The ClioPatria infrastructure is described in section section 10.4, together with some illustrations on how ClioPatria can be used. We conclude the paper with a discussion where we position our work in the Semantic Web community.

10.2 Materials and use cases

10.2.1 Metadata and vocabularies

In our case study we collected descriptions of 200,000 objects from six collections annotated with six established thesauri and several proprietary controlled keyword lists, which adds up to 20 million triples. We assume this material is representative for the described domain. Using Semantic Web technology it is possible to unify the data while preserving its richness. The procedure is described elsewhere (Tordai et al. 2007) and summarised here.[3]

The MultimediaN E-Culture demonstrator harvests metadata and vocabularies, but assumes the collection owner provides a link to the actual data object, typically an image of a work such as a painting, a sculpture or a book. When integrating a new collection

[1] `http://e-culture.multimedian.nl`

[2] Open source from `http://e-culture.multimedian.nl/software.html`

[3] The software can be found at `http://sourceforge.net/projects/annocultor`

into the demonstrator we typically receive one or more XML/database dumps containing the metadata and vocabularies of the collection. Thesauri are translated into RDF/OWL, where appropriate with the help of the W3C SKOS format for publishing vocabularies (Miles and Becchofer 2008). The metadata is transformed in a merely syntactic fashion to RDF/OWL triples, thus preserving the original structure and terminology. Next, the metadata schema is mapped to VRA[4], a specialisation of Dublin Core for visual resources. This mapping is realised using the 'dumb-down' principle by means of `rdfs:subPropertyOf` and `rdfs:subClassOf` relations. Subsequently, the metadata goes through an enrichment process in which we process plain-text metadata fields to find matching concepts from thesauri already in the knowledge base. For example, if the `dc:creator` field contains the string *Pablo Picasso*, than we will add the concept `ulan:500009666` from ULAN[5] to the metadata. Most enrichment concerns named entities (people, places) and materials. Finally, the thesauri are aligned using `owl:sameAs` and `skos:exactMatch` relations. For example, the art style *Edo* from a local ethnographic collection was mapped to the same art style in AAT[6] (see the use cases for an example why such mappings are useful). Our current database (April 2008) contains 38,508 `owl:sameAs` and 9,635 `skos:exactMatch` triples and these numbers are growing rapidly.

After this harvesting process we have a graph representing a connected network of works and thesaurus lemmas that provide background knowledge. VRA and SKOS provide —weak— structure and semantics. Underneath, the richness of the original data is still preserved. The data contains many relations that are not covered by VRA or SKOS, such as relations between artists (e.g., ULAN `teacherOf` relations) and between artists and art styles (e.g., relations between AAT art styles and ULAN artists; de Boer et al. 2007). These relations are covered by their original schema. Their diversity and lack of defined semantics make it hard to map them to existing ontologies and provide reasoning based on this mapping. As mentioned, the vocabularies and metadata are harvested onto a single server. This is a natural choice when starting from bulk-conversion of database dumps received from the participating institutes. Furthermore, a single repository allows for exploration of the search and annotation problem without the complexities connected to distributed data. We plan to explore distribution of metadata in future projects.

10.2.2 Use cases

Assume a user is typing in the query "picasso". Despite the fact that the name *Picasso* is reasonably unique in the art world, the user may still have many different intentions with this simple query: a painting by Picasso, a painting of Picasso or the styles Picasso has worked in. Without an elaborate disambiguation process it is impossible to tell in advance.

Figure 10.1 shows part of the results of this query in the MultimediaN demonstrator. We see several clusters of search results. The first cluster contains works from the Picasso

[4] Visual Resource Association, http://www.vraweb.org/projects/vracore4/
[5] Union List of Artist Names is a thesaurus of the Getty foundation
[6] Art & Architecture Thesaurus, another Getty thesaurus

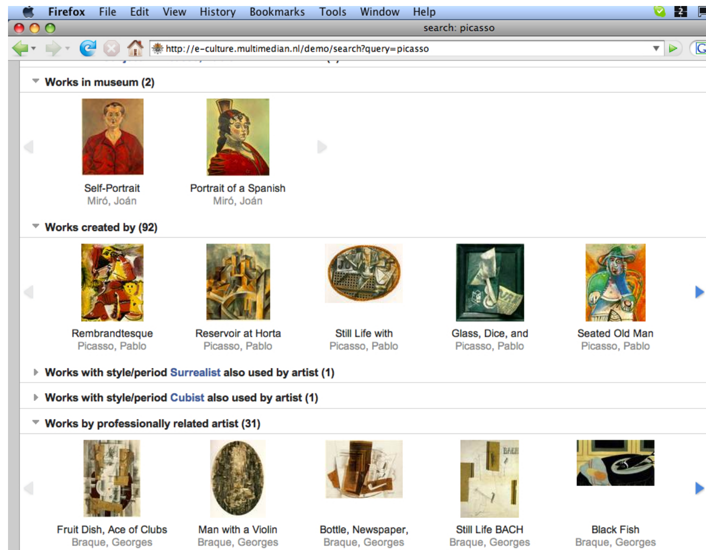


Figure 10.1: Subset of clusters of semantically related search results for query "picasso": works located in the Picasso Museum; works created by Pablo Picasso; works from the art styles used by Pablo Picasso (Cubist, Surrealist, for space reasons only heading shown); works by professional relations of Pablo Picasso (George Braque, colleague Cubist painter).

Museum, the second cluster contains works by Pablo Picasso (only first five hits shown; clicking on the arrow allows the user to inspect all results); clusters of surrealist and cubist paintings (styles that Picasso worked in; not shown for space reasons), and works by George Braque (a prominent fellow Cubist painter, but the works shown are not necessarily cubist). Other clusters include works made from *picasso marble* and works with *Picasso* in the title (including two self portraits). The basic idea is that we are aiming to create clusters of related objects such that the user can afterwards focus on a topic. We have found that even in relatively small collections of 100K objects, users discover interesting results they did not expect. We have termed this type of search tentatively 'post-query disambiguation': in response to a simple keyword query the user gets (in contrast to, for example, Google image search) semantically-grouped results that enable further detailing of the query. It should be pointed out that the knowledge richness of the cultural heritage domain allows this approach to work. Notably typed resources linked to a concept hierarchy and a hierarchy of relations give meaning to the path linking a literal to a target object and allow to abstract this path to arrive at a meaningful number of clusters. Without abstraction, each path is unique and there

is no opportunity for clustering.

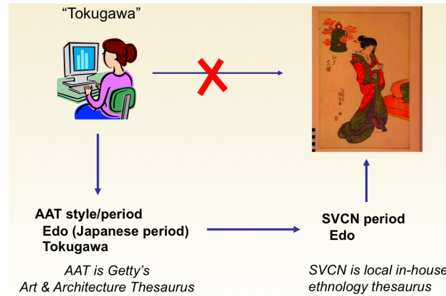


Figure 10.2: A user searches for "tokugawa". The Japanese painting in the top right matches this query, but is indexed with a thesaurus that does not contain the synonym "Tokugawa" for this Japanese style. Through a "same-as" link with another thesaurus that does contain this label, the semantic match can be made.

Another typical use case for search concerns the exploitation of vocabulary alignments. The Holy Grail of the unified cultural-heritage thesaurus does not exist and many collection owners have their own home-grown variants. Consider the situation in figure 10.2, which is based on real-life data. A user is searching for "tokugawa". This Japanese term has actually two major meanings in the heritage domain: it is the name of a 19th century shogun and it is a synonym for the Edo style period. Assume for a moment that the user is interested in finding works of the latter type. The Dutch ethnographic museum in Leiden actually has works in this style in its digital collection, such as the work shown in the top-right corner. However, the Dutch ethnographic thesaurus SVCN, which is being used by the museum for indexing purposes, only contains the label "Edo" for this style. Fortunately, another thesaurus in our collection, the aforementioned AAT, does contain the same concept with the alternative label "Tokugawa". In the harvesting process we learned this equivalence link (quite straightforward: both are Japanese styles with matching preferred labels). The objective of our graph search is to enable to make such matches.

Although this is actually an almost trivial alignment, it is still extremely useful. The cultural-heritage world (like any knowledge rich domain) is full of such small local terminology differences. Multilingual differences should also be taken into consideration here. If semantic-web technologies can help making such matches, there is a definite added value for users.

10.3 Required methods and components

In this section we study the methods and components we need to realise the keyword search described above. Our experiments indicate that meaningful matches between keyword and target often involve chains of up to about five relations. At this distance there is a potentially huge set of possible targets. The targets can be organised by rating based on semantics or statistics and by clustering based on the graph pattern linking a literal to the target. We discuss three possible approaches: querying using a fixed set of graph patterns, completely unconstrained graph search and best-first exploration of the graph.

10.3.1 Using a set of fixed queries

A cluster as shown in figure 10.1 is naturally represented as a graph pattern as found in many Semantic Web query languages. If we can enumerate all possible meaningful patterns of properties that link literals to targets we reduce the search process to finding instances of all these graph patterns. This would be a typical approach in Semantic Web applications such as DBin (Tummarello et al. 2006). This approach is, however, not feasible for highly heterogenous data sets. Our current data contains over 600 properties, most of which do not have a well defined meaning (e.g., `detailOf`, `cooperatedWith`, `usesStyle`). If we combine this with our observation that is it quite common to find valuable results at 4 or even 5 steps from the initial keywords, we have to evaluate a very large number of possible patterns. To a domain expert, it is obvious that the combination of `cooperatedWith` and `hasStyle` can be meaningful while the combination `bornIn` and `diedIn` (i.e., A is related to B because A died in P, where B was born) is generally meaningless, but the set of possible combinations to consider is too large for a human. Automatic rating of this type of relation pattern is, as far as we know, not feasible. Even if the above is possible, new collections and vocabularies often come with new properties, which must all be considered in combination to the already created patterns.

10.3.2 Using graph exploration

Another approach is to explore the graph, looking for targets that have, often indirectly, a property with matching literal. This implies we search the graph from *Object* to *Subject* over arbitrary properties, including triples entailed by `owl:inverseOf` and `owl:SymmetricProperty`. We examine the scalability issues using unconstrained graph patterns, after which we examine an iterative approach.

Considering a triple store that provides reasoning over `owl:inverseOf` and `owl:SymmetricProperty` it is easy to express an arbitrary path from a literal to a target object with a fixed length. The total result set can be expressed as a union of all patterns of fixed length up to (say) distance 5. Table 10.1 provides the statistics for some typical keywords at distances 3 and 5. The table shows total visited and unique results for both visited nodes and targets found which indicates that the graph contains a large number of alternative paths and the implementation must deal with these during the graph exploration to reduce the

amount of work. Even without considering the required post-processing to rank and cluster the results it is clear that we cannot obtain interactive response times for many queries using blind graph exploration.

Keyword	**Dist.**	**Literals**	**Nodes**		**Targets**		**Time**
			Visited	**Unique**	**Visited**	**Unique**	**(sec.)**
tokugawa	3	21	1,346	1,228	913	898	0.02
steen	3	1,070	21,974	7,897	11,305	3,658	0.59
picasso	3	85	9,703	2,399	2,626	464	0.26
rembrandt	3	720	189,611	9,501	141,929	4,292	3.83
impressionism	3	45	7,142	2,573	3,003	1,047	0.13
amsterdam	3	6,853	1,327,797	421,304	681,055	142,723	39.77
tokugawa	5	21	11,382	2,432	7,407	995	0.42
steen	5	1,070	1,068,045	54,355	645,779	32,418	19.42
picasso	5	85	919,231	34,060	228,019	6,911	18.76
rembrandt	5	720	16,644,356	65,508	12,433,448	34,941	261.39
impressionism	5	45	868,941	50,208	256,587	11,668	18.50
amsterdam	5	6,853	37,578,731	512,027	23,817,630	164,763	620.82

Table 10.1: Statistics for exploring the search graph for exactly *Distance* steps (triples) from a set of literals matching *Keyword*. *Literals* is the number of literals holding a word with the same stem as *Keyword*; *Nodes* is the number of nodes explored and *Targets* is the number of target objects found. *Time* is measured on an Intel Core duo X6800.

Fortunately, a query system that aims at human users only needs to produce the most promising results. This can be achieved by introducing a distance measure and doing *best-first* search until our resources are exhausted *(anytime algorithm)* or we have a sufficient number of results. The details of the distance measure are still subject of research (Rocha et al. 2004), but not considered vital to the architectural arguments in this article. The complete search and clustering algorithm is given in figure 10.3. In our experience, the main loop requires about 1,000 iterations to obtain a reasonable set of results, which leads to acceptable performance when the loop is pushed down to the triple store layer.

10.3.3 Term search

The combination of best-first graph exploration with semantic clustering, as described in figure 10.3, works well for 'post-query' disambiguation of results in exploratory search tasks. It is, however, less suited for quickly selecting a known thesaurus term. The latter is often needed in semantic annotation and 'pre-query' disambiguation search tasks. For such tasks we rely on the proven *autocompletion* technique, which allows us to quickly find resources related to the prefix of a label or a word inside a label, organise the results (e.g., organise cities by country) and provide sufficient context (e.g., date of birth and death of a person).

1. Find literals that contain the same stem as the keywords, rate them on minimal edit distance (short literal) or frequency (long literal) and sort them on the rating to form the initial *agenda*
2. Until satisfied or empty *agenda*, do
    (a) Take highest ranked value from *agenda* as O. Find `rdf`(*S,P,O*) terms. Rank the found S on the ranking of O, depending on P. If P is a subProperty of `owl:sameAs`, the ranking of S is the same as O. If S is already in the result set, combine their values using $R = 1 - ((1 - R_1) \times (1 - R_2))$. If S is new, insert it into *agenda*, else reschedule it in the agenda.
    (b) If S is a target, add it to the *targets*. Note that we must consider `rdf`(*O,IP,S*) if there is an `inverseOf`(*P,IP*) or P is symmetric.
3. Prune resulting graph from branches that do not end in a target.
4. Smush resources linked by `owl:sameAs`, keeping the resource with the highest number of links.
5. Cluster the results
    (a) Abstract all properties to their VRA or SKOS root property (if possible).
    (b) Abstract resources to their class, except for instances of `skos:Concept` and the top-10 ranked instances.
    (c) Place all triples in the abstract graph. Form (RDF) Bags of resources that match to an abstracted resource and use the lowest common ancestor for multiple properties linking two bags of resources.
6. Complete the nodes in the graph with label information for proper presentation.

Figure 10.3: Best first graph search and clustering algorithm

Often results can be limited to a sub-hierarchy of a thesaurus, expressed as an extra constraint using the transitive `skos:broader` property. Although the exact technique differs, the technical requirements to realise this type of search are similar to the keyword search described above.

10.3.4 Literal matching

Similar to document retrieval, we start our search from a rated list of literals that contain words with the same stem as the searched keyword. Unlike document retrieval systems such as Swoogle (Ding et al. 2004) or Sindice (Tummarello et al. 2007), we are not primarily interested in which RDF documents the matching literals occur, but which semantically related target concepts are connected to them. Note that term search (section 10.3.3) requires finding literals from the prefix of a contained word that is sufficiently fast to be usable in autocompletion interfaces (see also Bast and Weber 2007). RDF literal indexing is described in section 7.3.2.

10.3.5 Using SPARQL

If possible, we would like our search software to connect to an arbitrary SPARQL endpoint. Considering the *fixed query* approach, each pattern is naturally mapped onto a SPARQL graph pattern. *Unconstrained graph search* is easily expressed too. Expressed as a CONSTRUCT query, the query engine can return a minimal graph without duplicate paths.

Unfortunately, both approaches proved to be infeasible implementation strategies. The best-first graph exploration requires one (trivial) SPARQL query to find the neighbours of the next node in the *agenda* for each iteration to update the agenda and to decide on the next node to explore. Latency and volume of data transfer make this infeasible when using a remote triple store.

The reasoning for clustering based on the property hierarchy cannot be expressed in SPARQL, but given the size and stability of the property hierarchy we can transfer the entire hierarchy to the client and use local reasoning. After obtaining the clustered results, the results need to be enriched with domain specific key information (title and creator) before they can be presented to the user. Requesting the same information from a large collection of resources can be realised using a rather inelegant query as illustrated in figure 10.4.

```
SELECT ?l1 ?l2 ...
WHERE { { ulan:artists1 rdfs:label ?l1 } UNION
        { ulan:artists2 rdfs:label ?l2 } UNION
        ...
      }
```

Figure 10.4: Query the labels of many resources

We conclude that SPARQL is inadequate for adaptive graph exploration algorithms, incapable of expressing lowest common parent problems and impractical for enriching computed result sets. Finally, regular expression literal matching cannot support match on stem. Prefix and case insensitive search for contained word can be expressed. Ignoring diacritic marks during matching is generally required when dealing with text from multiple languages using multiple scripts, but is not supported by the SPARQL regular expression syntax.[7]

10.3.6 Summary of requirements for search

- Obtain rated list of literals from stem and prefix of contained words.
- The OWL primitives `owl:inverseOf` and `owl:SymmetricProperty` are used to specify which relations are searched in both directions.
- Entailment over `owl:TransitiveProperty` is used to limit results to a particular hierarchy in a SKOS thesaurus.
- Entailment over `owl:sameAs` for term search.
- The best-first graph exploration must be tightly connected to the triple store to enable fast exploration of the graph.

