



UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

[Link to publication](#)

Citation for published version (APA):

Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

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 up to 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 ClíoPatria after extending the web service support with JSON (section 7.5). JavaScript libraries for vector graphics are available and used by ClíoPatria 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 PIDoc (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 remediate 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 XPath³ 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 form `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 $\text{rdf}(\text{Subject}, \text{Predicate}, \text{Object})$ 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 $\text{rdf}(\text{Subject}, \text{Predicate}, \text{Object})$ 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 JPL⁴ 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 loses 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 PIDoc 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. PIDoc 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 DBtune⁵ 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.