[7] Some regex variations support diacritic mark matching. For example CQP `http://www.ims.uni-stuttgart.de/projekte/CorpusWorkbench/`

- Reasoning with types as well as the class, concept and property hierarchy. This includes finding the lowest common parent of a set of resources in these hierarchies. Note that none of these form strict trees (i.e., the relations form cycles and nodes have multiple parents).

10.4 The ClioPatria search and annotation toolkit

We have realised the functionality described in the previous section on top of the SWI-Prolog[8] web and Semantic Web libraries (chapter 7, Wielemaker et al. 2008; Wielemaker et al. 2007) that are distributed as standard packages of SWI-Prolog. This platform provides a scalable in-core RDF triple store (chapter 3, Wielemaker et al. 2003b) and a multi-threaded HTTP server library (section 7.4). ClioPatria is the name of the reusable core of the E-culture demonstrator, the architecture of which is illustrated in figure 10.5. First, we summarise some of the main features of ClioPatria.

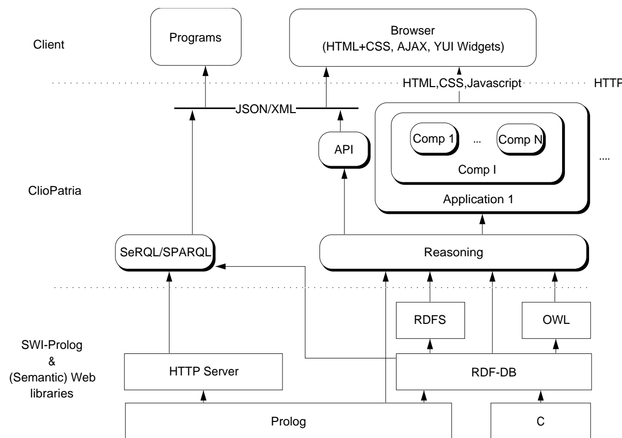


Figure 10.5: Overall architecture of the ClioPatria server

- Running on a Intel core duo X6800@2.93GHz, 8Gb, 64-bit Linux it takes 120 seconds elapsed time to load the 20 million triples. The server requires 4.3Gb memory for 20 million triples (2.3Gb in 32-bit mode). Time and space requirements grow practically linear in the amount of triples.

- The store provides safe persistency and maintenance of provenance and change history based on a (documented) proprietary file format.

[8] http://www.swi-prolog.org

- Different operations require a different amount of entailment reasoning. Notably deleting and modifying triples complicates maintenance of the pre-computed entailment. Therefore, reasoning is as much as possible based on backward chaining, a paradigm that fits naturally with Prolog's search driven programming.

10.4.1 Client-server architecture

In contrast to client-only architectures such as Simile's Exhibit (Huynh et al. 2007), ClioPatria has a client-server architecture. The core functionality is provided as HTTP APIs by the server. The results are served as presentation neutral data objects and can thus be combined with different presentation and interaction strategies. Within ClioPatria, the APIs are used by its web applications. In addition, the APIs can be used by third party applications to create mashups.

The ClioPatria toolkit contains web applications for search and annotation. The end-user applications are a combination of server side generated HTML and client side JavaScript interface widgets. The generated HTML contains the layout of the application page including place markers for the interface widgets. The interface widgets are loaded into the page on the client side and populate themselves by requesting data through one of the APIs.

The reusability of the design is demonstrated by a variety of applications that use ClioPatria, either as central server or as service within a larger application. Besides for the MultimediaN E-Culture demonstrator[9] for which ClioPatria was developed, it is currently in use by the following projects. The K-Space European Network of Excellence is using ClioPatria to search news.[10] At the time of writing Europeana[11] is setting up ClioPatria as a demonstrator to provide multilingual access to a large collection of very divers cultural heritage data. The ClioPatria API provided by the E-Culture Project is also used by the CATCH/CHIP project Tour Wizard that won the 3rd prize at the Semantic Web Challenge of 2007. For the semantic search functionality CHIP uses the web services provided by the ClioPatria API.

10.4.2 Output formats

The server provides two types of presentation oriented output routines. *Components* are Prolog grammar rules that define reusable parts of a page. A component produces HTML and optionally posts requirements for CSS and JavaScript. For example, the component `localview` emits an HTML `div` and requests the JavaScript code that realises the detailed view of a single resource using an AJAX widget. Components can embed each other. *Applications* produce an entire HTML page that largely consists of configured components. HTML pages, and therefore applications, cannot be nested. The HTML libraries define a resource infrastructure that tracks requests for CSS and JavaScript resources and uses this together with declarations on CSS and JavaScript dependencies to complete the HTML head information, turning components into clean modular entities.

[9] http://e-culture.multimedian.nl/demo/search
[10] http://newsml.cwi.nl/explore/search
[11] http://www.europeana.eu/

Client side presentation and interaction is realised by JavaScript interface widgets. The widgets are built on top of the YAHOO! User Interface (YUI) library.[12] ClioPatria contains widgets for autocompletion, a search result viewer, a detailed view on a single resource, and widgets for semantic annotation fields. The result viewer can visualise data in thumbnail clusters, a geographical map, Simile Exhibit, Simile Timeline and a Graphviz[13] graph visualisation.

The traditional language of choice for exchanging data over the network is XML. However, for web applications based on AJAX interaction an obvious alternative is provided by JSON (*JavaScript Object Notation*[14]), as this is processed natively by JavaScript capable browsers. JSON targets at object serialisation rather than document serialisation and is fully based on UNICODE. James Clark, author of the SP SGML parser and involved in many SGML and XML related developments acknowledges the value of JSON.[15] JSON is easy to generate and parse, which is illustrated by the fact that the Prolog JSON library, providing bi-directional translation between Prolog terms and JSON text counts only 792 lines. In addition the community is developing standard representations, such as the SPARQL result format (Clark et al. 2007).

10.4.3 Web services provided by ClioPatria (API)

ClioPatria provides programmatic access to the RDF data via several web services[16]. The query API provides standardised access to the data via the SeRQL and SPARQL. As we have shown in section 10.3 such a standard query API is not sufficient to provide the intended keyword search functionality. Therefore, ClioPatria provides an additional search API for keyword-based access to the RDF data. In addition, ClioPatria provides APIs to get resource-specific information, update the triple store and cache media items. In this paper we only discuss the query and search API in more detail.

10.4.3.1 Query API

The SeRQL/SPARQL library provides a Semantic Web query interface that is compatible with Sesame (Broekstra et al. 2002) and provides open and standardised access to the RDF data stored in ClioPatria.

Both SeRQL and SPARQL are translated into a Prolog query that relies on the `rdf`(*S,P,O*) predicate provided by SWI-Prolog's RDF library and on auxiliary predicates that realise functions and filters defined by SeRQL and SPARQL. Conjunctions of `rdf/3` statements and filter expressions are optimised through reordering based on statistical information provided by the RDF library chapter 4 (Wielemaker 2005). Finally, the query is executed and the result

[12] `http://developer.yahoo.com/yui/`
[13] `http://www.graphviz.org/`
[14] `http://www.json.org/`
[15] `http://blog.jclark.com/2007/04/xml-and-json.html`
[16] `http://e-culture.multimedian.nl/demo/doc/`

is handed to an output routine that emits tables and graphs in various formats specified by both SeRQL and SPARQL.

10.4.3.2 Search API

The search API provides services for graph search (figure 10.3) and term search (section 10.3.3). Both services return their result as a JSON object (using the serialisation for SPARQL SELECT queries, Clark et al. 2007). Both services can be configured with several parameters. General search API parameters are:

- **query**(*string* | URI): the search query.
- **filter**(`false` | *Filter*): constrains the results to match a combination of *Filter* primitives, typically OWL class descriptions that limit the results to instances that satisfy these descriptions. Additional syntax restricts results to resources used as values of properties of instances of a specific class.
- **groupBy**(`false` | `path` | *Property*): if `path`, cluster results by the abstracted path linking query to target. If a property is given, group the result by the value on the given property.
- **sort**(`path_length` | `score` | *Property*): Sort the results on path-length, semantic distance or the value of *Property*.
- **info**(`false` | *PropertyList*): augment the result with the given properties and their values. Examples are `skos:prefLabel`, `foaf:depicts` and `dc:creator`.
- **sameas**(*Boolean*): smushes equivalent resources, as defined by `owl:sameAs` or `skos:exactMatch` into a single resource.

Consider the use case discussed in section 10.2.2. Clustered results that are semantically related to keyword "picasso" can be retrieved through the graph search API with the HTTP request below. The `vra:Work` filter limits the results to museum objects. The expression `view=thumbnail` is a shorthand for `info = ["image":"thumbnail", "title":"vra:creator", "subtitle":"dc:creator"]`.

```
/api/search?query=picasso&filter=vra:Work&groupBy=path&view=thumbnail
```

Parameters specific to the graph search API are:

- **view**(`thumbnail` | `map` | `timeline` | `graph` | `exhibit`): shorthands for specific property lists of the `info` parameter.
- **abstract**(*Boolean*): enables the abstraction of the graph search paths over `rdfs:subClassOf` and `rdfs:subPropertyOf`, reducing the number of clusters.

- **bagify**(*Boolean*): puts (abstracted) resources of the same class with the same (abstracted) relations to the rest of the graph in an RDF bag. For example, convert a set of triples linking a painter over various sub properties of `dc:creator` to multiple instances of `vra:Work` into an RDF bag of works and a single triple linking the painter as `dc:creator` to this bag.

- **steps**(*Integer*): limits the graph exploration to expand no more than *Integer* nodes.

- **threshold**(*0.0..1.0*): cuts off the graph exploration at the given semantic distance (1.0: close; 0.0 infinitely far).

For annotation we can use the term search API to suggest terms for a particular annotation field. For example, suppose a user has typed the prefix “pari” in a location annotation field that only allows European locations. We can request matching suggestions by using the URI below, filtering the results to resources that can be reached from `tgn:Europe` using `skos:broader` transitively:

```
/api/autocomplete?query=pari&match=prefix&sort=rdfs:label&
filter={"reachable":{"relation":"skos:broader","value":"tgn:Europe"}}
```

Parameters specific to the term search API are:

- **match**(`prefix|stem|exact`): defines how the syntactic matching of literals is performed. Autocompletion, for example, requires `prefix` match.

- **property**(*Property*, *0.0..1.0*): is a list of RDF property-score pairs which define the values that are used for literal matching. The score indicates preference of the used literal in case a URI is found by multiple labels. Typically preferred labels are chosen before alternative labels.

- **preferred**(`skos:inScheme`, URI): in case URIs are smushed the information of the URI from the preferred thesaurus is used for augmentation and organisation.

- **compound**(*Boolean*): if `true`, filter results to those where the query matches the information returned by the `info` parameter. For example, a compound query *paris, texas* can be matched in two parts against a) the label of the place *Paris* and b) the label of the state in which *Paris* is located.

10.5 Discussion and conclusion

In this paper we analysed the requirements for searching in large, heterogeneous collections with many relations, many of which have no formal semantics. We presented the ClioPatria software architecture we used to explore this topic. Three characteristics of ClioPatria have proved to be a frequent source of discussion: the non-standard API, the central main memory store model and the lack of full OWL/DL support.

10.5.0.3 API standardisation

First, ClioPatria's architecture is based on various client-side JavaScript Web applications around a server-side Prolog-based reasoning engine and triple store. As discussed in this paper, the server functionality required by the Web clients can not be provided by an off-the-shelf SPARQL endpoint. This makes it hard for Semantic Web developers of other projects to deploy our Web applications on top of their own SPARQL-based triple stores. We acknowledge the need for standardised APIs in this area. We hope that the requirements discussed in this paper provide a good starting point to develop the next generation Semantic Web APIs that go beyond the traditional database-like query functionality currently supported by SPARQL.

10.5.0.4 Central, main memory storage model

From a data-storage perspective, the current ClioPatria architecture assumes images and other annotated resources to reside on the Web. All metadata being searched, however, is assumed to reside in main memory in a central, server-side triple store. We are currently using this setup with a 20M triples dataset, and are confident that our current approach will easily scale up to 300M triples on modern hardware (64Gb main memory). Our central main memory model will not scale, however, to the multi-billion triple sets supported by other state-of-the-art triple stores. For future work, we are planning to investigate to what extent we can move to disk-based or, given the distributed nature of the organisations in our domain, distributed storage strategies without giving up the key search functionalities of our current implementation. Distribution of the entire RDF graph is non-trivial. For example, in the keyword search, the paths in the RDF graph from the matching literals to the target resources tend to be unpredictable, varying highly with the types of the resources associated with the matching literals and the type of the target resources. Implementing a fast, semi-random graph walk in a distributed fashion will likely be a significant challenge. As another example, interface components such as a Web-based autocompletion Widget are based on the assumption that a client Web-application may request autocompletion suggestions from a single server, with response times in the 200ms range. Realising sufficiently fast responses from this server without the server having a local index of all literals that are potential suggestion candidates will also be challenging. Distributing carefully selected parts of the RDF graph, however, could be a more promising option. In our current datasets for example, the subgraphs with geographical information are both huge and connected to the rest of the graph in a limited and predictable fashion. Shipping such graphs to dedicated servers might be doable with only minor modifications to the search algorithms performed by the main server. This is a topic we need to address in future work.

10.5.0.5 Partial OWL reasoning

From a reasoning perspective, ClioPatria does not provide traditional OWL-DL support. First of all, the heterogeneous and open nature of our metadata repositories ensures that even

when the individual data files loaded are in OWL-DL, their combination will most likely not be. Typical DL violations in this domain are properties being used as a data property with name strings in one collection, and as an object property with URIs pointing to a biographical thesaurus such as ULAN in the other; or `rdfs:label` properties being used as an annotation property in the schema of one collection and as a data property on the instances of another collection. We believe that OWL-DL is a powerful and expressive subset of OWL for closed domains where all data is controlled by a single organisation. It has proved, however, to be unrealistic to use OWL DL for our open, heterogenous Semantic Web application where multiple organisations can independently contribute to the data set.

Secondly, our application requires the triple store to be able to flexibly turn on and off certain types of OWL reasoning on a per-query basis. For example, there are multiple URIs in our dataset, from different data sources, representing the Dutch painter *Rembrandt van Rijn*. Ideally, our vocabulary mapping tools have detected this and have all these URIs mapped to one another using `owl:sameAs`. For an end-user interested in viewing all information available on Rembrandt, it is likely beneficial to have the system perform `owl:sameAs` reasoning and present all information related to Rembrandt in a single interface, smushing all different URIs onto one. For an expert end-user annotating an artwork being painted by Rembrandt the situation is different. When selecting the corresponding entry from a biographical thesaurus, the expert is probably interested into which vocabulary source the URI is pointing, and how entries in other vocabularies differ from the selected one. This requires the system to largely ignore the traditional `owl:sameAs` semantics, present all triples associated with the different URIs separately, along with the associated provenance information. This type of ad-hoc turning on and off of specific OWL reasoning is, to our knowledge, not supported by any off-the-shelf SPARQL endpoint, but crucial in all realistic multi-thesauri Semantic Web applications.

Thirdly, we found that our application requirements seldomly rely on extensive subsumption or other typical OWL reasoning. In the weighted graph exploration we basically only consider the graph structure and ignore most of the underlying semantics, with only a few notable exceptions. Results are improved by assigning equivalence relations such as `owl:sameAs` and `skos:exactMatch` the highest weight of 1.0. We search the graph in only one direction, the exception being properties being declared as an `owl:SymmetricProperty`. In case of properties having an `owl:inverseOf`, we traverse the graph as we would have if all “virtual” inverse triples were materialised. Finally, we use a simple form of subsumption reasoning over the property and class hierarchy when presenting results to abstract from the many small differences in the schemas underlying the different search results.

10.5.0.6 Conclusion

Our conclusion is that knowledge rich domains such as cultural heritage fit well with Semantic Web technology. This is because of a) the clear practical needs this domain has for integrating information from heterogeneous sources, and b) its long tradition with semantic

annotations using controlled vocabularies and thesauri. We strongly feel that studying the real application needs of users working in such domains in terms of their search and reasoning requirements will move ahead the state of the art in Semantic Web research significantly.

Experience using the Prolog web infrastructure ClioPatria evolved from the SeRQL server described in chapter 4. The SeRQL server contained the RDF database (chapter 3) and the multi-threaded HTTP server (section 7.4.2) with basic administrative facilities such as user management and viewing server statistics. The infrastructure proved a good basis for the rapid development of a prototype annotation and search engine. Rules were implemented based on Prolog and `rdf/3`. Presentation used the HTML generation library (section 7.2.2.1). The prototype annotation tool used HTML forms and Java applets for navigating the concept hierarchy. As the project evolved, this design has been refined:

- RDF *storage*
  Literal indexing has been refined to include full text search (section 3.4.1.1). Support for reliable updates to the RDF store as required by annotation has been added. See section 3.4.2 and section 3.4.3.

- *User interface*
  The initial user interface based on classical HTML and Java applets was not satisfactory. Java applets integrate poorly in the HTML page and the development cycle is rather clumsy and error prone. It was first replaced by in-house JavaScript using AJAX technology. Poor browser compatibility and the desire to reuse presentation components made us switch to an externally developed AJAX widget library (YUI). This approach changes the server from producing HTML for presentation towards returning results as JSON data objects. This approach has two clear advantages: it enables interactivity required for e.g., autocompletion and it reuses externally developed widgets, such as an image carrousel for scrolling through a set of thumbnails.

  The current approach also has drawbacks. It implies programming in two languages which makes it much harder to track the exact location of code that produces a page. AJAX-based pages cooperate poorly with native browser history navigation and linking to a page.

- *Modularity*
  A growing number of HTTP paths served, resource dependencies between JavaScript and CSS files as well as multiple developers asked for better modularisation of the server. This resulted in the HTTP dispatch library (section 7.4.2.1) and tracking web page resource dependencies as described in section 7.5.1.

The current infrastructure supports semantic annotation and search as described in this paper satisfactory. We identified scalability aspects that will need to be addressed soon. First, identity mapping (`owl:sameAs`) requires low-level support in the RDF database to speed up and simplify application code that needs identity reasoning. As described in section 10.5.0.5,

we want to enable or disable `owl:sameAs` reasoning on a per-query basis and therefore *smushing* identical URIs to a single URI is not an option. Second, restoring the persistent database will soon become a bottleneck, notably for development. We will consider two approaches to solve that. One is to exploit multiple cores for reloading, which currently scales poorly due to frequent synchronisation in the RDF store and Prolog atom table. As an alternative, we will consider managing part of the Prolog/RDF database in a dedicated memory area that can be mapped to a file. This technique can realise short startup times, but the implementation is more complicated and vulnerable to changes in the program that make old states unusable. Combining both approaches can provide good results. Third, the read-write lock based mechanism to update the RDF database (section 3.4.2) might prove unusable for interactive annotation because continuous read access delays write requests too much. A more fine-grained locking mechanism can fix this as well as improve concurrency when restoring the persistent RDF database.

Chapter 11

Conclusions

Part I of this thesis describes language extensions and libraries for Prolog to deal with knowledge-intensive interactive (web) applications, while part II describes a selection of applications that use this infrastructure.

The libraries and extensions themselves are a mixture of novel and established ideas from other Prolog implementations or other programming environments, integrated in a coherent environment. **XPCE** and especially the way XPCE is connected to Prolog was novel. We implemented the basic interface in the mid-80's when there was no related work available. Achieving dedicated support for **RDF** in Prolog using a main-memory store that pushed the scalability by exploiting RDF-specific features was also a novel development. The same holds for **query optimisation** for main-memory-based RDF stores. Although the related problems of literal reordering and database join optimisation are well described in the literature, the long conjunctions that result from the primitive RDF model posed new problems. When we started implementing **concurrency**, a lot of research had been done in adding concurrency to Prolog. However, few systems provided a usable implementation of multi-threading and those that did were lacking facilities on which we depended, such as XPCE and our RDF library. We proposed a practical API, described solutions to complications such as atom garbage collection and demonstrated that multi-threading can be added to Prolog with limited effort and acceptable loss of performance. Adding support for **web services** by means of supporting HTTP and the web document formats makes well established standards available in Prolog. No single part of this process itself is particularly complicated, however the value of our work in this area is in presenting an integrated infrastructure for web services and showing that it works as demonstrated by ClioPatria.

Where (especially academic) language and library design often uses a *technology push* model, we use a model where this activity is closely embedded in the design and implementation of prototypes that serve a research goal that is not primarily related to infrastructure. For example, the primary goal of the MIA project described in chapter 9 was to study annotation of the subject of multi-media objects (photos, paintings, etc.). The secondary goal was to study the applicability of the emerging Semantic Web standards (RDF and RDFS) for annotation and search and finally, the ternary goal was to examine how we could deploy RDF

for knowledge representation inside an application and what implications that has for interactive graphical applications. Reasoning and interactivity based on the RDF triple model is the basis of chapter 2 about Triple20.

In these conclusions we first revisit the research questions, followed by an evaluation of the overall infrastructure based on our current experience. We conclude with challenges for the near future.

11.1 The research questions revisited

How to represent knowledge for interactive applications in Prolog? Frame-based knowledge representations are popular if knowledge has to be examined and manipulated using a graphical interface. Frame languages provide natural graphical representations and the MVC design pattern provides proven technology to interact with the knowledge. Before the existence of RDF we used a variety of proprietary representations for the knowledge representation. RDF provides us with a widely accepted standard model that fits naturally on the Prolog relational model, where RDF graph expressions map directly to conjunctions of `rdf/3` literals. The RDF standard comes with a serialisation format (RDF/XML) for storing and exchanging knowledge with other RDF-aware applications.

If RDF, with the languages and schemas defined on top of it (e.g., RDFS, OWL, Dublin Core, SKOS), is used for storing knowledge inside the application it provides a standardised interpretation of the knowledge. The RDF mapping relations (e.g., `owl:sameAs`, `rdfs:subPropertyOf`) can be used to integrate knowledge. Again, this feature is both valuable for exchange with other applications and for integrating heterogeneous knowledge inside one application. This approach has been applied first in chapter 9, then in chapter 2 where we provide further motivation and finally in chapter 10.

In section 9.3 we formulated the requirement to handle up to 10 million RDF triples. The demonstrator described in chapter 10 contains 20M triples. This scale is enough to store sufficiently rich background knowledge and sufficiently large collections for realistic experiments with search and annotation. Section 3.4 describes a state-of-the-art main-memory store that is capable of supporting this scale (section 3.6). In fact, the store currently supports upto 300 million triples using 64Gb memory.

Query languages such as SPARQL, but also reasoning inside ClioPatria needs to solve conjunctions of RDF literals. Done naively, this can result in poor performance. Section 4.7 shows how these conjunctions can be ordered optimally at low cost. This is used as a query optimiser in the SeRQL/SPARQL server and for dynamic optimisation of Prolog conjunctions of `rdf/3` literals inside ClioPatria.

In section 10.3 we formulate requirements for search in heterogeneous collections of meta-data and collections. In section 10.3.3 we describe best-first graph exploration and semantic clustering as a possible solution for search in this type of data.

How to support web applications in Prolog? Chapter 7 discusses the representation of RDF, HTML and XML documents in Prolog, as well as how the HTTP protocol can be sup-

ported. We have described the representation of HTML and XML as a Prolog term. The representation of an XML document as a Prolog term can only be simple and compact if the Prolog implementation supports UNICODE, atoms of unlimited length and atom garbage collection (section 7.6.2).

We have described how standard compliant HTML can be generated from Prolog using a generative grammar (section 7.2.2.1). This approach guarantees compliant HTML starting from Prolog data structures, including proper element structuring and escape sequences dealing with special characters. Our approach supports *modular* generation of HTML pages. This modularity is realised using embedded calls to rules from a document term, automatic management of resources needed by an HTML page such as JavaScript and CSS files and allowing for non-determinism in the generator. Non-determinism allows for alternative output controlled by conditions embedded at natural places in the code instead of being forced to evaluate conditions before emitting HTML.

Web services run on a server and need to serve many clients. Applications based on our main memory RDF store need a significant startup time and use considerable memory resources. This, together with the availability of multi-core hardware demands the support for multi-threading in Prolog. Chapter 6 describes a pragmatic API for adding threads to Prolog that has become the starting point for an ISO standard (Moura et al. 2008). Particularly considering web services, the thread API performs well and has hardly any noticeable implications for the programmer.

How to support graphical applications in Prolog? Chapter 5 proposes a mechanism to access and extend an object oriented system from Prolog, which has been implemented with XPCE. XPCE has been the enabling factor in many applications developed in SWI-Prolog. In this thesis we described the MIA tool (chapter 9) and the Triple20 ontology editor (chapter 2). The approach followed to connect Prolog to XPCE realises a small interface to object oriented systems that have minimal reflexive capabilities (section 5.3) and a portable mechanism that supports object oriented programming in Prolog that extends the core OO system (section 5.4). This architecture, together with section 2.4.1 which describes rule-based specification of appearance and behaviour is our answer to research questions 3a and 3b.

Recently the use of a web-browser for the GUI comes into view. This approach has become feasible since to the introduction of JavaScript widget libraries such as YUI based on 'AJAX' technology. We use this technology in ClioPatria after extending the web service support with JSON (section 7.5). JavaScript libraries for vector graphics are available and used by ClioPatria to display search results as a graph. JavaScript libraries for interactive vector graphics are emerging.

The discussion on providing graphics to Prolog applications is continued in the discussion (section 11.3.1).

Connecting knowledge to the GUI is the subject of research question 3c and, because we use RDF for knowledge representation, can be translated into the question how RDF can be connected to the GUI. The RDF model is often of too low level to apply the MVC model

directly for connecting the knowledge to the interface objects. This problem can be solved using *mediators* (section 2.4). Mediators play the same role as a high-level knowledge representation that is based on the tasks and structure used in the interface. Interface components can independently design mediators to support their specific task and visualisation. In other words, mediators do not harm the well understood and stable semantics of the RDF knowledge base, while they realise the required high-level view on the knowledge. Chapter 2 shows that this design can be used to browse and edit large RDF models efficiently.

11.2 Architectures

We have produced two architectures that bring the infrastructure together. Section 2.4 describe the Triple20 architecture that resulted from the lessons learned from the MIA project (section 9.3). The architecture of Triple20 was designed for our next generation of interactive tools for ontology management, search and annotation. Project focus and improving web infrastructure for interactive applications made us switch from highly interactive local GUI applications to a less interactive web-based application for annotation and search. As a result, the Triple20 architecture has not been evaluated in a second prototype.

Figure 10.5 presents our architecture for supporting web applications, a subset of which has also been used for the Prolog literate programming environment PlDoc (chapter 8). ClioPatria has been used in a number of projects (section 10.4.1). Despite the rather immature state of many parts of the ClioPatria code base, the toolkit has proven to be a productive platform that is fairly easy to master. An important reason for this is that the toolkit has a sound basis in the RDF infrastructure, the HTTP services and the modular generation of HTML with AJAX components. Or, to put it differently, it is based on sound and well understood standards.

On top of this stable infrastructure are libraries that provide semantic search facilities and interface components with a varying level of maturity. Because of the modular nature of interface components and because HTTP *locations* (paths) make it easy to support different versions of services in the same web-server, new applications and components can be developed using the 'bazaar'-model,[1] in which multiple developers can cooperate without much coordination.

11.3 Discussion: evaluation of our infrastructure

Some of the material presented in this section results from discussions, face-to-face, by private E-mail, in the SWI-Prolog mailinglist or on the `comp.lang.prolog` usenet group and has no immediate backup in this thesis. Where the body of this thesis presents isolated issues organised by chapter, this discussion section allows us to present an overall picture and introduce experience that is not described elsewhere in this thesis.

[1] `http://www.catb.org/Stildeesr/writings/cathedral-bazaar/cathedral-bazaar/`

This discussion starts with dimensions that apply to language and library design in a practical world where one has to accept history and historical mistakes, 'fashion' in language evolution as well as consider the 'learning curve' for programmers. These dimension expresses some of the experience we gained while developing our infrastructure. After introducing these dimensions we discuss the major decisions taken in the design of the SWI-Prolog-based infrastructure in section 11.3.1. Ideally, we should score our decisions on these dimensions, but that would require a level of detail that is unsuitable for this discussion.

1. *Compatibility*
   Although Prolog implementations are often incompatible when leaving the sublanguage defined by the ISO standard for Prolog, it is considered good practice to see whether there is an established solution for a problem and use this faithfully. If there is no acceptable existing API, the new API must be designed such that the naming is not confusing to programmers that know about an existing API and future standardisation is not made unnecessarily complex. An important implication is that one must try to avoid giving different semantics to (predicate) names used elsewhere.

2. *The learning curve*
   This is a difficult and fuzzy aspect of language and library design. Short learning curves can be obtained by reusing concepts from other popular systems as much as possible. Because Logic Programming provides concepts unknown outside the paradigm (e.g., logical variables and non-determinism), reusing concepts from other paradigms may result in inferior Prolog programs. For example, accessing RDF triples in an imperative language is generally achieved using a pattern language and a some primitive to iterating over matches. In Prolog we can use logical variables for pattern matching and non-determinism for iteration, providing a natural API where Prolog conjunctions represent graph patterns as defined in RDF query languages.

3. *Performance and scalability*
   Performance and scalability to large datasets is of utmost importance. We distinguish two aspects: the interface and the implementation. Interfaces must be designed to allow for optimised implementations. For example, interfaces that allow for lazy execution can be used to reduce startup time and if an interface allows for processing large batches of requests it may be possible to enhance the implementation by adding planning and concurrency.

4. *Standard compliance and code sharing*
   Especially where standard document formats (e.g., XML, RDF, JPEG) and protocols (e.g., HTTP, SSL) come into view there is a choice between implementing the interface from the specification of the standard or using an external library. The choice depends on stability of the standard, stability of API of an externally developed library and estimate on the implementation effort.

11.3.1 Key decisions about the infrastructure

This section summarises the original motivation for key decisions we took in designing the infrastructure and reconsiders some of these decisions based on our current knowledge and expectations for the future.

Graphics XPCE (chapter 5) resulted from the need to create graphical interactive applications for the KADS project in the mid-80s. For a long time, XPCE was our primary asset because it combines the power of Prolog with communication to the outside world in general and graphics in particular. The system was commercialised under the name ProWindows by Quintus. The design is, for a windowing environment, lightweight and fast; important properties with 4Mb main memory we had available when the core of XPCE was designed. Combined with a language with incremental (re-)compilation, it allows changing the code in a running application. Avoiding having to restart an interactive application and recreate the state where development happens greatly speeds up the implementation process of interactive applications.

Described in chapter 5 and evaluated in section 2 and section 9, we have realised an architecture that satisfies the aims expressed in research question 3. This architecture has proven to be a productive prototyping environment for the (few) programmers that master it. We attribute this primarily to the learning curve. There are two factors that make XPCE/Prolog hard to master.

- The size of a GUI library. For XPCE, this situation is even worse than for widely accepted GUI libraries because its terminology (naming of classes and methods) and organisation (functionality provided by the classes) is still based on forgotten GUI libraries such as SunView,[2] which harms the transfer of experience with other GUI libraries (cf. first dimension above). This is not a fundamental flaw of the design, but the result of a lack of resources.

- As already mentioned in section 5.9, the way XPCE classes can be created from Prolog is required to exploit functionality of the object system that must be achieved through subclassing. XPCE brings OO programming to Prolog, where the OO paradigm is optimised for deterministic side-effects required for (graphical) I/O. These are not the right choices if we wish to represent *knowledge* in a Prolog based OO paradigm, a task that is much better realised using an OO system designed for and implemented in Prolog such as Logtalk (Moura 2003). To put it differently, programming XPCE/Prolog is done in Prolog syntax, but it requires the programmer to *think* in another paradigm. Another example causing confusion are datatypes. Many Prolog data types have their counterpart in XPCE. For example, a Prolog list is similar to an XPCE *chain*. However, an XPCE *chain* is manipulated through destructive operations that have different syntax and semantics than Prolog list operations.

[2] `http://en.wikipedia.org/wiki/SunView`

There is no obvious way to remedie this situation. Replacing the graphics library underlying XPCE with a modern one fixes the first item at the cost of significant implementation work and loss of backward compatibility. This would not fix the second issue and it is not clear how this can be remedied. One option is to use a pure Prolog OO system and wrap that around a GUI library. However, Prolog is poorly equipped to deal with the low-level operations and side-effects required in GUI programming.

As stated in section 11.1 and realised in chapter 10, using a web-browser for GUI starts to become a viable alternative to native graphics. Currently, the state of widget libraries, support for interactive graphics and the development environment for JavaScript in a web-browser is still inferior to native graphics. As we have experienced during the development of ClioPatria, this situation is improving fast.

Using a web-browser for GUI programming will always involve programming in two languages and using a well defined interface in between. This is good for development in a production environment where the GUI and middleware are generally developed by different people anyway. It still harms productivity for single-developer prototyping though. Currently, we see no clear solution that provides a productive prototyping environment for Prolog-based interactive applications that fits well with Prolog and is capable of convincing the logic programming community.

XML/SGML support XML and SGML/HTML are often considered *the* standard document and data serialisation languages, especially in the web community. Supporting this family of languages is a first requirement for Prolog programming for the web.

The XML/SGML tree model maps naturally to a (ground) Prolog term, where we choose for `element`(*Tag, Attributes, Content*), the details of which are described in section 7.2. To exploit Prolog pattern matching, it is important to make the data canonical. In particular re-adding omitted tags to SGML/HTML documents and expanding entities simplifies processing the data. The representation is satisfactory for conversion purposes such as RDF/XML. Querying the model to select specific elements with generic Prolog predicates (e.g., `member/2`, `sub_term/2`) is possible, but sometimes cumbersome. Unification alone does not provide suitable pattern matching for XML terms. We have an experimental implementation of an XPath[3] inspired Prolog syntax that allows for non-deterministic querying of XML terms and which we have used for scraping information from HTML pages. The presented infrastructure is *not* a replacement for XML databases, but provides the basis for extracting information from XML documents in Prolog.

We opted for a more compact representation for *generating* HTML, mainly to enhance readability. A term `b`(*'Hello world'*) is much easier to read and write than `element`(*b, [], ['Hello world']*), but canonical terms of the from `element`(*Tag, Attributes, Content*) can be matched more easily in Prolog. The dual representation is unfortunate, but does not appear to cause significant confusion.

[3] `http://www.w3.org/TR/xpath`

RDF support The need for storing and querying RDF was formulated in the MIA project described in chapter 9. Mapping the RDF triple model to a predicate `rdf`(*Subject, Predicate, Object*) is obvious. Quickly growing scalability requirements changed the implementation from native Prolog to a dedicated C-library. The evaluation in section 3.6 shows that our RDF store is a state-of-the-art main memory store. Together with the neat fit between RDF and Prolog this has been an important enabling factor for building Triple20 and ClioPatria.

When we developed our RDF/XML parser, there were no satisfactory alternative. As we have shown in section 3.2, our technology allowed us to develop a parser from the specifications with little effort. Right now, our parser is significantly slower than the Raptor RDF parser. While the specification of RDF/XML and other RDF serialisations (e.g., Turtle) is subject to change, the API for an RDF parser is stable as it is based on a single task: read text into RDF triples.

A significant amount of effort is spent worldwide on implementing RDF stores with varying degrees of reasoning and reusing a third-party store allows us to concentrate on other aspects of the Semantic Web. Are there any candidates? We distinguish two types of stores: those with considerable reasoning capacity (e.g., OWLIM using the TTREE rule engine, Kiryakov et al. 2005) and those with limited reasoning capabilities (e.g., Redland, Beckett 2002). A store that provides few reasoning capabilities must allow for representing RDF resources as Prolog atoms because translating between text and atom at the interface level loses too much performance. Considering our evaluation of low-level stores, there is no obvious candidate to replace ours. A store that provides rule-based reasoning is not desirable because Prolog itself is a viable rule language. A store that only provides SPARQL-like graph queries is not desirable because Prolog performs graph pattern matching naturally itself and putting some layer in between only complicates usage (cf., relational database interfaces for Prolog, Jarke et al. 1984).

It is not clear where the balance is if support for more expressive languages such as OWL-DL is required. Writing a complete and efficient DL reasoner in Prolog involves an extensive amount of work. Reusing an external reasoner however, involves significant communication overhead. On 'batch' operations such as deducing the concept hierarchy, using an external reasoner is probably the best option. Answering specific questions (does class A subsume B, does individual I belong to class C?) the communication overhead will probably quickly become dominant and even a naive implementation in Prolog is likely to outperform an external reasoner.

If disk-based storage of RDF is required to satisfy scalability and startup-time requirements, it is much more likely that reusing an externally developed store becomes profitable. Disk-based stores are several orders of magnitude slower and because the relative costs of data conversion decreases, the need for a store that is closely integrates into Prolog to obtain maximal performance becomes less important.

Fortunately, the obvious and stable representation of the RDF model as `rdf`(*Subject, Predicate, Object*) facilitates switching between alternative implementations of the triple store without affecting application code.

HTTP support and embedding There are two ways to connect Prolog to the web. One option is to embed Prolog in an established HTTP server. For example, embed Prolog into the Java-based Tomcat HTTP server by means of the JPL[4] Prolog/Java interface. The second option is to write an HTTP server in Prolog as we have done.

Especially the Tomcat+JPL route is, judged from issues raised on the SWI-Prolog mailinglist, popular. We can understand this from the perspective of the learning curve. The question "we need a web-server serving pages with dynamic content?" quickly leads to Tomcat. Next issue, "we want to do things Prolog is good at", leads to embedding Prolog into Tomcat. Judging from the same mailinglist however, the combination has many problems. This is not surprising. Both systems come with a virtual machine that is designed to control the process, providing threads and garbage collection at different levels. Synchronising object lifetime between the two systems is complicated and the basic assumptions on control flow are so radically different that their combination in one process is cumbersome at best. XPCE suffers from similar problems, but because XPCE was designed to cooperate with Prolog (and Lisp) we have added hooks that facilitate cooperation in memory management.

Embedding Prolog into another language often looses the interactive and dynamic nature of Prolog, seriously reducing development productivity. It is generally recommended to connect well behaved and understood pieces of functionality to Prolog through its foreign interface or use the network to communicate. Network communication is particularly recommended for connecting to large programming environments such as Java or Python. Using separate processes, debugging and access to the respective development environments remains simple. The downside is of course that the communication bandwidth is much more limited, especially if latency is an issue.

Development, installation and deployment have been simplified considerably by providing a built-in web-server. Tools like PlDoc chapter 8 (Wielemaker and Anjewierden 2007) would suffer from too complicated installation requirements to be practical without native support for HTTP in Prolog.

Concurrency Adding multi-threading to Prolog is a requirement for the applications we want to build: all applications described in the thesis, except for the MIA tool, use multiple threads. Triple20 uses threads to update the *mediators* (see figure 2.2) in the background, thus avoiding blocking the GUI. PlDoc uses threads to serve documentation to the user's browser without interference with the development environment and ClioPatria uses threads to enhance its scalability as a web server. The API has proven to be adequate for applications that consist of a limited number of threads serving specific roles in the overall application.

The current threading support is less suitable for applications that wish to spread a single computation-intensive task over multiple cores because (1) thread creation is a relatively heavyweight task, (2) Prolog terms need to be *copied* between engines and (3) proper handling of exceptions inside a network of cooperating threads is complicated. Especially error handling can be remedied by putting more high-level work-distribution primitives into a library.

[4] `http://www.swi-prolog.org/packages/jpl/`

Logic Programming Finally, we come back to the use of Logic Programming for interactive knowledge-intensive applications. In the introduction we specialised Logic Programming to Prolog and we claim that the declarative reading of Prolog serves the representation of knowledge while the imperative reading serves the interactivity. Reflexiveness and incremental (re-)compilation add to the practical value of the language, notably for prototyping. This thesis shows that Prolog, after extending it and adding suitable libraries, is a competitive programming environment for knowledge-intensive (web) applications because

- Prolog is well suited for processing RDF. Thanks to its built-in resolution strategy and its ability to handle goals as data implementing something with the power of today's RDF query languages is relatively trivial. Defining more expressive reasoning on top of RDF using Prolog's backward reasoning is more complicated because the program will often not terminate. This can be resolved using forward reasoning at the cost of additional loading time, additional memory and slower updates. The termination problem can also be resolved using tabling (SLG resolution, Ramesh and Chen 1997) is a known technique, originally implemented in XSB and currently also supported by B-Prolog (Zhou et al. 2000), YAP (Rocha et al. 2001) and ALS-Prolog (Guo and Gupta 2001).

- Using Prolog for knowledge-intensive web-server tasks works surprisingly well, given proper libraries for handling the document formats and HTTP protocol. We established a stable infrastructure for generating (X)HTML pages in a modular way. The infrastructure allows for interleaving HTML represented as Prolog terms with Prolog generative grammar rules. Defining web-pages non-deterministically may seem odd at first sight but it allows one to eagerly start a page or page fragment and fallback to another page or fragment if some part cannot be generated without the need to check all preconditions rigorously before starting. This enhances modularity, as conditions can be embedded in the generating code.

11.4 Challenges

Declarative reasoning We have seen that Prolog itself is not suitable for declarative reasoning due to non-termination. Forward reasoning and tabling are possible solutions, each with their own problems. In the near future we must realise tabling, where we must consider the interaction with huge ground fact predicates implemented as a foreign extension (rdf/3). Future research should investigate appropriate design patterns to handle expressive reasoning with a proper mix of backward reasoning, forward reasoning, tabling and concurrency.

Scalability of the RDF store We claim that a main memory RDF store is a viable experimentation platform as it allows for many lookups to answer a single query where disk-based techniques require storing the result of (partial) forward reasoning to avoid slow repetitive lookups. Short retrieval times make it is much easier to experiment with in-core techniques

than with disk based techniques. From our experience in the E-culture project that resulted in ClioPatria, we can store enough vocabulary and collection data for meaningful experimenting with and evaluation of prototypes where the main challenge is *how* we must reason with our RDF data. See also section 10.5.0.4.

On the longer term, main memory techniques remain appropriate for relatively small databases as well as for local reasoning after fetching related triples from external stores. If we consider E-culture, a main memory store is capable of dealing with a large museum, but not with the entire collection of all Dutch museums, let alone all cultural heritage data available in Europe or the world. Using either large disk-based storage or a distributed network of main-memory-based servers are the options for a truly integrated Semantic Web on cultural heritage. Both pose challenges and the preferred approach depends highly on what will be identified as the adequate reasoning model for this domain, as well as organisational issues such as ownership of and control over the data.

Graphics The Prolog community needs a graphical interface toolkit that is appealing, powerful and portable. The GUI must provide a high level of abstraction and allow for reflexion to allow for generating interfaces from declarative specifications. XPCE is powerful, high-level, portable and reflexive. However, it cannot keep up with the developments in the GUI world. The OO paradigm that it brings to Prolog is poorly suited for knowledge representation, while the abundance of OO layers for Prolog indicate that there is a need for OO structuring in Logic Programming. This issue needs to be reconsidered. Given available GUI platforms, any solution is likely to involve the integration of Prolog with an external OO system and our work on XPCE should be considered as part of the solution.

Dissemination With the presented SWI-Prolog libraries, we created a great toolkit for our research. All described infrastructure is released as part of the Open Source SWI-Prolog distribution, while ClioPatria is available for download as a separate Open Source project. It is hard to judge how popular our RDF and web infrastructure is, but surely it is a minor player in the Semantic Web field. To improve this, we need success stories like ClioPatria, but preferably by third parties like DBtune[5] to attract more people from the web community. We also need porting the infrastructure to other Prolog systems to avoid fragmentation of the community and attract a larger part of the logic programmers. This has worked for some of the constraint libraries (e.g., CHR, clp(fd), clp(q)). Porting the infrastructure requires a portable foreign language interface. This requires a significant amount of work, but so did the introduction of extended unification that was required to implement constraint solvers. We hope that this thesis will motivate people in the (Semantic) Web and logic programming community to join their effort.

Logic programming is a powerful programming paradigm. We hope to have demon-

[5] http://dbtune.org/

strated in this thesis that Logic Programming can be successfully applied in the interactive distributed-systems world of the 21th century.

References

Adida, B. and M. Birbeck (2007, October). RDFa primer: Embedding structured data in web pages. `http://www.w3.org/TR/xhtml-rdfa-primer/`. W3C Working Draft 26.

Aleman-Meza, B., C. Halaschek, A. Sheth, I. Arpinar, and G. Sannapareddy (2004). SWETO: Large-scale semantic web test-bed. In *16th International Conference on Software Engineering and Knowledge Engineering (SEKE2004): Workshop on Ontology in Action*, Banff, Alberta, Canada.

Anjewierden, A. and L. Efimova (2006). Understanding weblog communities through digital traces: A framework, a tool and an example. In R. Meersman, Z. Tari, and P. Herrero (Eds.), *OTM Workshops (1)*, Volume 4277 of *Lecture Notes in Computer Science*, pp. 279–289. Springer.

Anjewierden, A., J. Wielemaker, and C. Toussaint (1990). Shelley — computer aided knowledge engineering. In B. Wielinga, B. G. John Boose, M. V. Someren, and G. Schreiber (Eds.), *Current Trends in Knowledge Acquisition*, Frontiers in Artificial Intelligence and Applications, pp. 41–59. Amsterdam: IOS Press. ISBN: 9051990367.

Anjewierden, A., B. Wielinga, and R. de Hoog (2004). Task and domain ontologies for knowledge mapping in operational processes. Metis Deliverable 4.2/2003, University of Amsterdam. tOKo home: `http://www.toko-sigmund.org/`.

Appleby, K., M. Carlsson, S. Haridi, and D. Sahlin (1988). Garbage collection for Prolog based on WAM. *Communications of the ACM 31*(6), 719–741.

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

Bagnara, R. and M. Carro (2002, May). Foreign language interfaces for Prolog: A terse survey. *ALP newsletter 15*.

Bast, H. and I. Weber (2007). The CompleteSearch Engine: Interactive, Efficient, and towards IR&DB Integration. In *CIDR 2007, Third Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, USA, pp. 88–95.

Bechhofer, S., R. Möller, and P. Crowther (2003). The dig description logic interface. In D. Calvanese, G. D. Giacomo, and E. Franconi (Eds.), *Description Logics*, Volume 81 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Beckett, D. (2002). The design and implementation of the redland rdf application framework. *Computer Networks 39*(5), 577–588.

Beckett, D. and B. McBride (2004, 10 February). Rdf/xml syntax specification (revised). Technical report, W3C Consortium. See: `http://www.w3.org/TR/2004/REC-rdf-syntax-grammar-20040210/`.

Berners-Lee, T. (1999). *Weaving the Web*. London: Orion Business.

Berners-Lee, T., J. Hendler, and O. Lassila (2001, May). The semantic Web. *Scientific American 284*(5), 34–43.

Brachman, R. J. and J. G. Schmolze (1985). An overview of the KL-ONE knowledge representation system. *Cognitive Science 9*(2), 171–216.

Brickley, D. and R. V. Guha (2000, 27 March). Resource description framework (RDF) schema specification 1.0. Candidate recommendation, W3C Consortium. See: `http://www.w3.org`.

Broekstra, J., A. Kampman, and F. van Harmelen (2002, July). Sesame: A generic architecture for storing and querying rdf and rdf schema. In I. Horrocks and J. Hendler (Eds.), *Proceedings ISWC'02*, Number 2342 in LNCS, pp. 54–68. Springer Verlag.

Butenhof, D. R. (1997). *Programming with POSIX threads*. Reading, MA, USA: Addison-Wesley.

Carriero, N. and D. Gelernter (1989, April). Linda in context. *Communications of the ACM 32*(4), 444–458.

Carro, M. and M. V. Hermenegildo (1999). Concurrency in Prolog using threads and a shared database. In *International Conference on Logic Programming*, pp. 320–334.

Chassell, R. J. and R. M. Stallman (1999). *Texinfo: The GNU Documentation Format*. Reiters.com.

Clark, K. G., L. Feigenbaum, and E. Torres (2007, June). Serializing sparql query results in json. W3C Working Group Note 18 June 2007.

Colmerauer, A. and P. Roussel (1996). *History of programming languages—II*, Chapter The birth of Prolog, pp. 331–367. New York, NY, USA: ACM.

de Boer, V., M. van Someren, and B. J. Wielinga (2007). A redundancy-based method for the extraction of relation instances from the web. *International Journal of Human-Computer Studies 65*(9), 816–831.

de Bosschere, K. and J.-M. Jacquet (1993). Multi-Prolog: Definition, operational semantics and implementation. In D. S. Warren (Ed.), *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, pp. 299–313. The MIT Press.

Dean, M., A. T. Schreiber, S. Bechofer, F. van Harmelen, J. Hendler, I. Horrocks, D. MacGuinness, P. Patel-Schneider, and L. A. Stein (2004, 10 February). OWL web ontology language reference. W3C recommendation, World Wide Web Consortium. Latest version: `http://www.w3.org/TR/owl-ref/`.

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

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

Ding, L., T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, , and J. Sachs (2004). Swoogle: A Search and Metadata Engine for the Semantic Web. In *Proceedings of the Thirteenth ACM Conference on Information and Knowledge Management*, Washington, D.C., USA, pp. 652–659.

Draxler, C. (1991). Accessing relational and NF^2 databases through database set predicates. In G. A. Wiggins, C. Mellish, and T. Duncan (Eds.), *ALPUK91: Proceedings of the 3rd UK Annual Conference on Logic Programming, Edinburgh 1991*, Workshops in Computing, pp. 156–173. Springer-Verlag.

Dublin Core Metadata Initiative (1999, July). *Dublin Core Metadata Element Set Version 1.1: Reference Description*.

Escalante, C. (1993). A simple model of prolog's performance: extensional predicates. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pp. 1119–1132. IBM Press.

Eskilson, J. and M. Carlsson (1998). SICStus MT—a multithreaded execution environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and K. Meinke (Eds.), *Programming Languages: Implementations, Logics, and Programs*, Volume 1490 of *Lecture Notes in Computer Science*, pp. 36–53. Springer-Verlag.

Fensel, D., I. Horrocks, F. van Harmelen, S. Decker, M. Erdmann, and M. Klein (2000). OIL in a nutshell. In *Knowledge Engineering and Knowledge Management: 12th International Conference EKAW2000, Juan-les-Pins*, Volume 1937 of *Lecture Notes in Artificial Intelligence*, Berlin/Heidelberg, pp. 1–16. Springer-Verlag.

Freire, J., D. S. Warren, K. Sagonas, P. Rao, and T. Swift (1997, jan). XSB: A system for efficiently computing well-founded semantics. In *Proceedings of LPNMR 97*, Berlin, Germany, pp. 430–440. Springer Verlag. LNCS 1265.

Fridman Noy, N., R. W. Fergerson, and M. A. Musen (2000). The knowledge model of Protégé-2000: combining interoperability and flexibility. In *Knowledge Engineering and Knowledge Management: 12th International Conference EKAW2000, Juan-les-Pins*, Volume 1937 of *Lecture Notes in Artificial Intelligence*, Berlin/Heidelberg, pp. 17–32. Springer-Verlag. Also as: Technical Report Stanford University, School of Medicine, SMI-2000-0830.

Frühwirth, T. (1998, October). Theory and Practice of Constraint Handling Rules. In P. Stuckey and K. Marriot (Eds.), *Special Issue on Constraint Logic Programming* (1–3 ed.), Volume 37.

Getty (2000). ULAN: Union List of Artist Names. `http://www.getty.edu/research/tools/vocabulary/ulan/`.

Gidenstam, A. and M. Papatriantafilou (2007). Lfthreads: A lock-free thread library. In E. Tovar, P. Tsigas, and H. Fouchal (Eds.), *OPODIS*, Volume 4878 of *Lecture Notes in Computer Science*, pp. 217–231. Springer.

Gil, Y., E. Motta, V. R. Benjamins, and M. A. Musen (Eds.) (2005). *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*, Volume 3729 of *Lecture Notes in Computer Science*. Springer.

Googley, M. M. and B. W. WAH (1989). Efficient reordering of PROLOG programs. *IEEE Transactions on Knowledge and Data Engineering*, 470–482.

Graham, S. L., P. B. Kessler, and M. K. McKusick (1982). gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pp. 120–126.

Gras, D. C. and M. V. Hermenegildo (2001). Distributed WWW programming using (Ciao-)Prolog and the PiLLoW library. *TPLP 1*(3), 251–282.

Grosso, W. E., H. Eriksson, R. W. Fergerson, J. H. Gennari, S. W. Tu, and M. A. Musen (1999). Knowledge modeling at the millennium: The design and evolution of Protégé-2000. In *12th Banff Workshop on Knowledge Acquisition, Modeling, and Management. Banff, Alberta*. URL: http://smi.stanford.edu/projects/protege (access date: 18 December 2000).

Guo, H.-F. and G. Gupta (2001). A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of the 17th International Conference on Logic Programming*, London, UK, pp. 181–196. Springer-Verlag.

Guo, Y., Z. Pan, and J. Heflin (2004). An evaluation of knowledge base systems for large owl datasets. See McIlraith, Plexousakis, and van Harmelen (2004), pp. 274–288.

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

Haase, P., J. Broekstra, A. Eberhart, and R. Volz (2004). A comparison of rdf query languages. See McIlraith, Plexousakis, and van Harmelen (2004), pp. 502–517.

Hassan Aït-Kaci (1991). *Warrens Abstract Machine*. MIT Press. Out of print; available from `http://www.isg.sfu.ca/~hak/documents/wam.html`.

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

Hermenegildo, M. V. (2000). A documentation generator for (C)LP systems. In J. W. Lloyd, V. Dahl, U. Furbach, M. Kerber, K.-K. Lau, C. Palamidessi, L. M. Pereira, Y. Sagiv, and P. J. Stuckey (Eds.), *Computational Logic*, Volume 1861 of *Lecture Notes in Computer Science*, pp. 1345–1361. Springer.

Holzbaur, C. (1990). Realization of forward checking in logic programming through extended unification. Report TR-90-11, Oesterreichisches Forschungsinstitut fuer Artificial Intelligence, Wien, Austria.

Horrocks, I. (1999). Fact and ifact. In P. Lambrix, A. Borgida, M. Lenzerini, R. Möller, and P. F. Patel-Schneider (Eds.), *Description Logics*, Volume 22 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Horrocks, I., P. F. Patel-Schneider, and F. van Harmelen (2003). From SHIQ and RDF to OWL: The Making of a Web Ontology Language . *Journal of Web Semantics 1*(1).

Huang, Z. and H. Stuckenschmidt (2005). Reasoning with multi-version ontologies: A temporal logic approach. See Gil, Motta, Benjamins, and Musen (2005), pp. 398–412.

Huang, Z., F. van Harmelen, and A. ten Teije (2005, august). Reasoning with inconsistent ontologies. In L. P. Kaelbling and A. Saffiotti (Eds.), *IJCAI*, Edinburgh, Scotland, pp. 454–459. Professional Book Center.

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

Huynh, D., D. Karger, and R. Miller (2007). Exhibit: Lightweight structured data publishing. In *16th International World Wide Web Conference*, Banff, Alberta, Canada. ACM.

Hyvönen, E., M. Junnila, S. Kettula, E. Mäkelä, S. Saarela, M. Salminen, A. Syreeni, A. Valo, and K. Viljanen (2005, October). MuseumFinland — Finnish museums on the semantic web. *Journal of Web Semantics 3*(2-3), 224–241.

Janik, M. and K. Kochut (2005). Brahms: A workbench rdf store and high performance memory system for semantic association discovery. See Gil, Motta, Benjamins, and Musen (2005), pp. 431–445.

Jarke, M., J. Clifford, and Y. Vassiliou (1984). An optimizing prolog front-end to a relational query system. In *SIGMOD '84: Proceedings of the 1984 ACM SIGMOD*

international conference on Management of data, New York, NY, USA, pp. 296–306. ACM.

Jeffery, D., F. Henderson, and Z. Somogyi (2000). Type classes in mercury. In *ACSC*, pp. 128–135. IEEE Computer Society.

Kim, T. (1993). *XWIP Reference Manual, Version 0.6*. UCLA Computer Science Department. Technical Report CSD-880079.

King, R. and A. Srinivasan (1996). Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives 104*(5), 1031–1040.

Kiryakov, A., D. Ognyanov, and D. Manov (2005). Owlim - a pragmatic semantic repository for owl. In M. Dean, Y. Guo, W. Jun, R. Kaschek, S. Krishnaswamy, Z. Pan, and Q. Z. Sheng (Eds.), *WISE Workshops*, Volume 3807 of *Lecture Notes in Computer Science*, pp. 182–192. Springer.

Knuth, D. E. (1984). Literate programming. *Comput. J. 27*(2), 97–111.

Krasner, G. E. and S. T. Pope (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program. 1*(3), 26–49.

Lafon, Y. and B. Bos (2000, 28 September). Describing and retrieving photographs using RDF and HTTP. Note, W3C Consortium. URL: http://www.w3.org/TR/2000/NOTE-photo-rdf-20000928.

Lassila, O. and R. R. Swick (1999, 22 February). Resource description framework (RDF) model and specification. Recommendation, W3C Consortium. URL: http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

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

Leuf, B. and W. Cunningham (2001). *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley.

Mäkelä, E., E. Hyvönen, S. Saarela, and K. Viljanen (2004). Ontoviews - a tool for creating semantic web portals. See McIlraith, Plexousakis, and van Harmelen (2004), pp. 797–811.

McBride, B. (2001). Jena: Implementing the rdf model and syntax specification. In *SemWeb*.

McCarthy, J. (1969). The advice taker. In M. L. Minsky (Ed.), *Semantic Information Processing*. The MIT Press.

McGuinness, D. L., R. Fikes, J. Rice, and S. Wilder (2000, April 12-15). An environment for merging and testing large ontologies. In *Proc. Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado*.

McIlraith, S. A., D. Plexousakis, and F. van Harmelen (Eds.) (2004). *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima, Japan, November 7-11, 2004. Proceedings*, Volume 3298 of *Lecture Notes in Computer Science*. Springer.

Merritt, D. (1995, May/June). Objects and logic. *PC AI magazine 9*(3), 16–22.

Miklos, Z., G. Neumann, U. Zdun, and M. Sintek (2005). Querying semantic web resources using triple views. In Y. Kalfoglou, M. Schorlemmer, A. Sheth, S. Staab, and M. Uschold (Eds.), *Semantic Interoperability and Integration*, Number 04391 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany. `http://drops.dagstuhl.de/opus/volltexte/2005/47`.

Miles, A. and S. Becchofer (2008, 25 January). SKOS simple knowledge organization system reference. W3C working draft, World-Wide Web Consortium. Latest version: http://www.w3.org/TR/skos-reference.

Miles, A. J. (2001). Owl ontology for thesaurus data. Deliverable, SWAD-Europe.

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

Moura, P. (2003, September). *Logtalk - Design of an Object-Oriented Logic Programming Language*. Ph. D. thesis, Department of Informatics, University of Beira Interior, Portugal.

Moura, P. (2008). Resources in object-oriented logic programming. `http://www.ci.uc.pt/oolpr/oolpr.html`.

Moura, P., J. Wielemaker, M. Carro, P. Nunes, P. Robinson, R. Marques, T. Swift, U. Neumerkel, and V. S. Costa (2008). Prolog multi-threading support. ISO/IEC DTR 132115:2007.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming 13*(3-4), 245–286.

Muggleton, S. and L. D. Raedt (1994). Inductive Logic Programming: Theory and Method. *Journal of Logic Programming 19-20*, 629–679.

Musen, M. A., R. W. Fergerson, W. E. Grosso, N. F. Noy, M. Crubézy, and J. H. Gennari (2000). Componentbased support for building knowledge-acquisition systems. In *Conference on Intelligent Information Processing (IIP 2000)*, Beijing, China. http://smi-web.stanford.edu/pubs/SMI_Abstracts/SMI-2000-0838.html.

Noy, N. F., M. Sintek, S. Decker, M. Crubezy, R. W. Fergerson, and M. A. Musen (2001). Creating Semantic Web contents with protege-2000. *IEEE Intelligent Systems 16*(2), 60–71.

Parsia, B. (2001). RDF applications with Prolog. O'Reilly XML.com, `http://www.xml.com/pub/a/2001/07/25/prologrdf.html`.

Patel, K. and G. Gupta (2003). Semantic processing of the semantic web. In D. Fensel, K. P. Sycara, and J. Mylopoulos (Eds.), *International Semantic Web Conference*, Volume 2870 of *Lecture Notes in Computer Science*, pp. 80–95. Springer.

Paulson, L. D. (2005). Building Rich Web Applications with Ajax. *IEEE Computer 38*(10), 14–17.

Peterson, T. (1994). *Introduction to the Art and Architecture Thesaurus*. Oxford University Press. See also: `http://www.getty.edu/research/tools/vocabulary/aat/`.

Philips, L. (2000, june). The double metaphone search algorithm. *j-CCCUJ 18*(6), 38–43.

Pieterse, V., D. G. Kourie, and A. Boake (2004). A case for contemporary literate programming. In *SAICSIT '04: Proceedings of the 2004 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, Republic of South Africa, pp. 2–9. South African Institute for Computer Scientists and Information Technologists.

Prud'hommeaux, E. and A. Seaborne (2008, January). Sparql query language for rdf. W3C Recommendation 15 January 2008.

Ramakrishnan, I. V., P. Rao, K. Sagonas, T. Swift, and D. S. Warren (1995, June 13–18). Efficient tabling mechanisms for logic programs. In L. Sterling (Ed.), *Proceedings of the 12th International Conference on Logic Programming*, Cambridge, pp. 697–714. MIT Press.

Ramesh, R. and W. Chen (1997). Implementation of tabled evaluation with delaying in prolog. *IEEE Trans. Knowl. Data Eng. 9*(4), 559–574.

Ramsey, N. and C. Marceau (1991). Literate programming on a team project. *Software - Practice and Experience 21*(7), 677–683.

RDFCore WG (2003, February). RDF/XML Syntax Specification (Revised)a. W3C Working Draft, World Wide Web Consortium. `http://www.w3.org/TR/rdf-syntax-grammar/`.

Rocha, C., D. Schwabe, and M. de Aragao (2004). A hybrid approach for searching in the semantic web. In *Proceedings of the 13th International World Wide Web Conference*, New York, NY, USA, pp. 374–383.

Rocha, R., F. M. A. Silva, and V. S. Costa (2001). On a tabling engine that can exploit or-parallelism. In *Proceedings of the 17th International Conference on Logic Programming*, London, UK, pp. 43–58. Springer-Verlag.

Schlobach, S. and Z. Huang (2005). Inconsistent ontology diagnosis: Framework and prototype. Deliverable D3.6.1, SEKT. URL = http://www.cs.vu.nl/˜huang/sekt/sekt361.pdf.

Schmidt, D. C. and I. Pyarali (2008, July). Strategies for implementing posix condition variables on win32. `http://www.cs.wustl.edu/~schmidt/win32-cv-1.html`.

Schreiber, G., B. Dubbeldam, J. Wielemaker, and B. Wielinga (2001, may/june). Ontology-based photo annotation. *IEEE Intelligent Systems 16*, 66–74.

Schrijvers, T. and B. Demoen (2004). The K.U. Leuven CHR system: implementation and application. In T. Frühwirth and M. Meister (Eds.), *First Workshop on Constraint Handling Rules: Selected Contributions*, pp. 430–440. ISSN 0939-5091.

Shen, K., J. Schimpf, S. Novello, and J. Singer (2002). A high-level generic interface to external programming language for ECLiPSe. In *Practical Aspects of Declarative Languages*, Berlin, Germany. Springer Verlag. LNCS 2257.

Shum, A. and C. Cook (1993). AOPS: an abstraction-oriented programming system for literateprogramming. *Software Engineering Journal 8*(3), 113–120.

Shum, S. and C. Cook (1994). Using literate programming to teach good programming practices. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, New York, NY, USA, pp. 66–70. ACM Press.

SICS (1997). *Quintus Prolog, User Guide and Reference Manual*. SICS.

SICS (1998). *Quintus ProXT Manual*. SICS. http://www.sics.se/isl/quintus/proxt/frame.html.

Sintek, M. and S. Decker (2002). *TRIPLE* — A query, inference, and transformation language for the Semantic Web. *Lecture Notes in Computer Science 2342*, 364–.

Srinivasan, A. (2003). *The Aleph Manual*.

Stallman, R. M. (1981). Emacs the extensible, customizable self-documenting display editor. *SIGPLAN Not. 16*(6), 147–156.

Struyf, J. and H. Blockeel (2003). Query optimization in inductive logic programming by reordering literals. In T. Horváth and A. Yamamoto (Eds.), *Proceedings of the 13th International Conference on Inductive Logic Programming*, Volume 2835 of *Lecture Notes in Artificial Intelligence*, pp. 329–346. Springer-Verlag.

Szeredi, P., K. Molnár, and R. Scott (1996, September). Serving Multiple HTML Clients from a Prolog Application. In *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn. Available from http://clement.info.umoncton.ca/˜lpnet/lpnet9.html.

Tam, A. M. and C. H. C. Leung (2001). Structured natural-language descriptions for semantic content retrieval of visual materials. *JASIST 52*(11), 930–937.

Tordai, A., B. Omelayenko, and G. Schreiber (2007). Semantic excavation of the city of books. In *Proc. Semantic Authoring, Annotation and Knowledge Markup Workshop (SAAKM2007)*, Volume 314, pp. 39–46. CEUR-WS. http://ceur-ws.org/Vol-314.

Tummarello, G., C. Morbidoni, and M. Nucci (2006). Enabling Semantic Web communities with DBin: an overview. In *Proceedings of the Fifth International Semantic Web Conference ISWC 2006*, Athens, GA, USA.

Tummarello, G., E. Oren, and R. Delbru (2007, November). Sindice.com: Weaving the open linked data. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, Berlin, Heidelberg, pp. 547–560.

van den Berg, J. (1995). Subject retrieval in pictorial information systems. In *Electronic Filing, Registration, and Communication of Visual Historical Data. Abstracts for Round Table no 34 of the 18th International Congress of Historical Sciences. Copenhagen*, pp. 21–28. See also http://www.iconclass.nl.

van der Waal, H. (1985). IconClass: An iconographic classification system. Technical report, Royal Dutch Academy of Sciences (KNAW).

van Gendt, M., A. Isaac, L. van der Meij, and S. Schlobach (2006). Semantic web techniques for multiple views on heterogeneous collections: A case study. In J. Gonzalo, C. Thanos, M. F. Verdejo, and R. C. Carrasco (Eds.), *ECDL*, Volume 4172 of *Lecture Notes in Computer Science*, pp. 426–437. Springer.

van Heesch, D. (2007). *Doxygen, a documentation system for C++*. http://www.stack.nl/~dimitri/doxygen/.

Visual Resources Association Standards Committee (2000, July). VRA Core Categories, Version 3.0. Technical report, Visual Resources Association. URL: http://www.vraweb.org/vracore3.htm.

Železný, F., A. Srinivasan, and D. Page (2003). Lattice-search runtime distributions may be heavy-tailed. In S. Matwin and C. Sammut (Eds.), *Proceedings of the 12th International Conference on Inductive Logic Programming*, Volume 2583 of *Lecture Notes in Artificial Intelligence*, pp. 333–345. Springer-Verlag.

W3C Web Ontology Working Group (2001). http://www.w3.org/2001/sw/WebOnt/.

Wiederhold, G. (1992, March). Mediators in the architecture of future information systems. *IEEE Computer 25*(3), 38–49.

Wielemaker, J. (1997-2008). SWI-Prolog 5.6: Reference manual. http://www.swi-prolog.org/documentation.html.

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

Wielemaker, J. (2003b). An overview of the swi-prolog programming environment. In F. Mesnard and A. Serebrenik (Eds.), *WLPE*, Volume CW371 of *Report*, pp. 1–16. Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Heverlee (Belgium).

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

Wielemaker, J. and A. Anjewierden (1989, November). Separating User Interface and Functionality Using a Frame Based Data Model. In *Proceedings Second Annual Symposium on User Interface Software and Technology*, Williamsburg, Virginia, pp. 25–33. ACM Press.

Wielemaker, J. and A. Anjewierden (1992). *Programming in XPCE/Prolog*. Roetersstraat 15, 1018 WB Amsterdam, The Netherlands: SWI, University of Amsterdam. E-mail: jan@swi.psy.uva.nl.

Wielemaker, J. and A. Anjewierden (2002). An architecture for making object-oriented systems available from prolog. In *WLPE*, pp. 97–110.

Wielemaker, J. and A. Anjewierden (2007). PlDoc: Wiki style literate programming for Prolog. In P. Hill and W. Vanhoof (Eds.), *Proceedings of the 17th Workshop on Logic-Based methods in Programming Environments*, pp. 16–30.

Wielemaker, J., M. Hildebrand, and J. van Ossenbruggen (2007). Using Prolog as the fundament for applications on the semantic web. In S. Heymans et al (Ed.), *Proceedings of ALPSWS2007*, pp. 84–98.

Wielemaker, J., M. Hildebrand, J. van Ossenbruggen, and G. Schreiber (2008). Thesaurus-based search in large heterogeneous collections. In A. P. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. W. Finin, and K. Thirunarayan (Eds.), *International Semantic Web Conference*, Volume 5318 of *Lecture Notes in Computer Science*, pp. 695–708. Springer.

Wielemaker, J., Z. Huang, and L. van der Meij (2008). SWI-Prolog and the web. *Theory and Practice of Logic Programming 8*(3), 363–392.

Wielemaker, J., A. T. Schreiber, and B. J. Wielinga (2003a). Supporting semantic image annotation and search. In S. Handschuh and S. Staab (Eds.), *Annotation for the Semantic Web*, Volume 96 of *Frontiers in Artificial Intelligence and Applications*, pp. 147–155. Amsterdam: IOS Press. ISBN: 1 58603 345 X.

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

Wielemaker, J., G. Schreiber, and B. Wielinga (2005, November). Using triples for implementation: the Triple20 ontology-manipulation tool. In *ISWC 2005*, Berlin, Germany, pp. 773–785. Springer Verlag. LNCS 3729.

Wielinga, B. J., A. T. Schreiber, and J. A. Breuker (1992). Kads: a modelling approach to knowledge engineering. *Knowl. Acquis. 4*(1), 5–53.

Wielinga, B. J., A. T. Schreiber, J. Wielemaker, and J. A. C. Sandberg (2001, 21-23 October). From thesaurus to ontology. In Y. Gil, M. Musen, and J. Shavlik (Eds.), *Proceedings 1st International Conference on Knowledge Capture, Victoria, Canada*, New York, pp. 194–201. ACM Press.

Zhou, N.-F., Y.-D. Shen, L.-Y. Yuan, and J.-H. You (2000). Implementation of a linear tabling mechanism. In *PADL '00: Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, London, UK, pp. 109–123. Springer-Verlag.

Summary

Logic programming is the use of mathematical logic for computer programming. In its pure form, a logic program is a declarative description of a solution that is completely separated from the *task* of solving the problem, the procedural aspect. Lacking an efficient resolution strategy for expressive logics and the difficulty of expressing problems that are by nature procedural limit the usability of pure logic programs for building real-world applications.

An important step towards a usable language for logic programming was the invention of the programming language 'Prolog'. This language is based on a simplified form of first order logic (Horn Clauses) and a simple resolution strategy (SLD resolution). Prolog programs can, at the same time, be read as a declarative description of a solution and as a procedural strategy to find this solution. In other words, Prolog can describe both knowledge and procedures. Interaction is a typical aspect of applications that have procedural aspects. The combination of declarative and procedural aspects in one language (Prolog) is what makes this language promising for the applications that are considered in this thesis: knowledge-intensive interactive applications.

There are dozens of implementations of the Prolog language. Most concentrate on Prolog as a rule-based system and leave the interaction task to other programming environments. This implies that an interactive application can only be built using a *hybrid* environment. The required bridge between Prolog and the language used for the interactive aspects of the application makes programming, and in particular prototyping of interactive applications, complicated and can cause a significant performance degradation. In this thesis, we study the requirements, design and implementation of an integrated Prolog-based environment that can be used for developing knowledge-intensive interactive applications.

Building applications is important in this study. The primary goal of these applications was *not* their architecture but, for example, tools to support knowledge-engineers with modelling knowledge or explore how differently structured databases that contain collections from different museums can be joined and searched together. The architecture of these applications was irrelevant to projects in which they were developed, but the search for adequate building blocks for these applications forms the basis of this thesis. While developing these applications we identified reusable components and added these to Prolog and its libraries. In

subsequent implementations we tested these extensions and refined or, if necessary, replaced them. Stable extensions have been described in scientific publications and are distributed as Open Source. Feedback from the Open Source community provided suggestions for alternatives or further refinement. In this study we particularly pay attention to the following three problems:

- *Knowledge representation*
  Although Prolog programs can be read as a declarative specification, the language is limited. Many Prolog programs that are a correct declarative description of a problem do not terminate or do not terminate within a reasonable time. In addition to the termination and performance problem, logic-based languages are considered inferior to frame-based languages (e.g., KL-ONE) for knowledge-representation that supports interactive inspection and manipulation.

  At the start of this century, the Semantic Web community proposed the RDF language for knowledge-representation. This language provides a very simple datamodel: triples of the format {subject, predicate, value}. More expressive languages such as RDFS and OWL are layered on top of this datamodel. The simple relational model of RDF fits perfectly with Prolog and now forms the basis of our knowledge-representation.

- *Graphical applications*
  Most Prolog implementations do not provide graphics and use a bridge to a conventional graphical environment. In our opinion this harms productivity too much, while the declarative aspects of Prolog are useful for the description of interfaces and user-interaction. Most graphical environments are structured as an object-oriented system. We realised a generic bridge between Prolog and an external object-oriented system and described the conditions to use this architecture with other object-oriented systems.

- *Web applications*
  In the recent past the web developed from a web of documents to a web of interactive applications. The use of knowledge in such applications is an active research topic. The integration of museum data mentioned earlier is an example. Because RDF is particularly suitable for integrating diverse knowledge-bases and RDF fits well with Prolog, the use of Prolog for the knowledge-aspects of this research is promising. If we embed Prolog in a traditional web-server and page-generation language (e.g., Apache with PHP), we loose many of Prolog's advantages for fast interactive development, i.e., we need a more 'Prolog-friendly' way to generate web-applications.

Our challenge is to extend Prolog and add libraries that make the language suitable for building large and innovative applications entirely in Prolog. In this architecture, only well understood primitives such as interfaces to the operating system, network and graphics are written in other languages and made available as a Prolog library.

Overview of this thesis Part I of this thesis describes extensions to Prolog and Prolog libraries that allow for writing knowledge-intensive interactive applications entirely in Prolog. Were possible, we compare our components with approaches used for similar problems in other (often Prolog) environments. Part II is a formative evaluation of these extensions: the described case-studies show that the programming environment can support the application, and at the same time both the development and evaluation of these applications have contributed to the development or refinement of the extensions described in part I.

Part I Chapter 2 describes Triple20, a tool for browsing and limited editing of ontologies. Triple20 is not part of Prolog and its libraries, which suggests it should not be in part I of this thesis. Although Triple20 can be used as a Prolog library for RDF-based graphical applications, the main reason to start this thesis with Triple20 is that this chapter describes how knowledge is represented in the RDF model and how we can use the RDF triple model as the foundation for an architecture to build interactive applications. This chapter describes an extension of the *model-view-controller* reference model for interactive applications which is necessary to bridge the large gap between the low-level knowledge-representation and the interface efficiently. Concurrency (threads) is one of the necessary building blocks. The design and implementation of Triple20 is based on requirements formulated by evaluating the 'MIA' tools, described in chapter 9.

The remainder of part I describes extensions to Prolog and Prolog libraries that made, for example, Triple20 possible: storage and query of RDF triples, creating execution plans for complex queries on this data, connection to graphical libraries, concurrency and an infrastructure for building web-applications.

The RDF triple model fits perfectly on the relational model of Prolog. A triple-database that is suitable for researching the aforementioned integration of museum collections contains at least 10 million triples, must be accessible from multiple threads and allow for efficient full-text search. RDF predicates are organised in one or more hierarchies, which must be used for reasoning over integrated knowledge-bases. This type of *entailment reasoning* must be supported with minimal overhead. The order in which individual RDF patterns are matched to the knowledge-base is of utmost importance for executing compound search requests (conjunctions). This optimisation problem, which can be compared to database 'join' optimisation, is the subject of chapter 4. It is based on metrics from the RDF database. Part of the requirements on the RDF store are too specific to base the implementation on an optimised version of the Prolog dynamic database. The implementation of the RDF store is the subject of chapter 3.

Chapter 5 discusses the bridge between Prolog and graphical libraries. Almost all such libraries are based on the object-oriented paradigm. First, we describe a compact interface to manipulate objects from Prolog. New functionality in object-oriented systems is typically realised by deriving new classes from the base-classes (sub-classing). This cannot be achieved using the above mentioned simple interface, which would imply we still need to program in both Prolog and the object-oriented language to realise an application. We can solve this for most object-oriented systems by describing the class in Prolog and use this description

to generate classes and *wrappers* that call the implementation in Prolog. The chapter continues with describing further integration of Prolog datatypes and non-determinism in the object-oriented environment and concludes with a critical evaluation of this object system for Prolog.

Chapter 6 discusses concurrency in Prolog. Traditionally, most research on concurrency in Prolog aims at executing a single program using multiple CPUs. The declarative nature of the language make people believe that Prolog is more suitable to automatic introduction of concurrency than procedural languages. Still, this research did not convince many software developers. We had more simple demands: background computations in graphical environments, scalability of web-servers and exploiting recent *multi-core* hardware for these tasks. On this basis we created a simple model of cooperating Prolog engines, which has been adopted by two other Prolog implementations and is the basis for standardisation within ISO WG17. Chapter 6 describes the model, consequences for the Prolog implementation and two performance evaluations.

Chapter 7 provides an overview of the developments that support web-applications. This chapter gives a summary of the previous chapters about RDF and positions this material in the wider context of web-applications. Parsing, representing and generating web-documents is an important topic in this chapter. Where possible, we compare our approach to PiLLoW, another infrastructure for web-programming in the Prolog community. We discuss a Prolog-based web-server (HTTP), stressing code organisation and scalability aspects. Chapter 7 enumerates necessary features of a Prolog implementation for this type of applications: concurrency, unlimited atoms, atom garbage collection and support for the international UNICODE character set.

Part II The second part discusses three applications that contributed to the development and evaluation of the infrastructure described in the first part.

The first application (PlDoc, chapter 8) is also part of the SWI-Prolog infrastructure. PlDoc is an environment for *literate programming* in Prolog. Its embedded web-server provides an interactive environment to consult and improve the documentation. PlDoc is part of the development environment that is necessary to create a productive programming environment. Building the development tools has contributed significantly to testing and refinement of the infrastructure, notably for graphical applications.

Chapter 9 describes two experiments using a prototype tool for annotation and search of multimedia collections supported by background knowledge. This chapter also describes the software architecture, which is based on XPCE (chapter 5). Apart from XPCE, the infrastructure described in part I did not yet exist. This was our first tool which used RDF for exchanging knowledge and where the RDF model was central to the design of the tool. This chapter was included for three reasons: description of the application context, stress that tool-development and —in this case— research to the role of knowledge in annotations are closely integrated and formulate requirements for the next generation of infrastructure and tools. This work lead to the development of Triple20 and a scalable RDF store. It also provided an additional motivation to introduce concurrency in Prolog.

Part II of this thesis concludes with chapter 10 which describes the semantic search and annotation toolkit called ClioPatria. This chapter motivates the architecture of ClioPatria, which uses all technology described in part I.[6] The *use-case* for ClioPatria is the development of a prototype application that makes descriptions of works of art from museums, together with available background information such as databases on artists, art vocabulary and geographical information, searchable. One of the challenges is the diversity in data formats, schemas and terminology used by the museums. The structure of the integrated knowledge is so complex that structured queries (e.g., SQL queries) cannot be realistically formulated. We made this data searchable by exploring the RDF graph based on *semantic distance*, after which we cluster the results based on the semantic relation between search-term and the works of art found. ClioPatria provides a modern web-based interactive interface based on AJAX technology and Yahoo! (YUI). The server is a Prolog web-server that provides all services.

Discussion The assumption in this thesis is that logic programming, and particularly Prolog, is suitable for the development of knowledge-intensive interactive software. The introduction of RDF and RDF-based knowledge-representation languages has contributed to the credibility of Prolog in this domain. Some properties of RDF models, such as their scale and required inferences, make a pure Prolog implementation less suitable. However, RDF models can be implemented as a library in another language (e.g., C), which acts as a natural extension to Prolog. The web has a large influence on the architecture of applications. Prolog is, extended with suitable libraries, an adequate language for web-applications.

Next to further standardisation of the Prolog language, it is of vital importance that the Prolog community establishes standards for representing and generation of (web-)documents in XML and HTML. Such standards are needed to share resources for web-applications within the Prolog community. The RDF model can play this role for knowledge-representation because accessing the RDF model from Prolog leaves few options.

This thesis is a direct consequence of how the SWI department (now called HCS) has integrated software development with research, providing room for the software developers to establish their own research agenda: software architecture. This approach has clear advantages: (1) it provides an opportunity for scientifically motivated software developers to carry out research into software architecture in the context of a concrete requirement for an application and (2) development of the target application is based on direct involvement. Direct involvement supports a short development and evaluation cycle, which is a fruitful process model for a large class of applications. This certainly applies to academic demonstrators.

With SWI-Prolog we realised a productive environment for our own research. Wide acceptance of the Open Source philosophy has contributed to the popularity of SWI-Prolog, through which we helped spreading the logic programming paradigm in academic and commercial environments. At the same time, this user community motivates us and feeds us with problems and solutions.

[6] The graphical interface is only used as part of the Prolog development environment.

Samenvatting

Logisch programmeren gaat over het gebruik van mathematische logica voor het programmeren van een computer. In haar pure zin betekent dit volledig ontkoppelen van de declaratieve beschrijving van de oplossing en de oplostaak, het procedurele aspect. Bij gebrek aan efficiënte oplosstrategieën voor expressieve logicas en de moeilijkheid om inherent procedurele aspecten te beschrijven heeft deze pure vorm slechts een beperkt toepassingsgebied.

Een belangrijke stap naar een practisch bruikbare taal voor logisch programeren was de uitvinding van de programmeertaal Prolog. Deze taal is gebaseerd op een vereenvoudigde logica (Horn clauses) en een eenvoudige bewijsstrategie (SLD resolution). Prolog programmas kunnen gelezen worden als een declaratieve beschrijving van de oplossing en tegelijkertijd als een procedurele strategie om deze oplossing af te leiden. Met andere woorden, Prolog is in staat zowel kennis als procedures te beschrijven. Met name de beschrijving van interactie heeft procedurele aspecten. Deze combinatie in één programmeertaal (Prolog) maakt deze taal veelbelovend voor het type applicaties dat in dit proefschrift behandeld wordt: kennisintensieve interactieve applicaties.

Er bestaan enkele tientallen implementaties van de taal Prolog. Vrijwel allemaal concentreren deze zich op Prolog als regelgebaseerd systeem en laten zij interactie over aan andere programmeeromgevingen. Dit impliceert dat voor het bouwen van interactieve applicaties een *hybride* omgeving gebruikt moet worden. De noodzakelijke brug tussen Prolog en de taal die voor de interactieve aspecten gebruikt wordt maakt programmeren, en met name prototyping van interactieve applicaties, ingewikkeld en kan daarnaast zorgen voor een significant verlies aan performance. Dit proefschrift gaat over de vraag hoe een op Prolog gebaseerde infrastructuur er moet uitzien om kennisintensieve interactieve applicaties in een zoveel mogelijk geïntegreerde omgeving te kunnen schrijven.

Om deze vraag te beantwoorden zijn een aantal applicaties gebouwd. Het primaire doel van deze applicaties was *niet* hun architectuur, maar bijvoorbeeld onderzoek naar middelen om kennistechnologen te ondersteunen in het modelleren van kennis of onderzoeken hoe verschillende bestanden over museum collecties met elkaar geïntegreerd en doorzoekbaar gemaakt kunnen worden. De architectuur van deze applicaties was een secundaire doelstelling in de projecten waarvoor ze zijn gebouwd, maar de zoektocht naar de bouwstenen van

een geschikte architectuur staat aan de basis van dit proefschrift. Tijdens de bouw van deze applicaties zijn naar behoefte herbruikbare uitbreidingen aan Prolog en zijn bibliotheken gemaakt. In opvolgende applicaties zijn deze uitbreidingen getest en daarna verfijnd of, indien nodig, vervangen. Stabiele uitbreidingen zijn beschreven in wetenschappelijke publicaties en gepubliceerd als Open Source. Terugkoppeling vanuit de Open Source gemeenschap levert suggesties op voor alternatieven of verdere verfijning. In deze studie besteden we in het bijzonder aandacht aan de volgende drie probleemgebieden:

- *Kennisrepresentatie*
  Hoewel Prolog een declaratieve lezing kent, heeft de taal een aantal beperkingen. Veel Prolog programmas die een correcte declaratieve representatie van het probleem zijn termineren niet of niet binnen een redelijke tijd. Frame geörienteerde talen (b.v., KL-ONE) voorzien in een vorm van kennisrepresentatie die beter geschikt is voor interactieve inspectie en manipulatie dan op logica gebaseerde talen.

  Begin deze eeuw is de semantische web gemeenschap gekomen met de kennisrepresentatietaal RDF. Deze taal heeft een zeer eenvoudig datamodel, bestaande uit triples van de vorm {onderwerp, predikaat, waarde}. Daarbovenop zijn een aantal meer expressieve talen gedefiniëerd: RDFS en diverse dialecten van OWL. Het eenvoudige relationele model van RDF past uitstekend op Prolog en vormt sindsdien de kern van onze kennisrepresentatie.

- *Grafische toepassingen*
  Veel Prolog systemen hebben geen voorzieningen voor grafische toepassingen en gebruiken een brug naar een conventionele grafische omgeving. Wij zijn van mening dat dit de productiviteit te veel schaadt, terwijl gebruik van de declaratieve aspecten van Prolog die ook zinvol zijn voor het beschrijven van interfaces en gebruikerinteracties bemoeilijkt wordt. Grafische omgevingen zijn doorgaans gestructureerd als objectgeoriënteerde systemen. Daarom hebben wij een uniforme koppeling met een extern object-georiënteerde systeem gerealiseerd en beschreven onder welke voorwaarden dezelfde architectuur voor andere object-georiënteerde systemen gebruikt kan worden.

- *Web-applicaties*
  In de afgelopen jaren heeft het web zich ontwikkeld van een web van documenten naar een web met interactieve applicaties. Er is onderzoek gaande hoe dit soort applicaties door toevoegen van kennis bruikbaarder gemaakt kunnen worden. Bovengenoemde integratie van museum bestanden is een voorbeeld. Mede doordat de voornoemde taal RDF bij uitstek geschikt is voor het integreren van diverse zeer verschillende kennisbestanden is het gebruik van Prolog voor de kennisaspecten van dit onderzoek een voor de hand liggende keuze. Als we Prolog echter inbedden in een architectuur die gebaseerd is op een traditionele webserver en pagina generatietaal (b.v., Apache met PHP) gaan veel van de voordelen voor snelle interactieve ontwikkeling verloren. Er is behoefte aan een meer 'Prolog-gezinde' manier om web-applicaties te maken.

Het is onze uitdaging om Prolog zodanig uit te breiden en van dusdanige bibliotheken te voorzien dat de taal geschikt is om grote en innovatieve applicaties geheel in Prolog te bouwen. In deze opzet worden alleen welbegrepen primitieven en benodigde interfaces naar besturingssysteem, netwerk en grafische primitieven in andere talen geschreven en als bibliotheek voor Prolog beschikbaar gesteld.

Overzicht van dit proefschrift Deel I van die proefschrift beschrijft uitbreidingen aan Prolog en bibliotheken voor Prolog die het mogelijk maken kennisintensieve interactieve applicaties geheel in Prolog te schrijven. Deze componenten worden waar mogelijk vergeleken met de aanpak voor soortgelijke problemen in andere (doorgaans Prolog) omgevingen. Deel II vormt een formatieve evaluatie van deze uitbreidingen: de beschreven cases tonen aan dat de gemaakte programmeeromgeving in staat is dit soort applicaties te ondersteunen, maar tegelijkertijd hebben zowel de bouw als de evaluatie van deze systemen geleid tot de ontwikkeling of verfijning van de uitbreidingen beschreven in deel I.

Deel I Hoofdstuk 2 beschrijft Triple20, een tool om ontologieën te browsen en beperkt te wijzigen. In strikte zin is Triple20 geen bouwsteen voor Prolog en hoort het dus niet thuis in deel I van dit proefschrift. Triple20 kan gebruikt worden als een bibliotheek, maar de voornaamste reden om dit systeem eerst te behandelen is dat dit hoofdstuk een goed inzicht heeft in de rol van kennis opgeslagen in het primitieve RDF model en hoe deze kennis gebruikt kan worden als fundament voor de architectuur van een interactieve applicatie. Het hoofdstuk behandelt een uitbreiding van het *model-view-controller* referentiemodel voor interactieve applicaties dat nodig is om de grote afstand in representatie tussen de kennisopslag en de interface efficiënt te overbruggen. Concurrency (threads; meerdere draden van executie) is hier een van de noodzakelijke bouwstenen. Triple20 vloeit voort uit de 'MIA' tools, beschreven in hoofdstuk 9.

De rest van deel I beschrijft de gerealiseerde uitbreiden aan Prolog en Prolog bibliotheken om ondermeer Triple20 mogelijk te maken. Het opslaan en opvragen van RDF triples, planning van complexe queries op deze database, koppelen van grafische bibliotheken, concurrency en infrastructuur voor het bouwen van web applicaties.

Het RDF triple model past naadloos op het relationele model van Prolog. Een triple database om onderzoek te doen naar eerder genoemde integratie van museum collecties dient minstens 10 miljoen triples kunnen bevatten, gelijktijdig vanuit meerdere threads gelezen kunnen worden en efficiënt kunnen zoeken naar tekst waarin sleutelwoorden voorkomen. RDF predikaten zijn geörganiseerd in een of meerdere hierarchieën welke in acht genomen dienen te worden om te kunnen redeneren over geïntegreerde bestanden. Dit type *entailment reasoning* dient ondersteund te worden met minimale overhead. Bij samengestelde zoekopdrachten (conjuncties) is de volgorde waarin de individuele RDF patronen vergeleken worden met de database van groot belang (vgl., database 'join' optimalisatie). Dit optimalisatieprobleem is het onderwerp van hoofdstuk 4 en is gebaseerd op metrieken van de RDF database. Een deel van bovengenoemde eisen is te specifiek om door de standaard Prolog dynamische database met maximale efficiëntie te kunnen worden opgevangen. De uiteindelijke

implementatie van de RDF database is onderwerp van hoofdstuk 3.

Hoofdstuk 5 gaat in op het koppelen van Prolog aan grafische bibliotheken. Zulke bibliotheken maken vrijwel zonder uitzondering gebruik van het object geörienteerde paradigma. Dit hoofdstuk behandelt eerst een compacte interface die het mogelijk maakt objecten vanuit Prolog te manipuleren. Binnen object geörienteerde omgevingen wordt nieuwe functionaliteit vaak gecreëerd door middel van afgeleide klassen. Bovenstaande interface geeft ons daar geen toegang toe vanuit Prolog, hetgeen zou betekenen dat het ontwikkelen van een grafisch programma in twee talen moet gebeuren: Prolog en de object taal om nieuwe klassen te maken. Dit schaadt de gewenste transparante ontwikkeling vanuit Prolog. Dit kan voor vrijwel alle object systemen opgelost worden door de nieuwe klasse in Prolog te beschrijven, waarbij de klasse declaraties gebruikt worden om een klasse binnen het object systeem te creëren en *wrappers* te genereren die de implementatie aanroepen binnen Prolog. Dit hoofdstuk vervolgt met integratie van Prolog data en non-determinisme in het object systeem en eindigt met een kritische evaluatie van dit object systeem voor Prolog.

Hoofdstuk 6 is een zijsprong naar concurrency binnen Prolog. Traditioneel is onderzoek naar concurrency binnen Prolog veelal gericht op hoe een enkel programma geschreven in Prolog door meerdere CPUs uitgevoerd kan worden. De declaratieve interpretatie van de taal geeft aanleiding te verwachten dat dit beter gaat dan met meer procedurele talen. Helaas heeft dit werk niet veel ontwikkelaars overtuigd. Onze eisen zijn meer bescheiden: achtergrond berekeningen in grafische omgevingen, schaalbaarheid van web-servers en benutten van recente *multi-core* hardware. Op deze basis is een eenvoudig model van samenwerkende Prolog machines bedacht dat inmiddels door twee andere Prolog implementaties is overgenomen en de basis is voor standaardisatie binnen ISO WG17. Hoofdstuk 6 beschrijft het model, de consequenties voor de implementatie van Prolog en twee performance evaluaties.

Hoofdstuk 7 geeft een overzicht van wat er binnen SWI-Prolog is gebeurd om het maken van web-applicaties mogelijk te maken. Dit hoofdstuk geeft een samenvatting van de voorgaande hoofdstukken over RDF en plaatst dit materiaal in het bredere kader van web applicaties. Speciale aandacht gaat uit naar het parseren en representeren van web documenten (HTML, XML) en het genereren van zulke documenten vanuit Prolog. Waar mogelijk wordt onze infrastructuur vergeleken met PiLLoW, een andere infrastructuur voor web-programeren binnen de Prolog gemeenschap. Ook wordt aandacht besteed aan web-servers (HTTP), waarbij met name code organisatie en schaalbaarheidsaspecten aan de orde komen. Hoofdstuk 7 benadrukt een aantal features die niet aan Prolog mogen ontbreken voor dit type applicaties: concurrency, geen limieten aan atomen, garbage collection van atomen en ondersteuning van de internationale UNICODE tekenset.

Deel II In het tweede deel worden een drietal applicaties besproken die hebben bijgedragen aan de ontwikkeling en evaluatie van de infrastructuur beschreven in deel I.

De eerste applicatie (PlDoc, hoofdstuk 8) is tegelijkertijd ook een deel van de SWI-Prolog infrastructuur. PlDoc is een omgeving voor *literate programming* in Prolog die, dankzij de embedded web-server, een interactieve omgeving biedt om documentatie te raadplegen en te verbeteren. PlDoc is onderdeel van de SWI-Prolog ontwikkeltools die noodzakelijk zijn voor

de productiviteit van de programmeeromgeving. Het bouwen van veel van deze tools heeft ook bijgedragen aan het testen en verfijnen van de infrastructuur, men name voor grafische applicaties.

Hoofdstuk 9 beschrijft experimenten met en de software architectuur van een prototype tool om annotatie en zoeken van multimediale bestanden met behulp van achtergrond kennis te bestuderen. Behalve XPCE (hoofdstuk 5) voor de grafische aspecten gebruikt dit prototype niets van de in deel I beschreven infrastructuur. Dit was de eerste tool waar RDF gebruikt werd om kennis uit te wisselen en waar het RDF datamodel tot in het design van de tool was doorgevoerd. In dit tool werd RDF opgeslagen in de Prolog dynamische database. Opname van dit hoofdstuk dient drie doelen: beschrijving van de applicatiecontext waarin tools worden ontwikkeld, benadrukken dat tool ontwikkeling en—in dit geval—onderzoek naar de rol van kennis in annotaties nauw geïntegreerd zijn en tot slot het formuleren van eisen voor de volgende generatie infrastructuur en tools. Dit werk vormde de directe aanleiding tot de ontwikkeling van Triple20 en een schaalbare RDF database, alsmede een extra motivatie om concurrency te introduceren.

Hoofdstuk 10 sluit deel II van dit proefschrift af met een beschrijving van de semantische zoek en annotatie toolkit ClioPatria. In dit hoofdstuk wordt de architectuur van ClioPatria, waarin alle technologie uit deel I samen komt[7] gemotiveerd vanuit een *use case*. De *use case* is het bouwen van een prototype applicatie om beschrijvingen van kunstwerken die musea hebben samen met beschikbare achtergrondinformatie zoals diverse databases van kunstenaars, kunsttermen en geografische informatie doorzoekbaar te maken. De grote diversiteit in dataformaten, dataschemas en teminologie die te vinden is in de diverse musea is een van de uitdagingen. Een andere belangrijke uitdaging is om geschikete methoden te vinden om deze data te doorzoeken. De struktuur van de geïntegreerde data is zo divers dat gestructureerde vragen (vgl., SQL queries) vrijwel niet te formuleren zijn. Om deze data toch doorzoekbaar te maken hebben we een algoritme ontwikkeld dat de data exploreert op basis van semantische afstand en de resultaten groepeert naar de semantische relatie tussen de zoekterm en het gevonden kunstwerk. ClioPatria biedt een moderne web-gebaseerde interactieve interface gebaseerd op AJAX technologie van Yahoo! (YUI). De server is een Prolog web-server die alle services verzorgt.

Discussie De assumptie van dit proefschrift is dat logisch programmeren, en specifiek Prolog, geschikt is voor het maken van kennisintensieve en interactieve software. De opkomst van RDF en daarop gebouwde kennisrepresentatietalen heeft de bruikbaarheid van Prolog geloofwaardiger gemaakt. Eigenschappen van RDF modellen, zoals schaal en gewenste inferenties, maken een implementatie in puur Prolog minder geschikt, maar implementatie als bibliotheek in een andere taal (C) kan dienen als een natuurlijke uitbreiding van Prolog. De opkomst van het web heeft grote invloed op de architectuur van applicaties en past, mits voorzien van de benodigde web-bibliotheken, uitstekend binnen het domein waarvoor Prolog geschikt is.

[7] De grafische interface wordt alleen gebruikt als onderdeel van de Prolog ontwikkelomgeving.

Naast verdere standaardisatie van Prolog is het mogelijk van nog groter belang dat de Prolog gemeenschap standaarden vaststelt voor het representeren en genereren van (web-)documenten in XML en HTML. Alleen met zulke standaarden wordt het mogelijk resources voor het maken van web applicaties binnen de gemeenschap te delen. RDF kan deze rol van nature op zich nemen voor de kenniskant omdat de manier waarop het RDF datamodel aan Prolog gekoppeld dient te worden niet veel keuzes laat.

Dit proefschrift is een direct gevolg van de werkwijze t.a.v. software ontwikkeling binnen de vakgroep SWI (nu HCS), waar software ontwikkeling voor onderzoek uitgevoerd wordt door ontwikkelaars met een eigen onderzoeksdoelstelling: software architectuur. Deze werkwijze heeft duidelijke voordelen: (1) het biedt de mogelijkheid om wetenschappelijk geïnteresseerde software ontwikkelaars te voorzien van uitdagingen op architectuur niveau in het kader van een concrete vraag naar een applicatie en (2) ontwikkeling van de applicatie die het oorspronkelijke onderzoeksdoel dient gebeurt vanuit een zeer directe betrokkenheid. Korte lijnen en een korte ontwikkel- en evaluatiecyclus is voor een grote klasse van softwareproducten een productieve werkwijze. Dit geldt zeker voor academische demonstrators.

Met SWI-Prolog hebben we een productieve omgeving gerealiseerd voor ons onderzoek. Daarnaast hebben we, dankzij de brede acceptatie van de Open Source gedachte en getuige de populariteit van SWI-Prolog, een bijdrage kunnen leveren aan het verspreiden van het gedachtengoed van logisch programmeren in een veel grotere kring van academische en commerciële gebruikers. Tegelijkertijd blijft deze gebruikersgemeenschap ons motiveren en van problemen en oplossingen voorzien.

SIKS Dissertatiereeks

1998

1998-1 Johan van den Akker (CWI)
DEGAS - An Active, Temporal Database of Autonomous Objects

1998-2 Floris Wiesman (UM)
Information Retrieval by Graphically Browsing Meta-Information

1998-3 Ans Steuten (TUD)
A Contribution to the Linguistic Analysis of Business Conversations within the Language/Action Perspective

1998-4 Dennis Breuker (UM)
Memory versus Search in Games

1998-5 E.W.Oskamp (RUL)
Computerondersteuning bij Straftoemeting

1999

1999-1 Mark Sloof (VU)
Physiology of Quality Change Modelling; Automated modelling of Quality Change of Agricultural Products

1999-2 Rob Potharst (EUR)
Classification using decision trees and neural nets

1999-3 Don Beal (UM)
The Nature of Minimax Search

1999-4 Jacques Penders (UM)
The practical Art of Moving Physical Objects

1999-5 Aldo de Moor (KUB)
Empowering Communities: A Method for the Legitimate User-Driven Specification of Network Information Systems

1999-6 Niek J.E. Wijngaards (VU)
Re-design of compositional systems

1999-7 David Spelt (UT)
Verification support for object database design

1999-8 Jacques H.J. Lenting (UM)
Informed Gambling: Conception and Analysis of a Multi-Agent Mechanism for Discrete Reallocation

2000

2000-1 Frank Niessink (VU)
Perspectives on Improving Software Maintenance

2000-2 Koen Holtman (TUE)
Prototyping of CMS Storage Management

2000-3 Carolien M.T. Metselaar (UvA)
Sociaal-organisatorische gevolgen van kennistechnologie; een procesbenadering en actorperspectief

2000-4 Geert de Haan (VU)
ETAG, A Formal Model of Competence Knowledge for User Interface Design

2000-5 Ruud van der Pol (UM)
Knowledge-based Query Formulation in Information Retrieval

2000-6 Rogier van Eijk (UU)
Programming Languages for Agent Communication

2000-7 Niels Peek (UU)
Decision-theoretic Planning of Clinical Patient Management

2000-8 Veerle Coup (EUR)
Sensitivity Analyis of Decision-Theoretic Networks

2000-9 Florian Waas (CWI)
Principles of Probabilistic Query Optimization

2000-10 Niels Nes (CWI)
Image Database Management System Design Considerations, Algorithms and Architecture

2000-11 Jonas Karlsson (CWI)
Scalable Distributed Data Structures for Database Management

2001

2001-1 Silja Renooij (UU)
Qualitative Approaches to Quantifying Probabilistic Networks

2001-2 Koen Hindriks (UU)
Agent Programming Languages: Programming with Mental Models

2001-3 Maarten van Someren (UvA)
Learning as problem solving

2001-4 Evgueni Smirnov (UM)
Conjunctive and Disjunctive Version Spaces with Instance-Based Boundary Sets

2001-5 Jacco van Ossenbruggen (VU)
Processing Structured Hypermedia: A Matter of Style

2001-6 Martijn van Welie (VU)
Task-based User Interface Design

2001-7 Bastiaan Schonhage (VU)

Diva: Architectural Perspectives on Information Visualization

2001-8 Pascal van Eck (VU)
A Compositional Semantic Structure for Multi-Agent Systems Dynamics

2001-9 Pieter Jan 't Hoen (RUL)
Towards Distributed Development of Large Object-Oriented Models, Views of Packages as Classes

2001-10 Maarten Sierhuis (UvA)
Modeling and Simulating Work Practice BRAHMS: a multiagent modeling and simulation language for work practice analysis and design

2001-11 Tom M. van Engers (VUA)
Knowledge Management: The Role of Mental Models in Business Systems Design

2002

2002-01 Nico Lassing (VU)
Architecture-Level Modifiability Analysis

2002-02 Roelof van Zwol (UT)
Modelling and searching web-based document collections

2002-03 Henk Ernst Blok (UT)
Database Optimization Aspects for Information Retrieval

2002-04 Juan Roberto Castelo Valdueza (UU)
The Discrete Acyclic Digraph Markov Model in Data Mining

2002-05 Radu Serban (VU)
The Private Cyberspace Modeling Electronic Environments inhabited by Privacy-concerned Agents

2002-06 Laurens Mommers (UL)
Applied legal epistemology; Building a knowledge-based ontology of the legal domain

2002-07 Peter Boncz (CWI)
Monet: A Next-Generation DBMS Kernel For Query-Intensive Applications

2002-08 Jaap Gordijn (VU)
Value Based Requirements Engineering: Exploring Innovative E-Commerce Ideas

2002-09 Willem-Jan van den Heuvel (KUB)
Integrating Modern Business Applications with Objectified Legacy Systems

2002-10 Brian Sheppard (UM)
Towards Perfect Play of Scrabble

2002-11 Wouter C.A. Wijngaards (VU)
Agent Based Modelling of Dynamics: Biological and Organisational Applications

2002-12 Albrecht Schmidt (Uva)
Processing XML in Database Systems

2002-13 Hongjing Wu (TUE)
A Reference Architecture for Adaptive Hypermedia Applications

2002-14 Wieke de Vries (UU)
Agent Interaction: Abstract Approaches to Modelling, Programming and Verifying Multi-Agent Systems

2002-15 Rik Eshuis (UT)
Semantics and Verification of UML Activity Diagrams for Workflow Modelling

2002-16 Pieter van Langen (VU)
The Anatomy of Design: Foundations, Models and Applications

2002-17 Stefan Manegold (UvA)
Understanding, Modeling, and Improving Main-Memory Database Performance

2003

2003-01 Heiner Stuckenschmidt (VU)
Ontology-Based Information Sharing in Weakly Structured Environments

2003-02 Jan Broersen (VU)
Modal Action Logics for Reasoning About Reactive Systems

2003-03 Martijn Schuemie (TUD)
Human-Computer Interaction and Presence in Virtual Reality Exposure Therapy

2003-04 Milan Petkovic (UT)
Content-Based Video Retrieval Supported by Database Technology

2003-05 Jos Lehmann (UvA)
Causation in Artificial Intelligence and Law - A modelling approach

2003-06 Boris van Schooten (UT)
Development and specification of virtual environments

2003-07 Machiel Jansen (UvA)
Formal Explorations of Knowledge Intensive Tasks

2003-08 Yongping Ran (UM)
Repair Based Scheduling

2003-09 Rens Kortmann (UM)
The resolution of visually guided behaviour

2003-10 Andreas Lincke (UvT)
Electronic Business Negotiation: Some experimental studies on the interaction between medium, innovation context and culture

2003-11 Simon Keizer (UT)
Reasoning under Uncertainty in Natural Language Dialogue using Bayesian Networks

2003-12 Roeland Ordelman (UT)
Dutch speech recognition in multimedia information retrieval

2003-13 Jeroen Donkers (UM)
Nosce Hostem - Searching with Opponent Models

2003-14 Stijn Hoppenbrouwers (KUN)
Freezing Language: Conceptualisation Processes across ICT-Supported Organisations

2003-15 Mathijs de Weerdt (TUD)
Plan Merging in Multi-Agent Systems

2003-16 Menzo Windhouwer (CWI)
Feature Grammar Systems - Incremental Maintenance of Indexes to Digital Media Warehouses

2003-17 David Jansen (UT)
Extensions of Statecharts with Probability, Time, and Stochastic Timing

2003-18 Levente Kocsis (UM)
Learning Search Decisions

2004

2004-01 Virginia Dignum (UU)
A Model for Organizational Interaction: Based on Agents, Founded in Logic

2004-02 Lai Xu (UvT)
Monitoring Multi-party Contracts for E-business

2004-03 Perry Groot (VU)
A Theoretical and Empirical Analysis of Approximation in Symbolic Problem Solving

2004-04 Chris van Aart (UvA)
Organizational Principles for Multi-Agent Architectures

2004-05 Viara Popova (EUR)
Knowledge discovery and monotonicity

2004-06 Bart-Jan Hommes (TUD)
The Evaluation of Business Process Modeling Techniques

2004-07 Elise Boltjes (UM)
Voorbeeldig onderwijs; voorbeeldgestuurd onderwijs, een opstap naar abstract denken, vooral voor meisjes

2004-08 Joop Verbeek (UM)
Politie en de Nieuwe Internationale Informatiemarkt, Grensregionale politiële gegevensuitwisseling en digitale expertise

2004-09 Martin Caminada (VU)
For the Sake of the Argument; explorations into argument-based reasoning

2004-10 Suzanne Kabel (UvA)
Knowledge-rich indexing of learning-objects

2004-11 Michel Klein (VU)
Change Management for Distributed Ontologies

2004-12 The Duy Bui (UT)
Creating emotions and facial expressions for embodied agents

2004-13 Wojciech Jamroga (UT)
Using Multiple Models of Reality: On Agents who Know how to Play

2004-14 Paul Harrenstein (UU)
Logic in Conflict. Logical Explorations in Strategic Equilibrium

2004-15 Arno Knobbe (UU)
Multi-Relational Data Mining

2004-16 Federico Divina (VU)
Hybrid Genetic Relational Search for Inductive Learning

2004-17 Mark Winands (UM)
Informed Search in Complex Games

2004-18 Vania Bessa Machado (UvA)
Supporting the Construction of Qualitative Knowledge Models

2004-19 Thijs Westerveld (UT)
Using generative probabilistic models for multimedia retrieval

2004-20 Madelon Evers (Nyenrode)
Learning from Design: facilitating multidisciplinary design teams

2005

2005-01 Floor Verdenius (UvA)
Methodological Aspects of Designing Induction-Based Applications

2005-02 Erik van der Werf (UM)
AI techniques for the game of Go

2005-03 Franc Grootjen (RUN)
A Pragmatic Approach to the Conceptualisation of Language

2005-04 Nirvana Meratnia (UT)
Towards Database Support for Moving Object data

2005-05 Gabriel Infante-Lopez (UvA)
Two-Level Probabilistic Grammars for Natural Language Parsing

2005-06 Pieter Spronck (UM)
Adaptive Game AI

2005-07 Flavius Frasincar (TUE)
Hypermedia Presentation Generation for Semantic Web Information Systems

2005-08 Richard Vdovjak (TUE)
A Model-driven Approach for Building Distributed Ontology-based Web Applications

2005-09 Jeen Broekstra (VU)
Storage, Querying and Inferencing for Semantic Web Languages

2005-10 Anders Bouwer (UvA)
Explaining Behaviour: Using Qualitative Simulation in Interactive Learning Environments

2005-11 Elth Ogston (VU)
Agent Based Matchmaking and Clustering - A Decentralized Approach to Search

2005-12 Csaba Boer (EUR)
Distributed Simulation in Industry

2005-13 Fred Hamburg (UL)
Een Computermodel voor het Ondersteunen van Euthanasiebeslissingen

2005-14 Borys Omelayenko (VU)
Web-Service configuration on the Semantic Web; Exploring how semantics meets pragmatics

2005-15 Tibor Bosse (VU)
Analysis of the Dynamics of Cognitive Processes

2005-16 Joris Graaumans (UU)
Usability of XML Query Languages

2005-17 Boris Shishkov (TUD)
Software Specification Based on Re-usable Business Components

2005-18 Danielle Sent (UU)
Test-selection strategies for probabilistic networks

2005-19 Michel van Dartel (UM)
Situated Representation

2005-20 Cristina Coteanu (UL)
Cyber Consumer Law, State of the Art and Perspectives

2005-21 Wijnand Derks (UT)
Improving Concurrency and Recovery in Database Systems by Exploiting Application Semantics

2006

2006-01 Samuil Angelov (TUE)
Foundations of B2B Electronic Contracting

2006-02 Cristina Chisalita (VU)
Contextual issues in the design and use of information technology in organizations

2006-03 Noor Christoph (UvA)
The role of metacognitive skills in learning to solve problems

2006-04 Marta Sabou (VU)
Building Web Service Ontologies

2006-05 Cees Pierik (UU)
Validation Techniques for Object-Oriented Proof Outlines

2006-06 Ziv Baida (VU)

Software-aided Service Bundling - Intelligent Methods & Tools for Graphical Service Modeling

2006-07 Marko Smiljanic (UT)
XML schema matching – balancing efficiency and effectiveness by means of clustering

2006-08 Eelco Herder (UT)
Forward, Back and Home Again - Analyzing User Behavior on the Web

2006-09 Mohamed Wahdan (UM)
Automatic Formulation of the Auditor's Opinion

2006-10 Ronny Siebes (VU)
Semantic Routing in Peer-to-Peer Systems

2006-11 Joeri van Ruth (UT)
Flattening Queries over Nested Data Types

2006-12 Bert Bongers (VU)
Interactivation - Towards an e-cology of people, our technological environment, and the arts

2006-13 Henk-Jan Lebbink (UU)
Dialogue and Decision Games for Information Exchanging Agents

2006-14 Johan Hoorn (VU)
Software Requirements: Update, Upgrade, Redesign - towards a Theory of Requirements Change

2006-15 Rainer Malik (UU)
CONAN: Text Mining in the Biomedical Domain

2006-16 Carsten Riggelsen (UU)
Approximation Methods for Efficient Learning of Bayesian Networks

2006-17 Stacey Nagata (UU)
User Assistance for Multitasking with Interruptions on a Mobile Device

2006-18 Valentin Zhizhkun (UvA)
Graph transformation for Natural Language Processing

2006-19 Birna van Riemsdijk (UU)
Cognitive Agent Programming: A Semantic Approach

2006-20 Marina Velikova (UvT)
Monotone models for prediction in data mining

2006-21 Bas van Gils (RUN)
Aptness on the Web

2006-22 Paul de Vrieze (RUN)
Fundaments of Adaptive Personalisation

2006-23 Ion Juvina (UU)
Development of Cognitive Model for Navigating on the Web

2006-24 Laura Hollink (VU)
Semantic Annotation for Retrieval of Visual Resources

2006-25 Madalina Drugan (UU)
Conditional log-likelihood MDL and Evolutionary MCMC

2006-26 Vojkan Mihajlović (UT)
Score Region Algebra: A Flexible Framework for Structured Information Retrieval

2006-27 Stefano Bocconi (CWI)
Vox Populi: generating video documentaries from semantically annotated media repositories

2006-28 Borkur Sigurbjornsson (UvA)
Focused Information Access using XML Element Retrieval

2007

2007-01 Kees Leune (UvT)
Access Control and Service-Oriented Architectures

2007-02 Wouter Teepe (RUG)
Reconciling Information Exchange and Confidentiality: A Formal Approach

2007-03 Peter Mika (VU)
Social Networks and the Semantic Web

2007-04 Jurriaan van Diggelen (UU)
Achieving Semantic Interoperability in Multi-agent Systems: a dialogue-based approach

2007-05 Bart Schermer (UL)
Software Agents, Surveillance, and the Right to Privacy: a Legislative Framework for Agent-enabled Surveillance

2007-06 Gilad Mishne (UvA)
Applied Text Analytics for Blogs

2007-07 Natasa Jovanovic' (UT)
To Whom It May Concern - Addressee Identification in Face-to-Face Meetings

2007-08 Mark Hoogendoorn (VU)
Modeling of Change in Multi-Agent Organizations

2007-09 David Mobach (VU)
Agent-Based Mediated Service Negotiation

2007-10 Huib Aldewereld (UU)
Autonomy vs. Conformity: an Institutional Perspective on Norms and Protocols

2007-11 Natalia Stash (TUE)
Incorporating Cognitive/Learning Styles in a General-Purpose Adaptive Hypermedia System

2007-12 Marcel van Gerven (RUN)
Bayesian Networks for Clinical Decision Support: A Rational Approach to Dynamic Decision-Making under Uncertainty

2007-13 Rutger Rienks (UT)
Meetings in Smart Environments; Implications of Progressing Technology

2007-14 Niek Bergboer (UM)
Context-Based Image Analysis

2007-15 Joyca Lacroix (UM)
NIM: a Situated Computational Memory Model

2007-16 Davide Grossi (UU)
Designing Invisible Handcuffs. Formal investigations in Institutions and Organizations for Multi-agent Systems

2007-17 Theodore Charitos (UU)
Reasoning with Dynamic Networks in Practice

2007-18 Bart Orriens (UvT)
On the development an management of adaptive business collaborations

2007-19 David Levy (UM)
Intimate relationships with artificial partners

2007-20 Slinger Jansen (UU)
Customer Configuration Updating in a Software Supply Network

2007-21 Karianne Vermaas (UU)
Fast diffusion and broadening use: A research on residential adoption and usage of broadband internet in the Netherlands between 2001 and 2005

2007-22 Zlatko Zlatev (UT)
Goal-oriented design of value and process models from patterns

2007-23 Peter Barna (TUE)
Specification of Application Logic in Web Information Systems

2007-24 Georgina Ramrez Camps (CWI)
Structural Features in XML Retrieval

2007-25 Joost Schalken (VU)
Empirical Investigations in Software Process Improvement

2008

2008-01 Katalin Boer-Sorbn (EUR)
Agent-Based Simulation of Financial Markets: A modular,continuous-time approach

2008-02 Alexei Sharpanskykh (VU)
On Computer-Aided Methods for Modeling and Analysis of Organizations

2008-03 Vera Hollink (UvA)
Optimizing hierarchical menus: a usage-based approach

2008-04 Ander de Keijzer (UT)
Management of Uncertain Data - towards unattended integration

2008-05 Bela Mutschler (UT)
Modeling and simulating causal dependencies on process-aware information systems from a cost perspective

2008-06 Arjen Hommersom (RUN)
On the Application of Formal Methods to Clinical Guidelines, an Artificial Intelligence Perspective

2008-07 Peter van Rosmalen (OU)
Supporting the tutor in the design and support of adaptive e-learning

2008-08 Janneke Bolt (UU)
Bayesian Networks: Aspects of Approximate Inference

2008-09 Christof van Nimwegen (UU)
The paradox of the guided user: assistance can be counter-effective

2008-10 Wauter Bosma (UT)
Discourse oriented summarization

2008-11 Vera Kartseva (VU)
Designing Controls for Network Organizations: A Value-Based Approach

2008-12 Jozsef Farkas (RUN)
A Semiotically Oriented Cognitive Model of Knowledge Representation

2008-13 Caterina Carraciolo (UvA)
Topic Driven Access to Scientific Handbooks

2008-14 Arthur van Bunningen (UT)
Context-Aware Querying; Better Answers with Less Effort

2008-15 Martijn van Otterlo (UT)
The Logic of Adaptive Behavior: Knowledge Representation and Algorithms for the Markov Decision Process Framework in First-Order Domains

2008-16 Henriette van Vugt (VU)
Embodied agents from a user's perspective

2008-17 Martin Op 't Land (TUD)
Applying Architecture and Ontology to the Splitting and Allying of Enterprises

2008-18 Guido de Croon (UM)
Adaptive Active Vision

2008-19 Henning Rode (UT)
From Document to Entity Retrieval: Improving Precision and Performance of Focused Text Search

2008-20 Rex Arendsen (UvA)
Geen bericht, goed bericht. Een onderzoek naar de effecten van de introductie van elektronisch berichtenverkeer met de overheid op de administratieve lasten van bedrijven

2008-21 Krisztian Balog (UvA)
People Search in the Enterprise

2008-22 Henk Koning (UU)
Communication of IT-Architecture

2008-23 Stefan Visscher (UU)
Bayesian network models for the management of ventilator-associated pneumonia

2008-24 Zharko Aleksovski (VU)
Using background knowledge in ontology matching

2008-25 Geert Jonker (UU)
Efficient and Equitable Exchange in Air Traffic Management Plan Repair using Spender-signed Currency

2008-26 Marijn Huijbregts (UT)
Segmentation, Diarization and Speech Transcription: Surprise Data Unraveled

2008-27 Hubert Vogten (OU)
Design and Implementation Strategies for IMS Learning Design

2008-28 Ildiko Flesch (RUN)
On the Use of Independence Relations in Bayesian Networks

2008-29 Dennis Reidsma (UT)
Annotations and Subjective Machines - Of Annotators, Embodied Agents, Users, and Other Humans

2008-30 Wouter van Atteveldt (VU)
Semantic Network Analysis: Techniques for Extracting, Representing and Querying Media Content

2008-31 Loes Braun (UM)
Pro-Active Medical Information Retrieval

2008-32 Trung H. Bui (UT)
Toward Affective Dialogue Management using Partially Observable Markov Decision Processes

2008-33 Frank Terpstra (UvA)
Scientific Workflow Design; theoretical and practical issues

2008-34 Jeroen de Knijf (UU)
Studies in Frequent Tree Mining

2008-35 Ben Torben Nielsen (UvT)
Dendritic morphologies: function shapes structure

2009

2009-01 Rasa Jurgelenaite (RUN)
Symmetric Causal Independence Models

2009-02 Willem Robert van Hage (VU)
Evaluating Ontology-Alignment Techniques

2009-03 Hans Stol (UvT)
A Framework for Evidence-based Policy Making Using IT

2009-04 Josephine Nabukenya (RUN)
Improving the Quality of Organisational Policy Making using Collaboration Engineering

2009-05 Sietse Overbeek (RUN)
Bridging Supply and Demand for Knowledge Intensive Tasks - Based on Knowledge, Cognition, and Quality

2009-06 Muhammad Subianto (UU)

	Understanding Classification
2009-07	Ronald Poppe (UT) Discriminative Vision-Based Recovery and Recognition of Human Motion
2009-08	Volker Nannen (VU) Evolutionary Agent-Based Policy Analysis in Dynamic Environments
2009-09	Benjamin Kanagwa (RUN) Design, Discovery and Construction of Service-oriented Systems