



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 8

PIDoc: 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Notes on a L^AT_EX backend in the future work section of the original paper have been replaced by a new section 8.7.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 PIDoc, a literate programming system for Prolog. Starting point for PIDoc 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 T_EX source (Knuth 1984) and the self documenting editor GNU-Emacs (Stallman 1981). Where the aim of the T_EX 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 JavaDoc¹ 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 `PIDoc` using and example, followed by a more detailed overview of the system. Section 8.5 tells the story of introducing `PIDoc` 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.

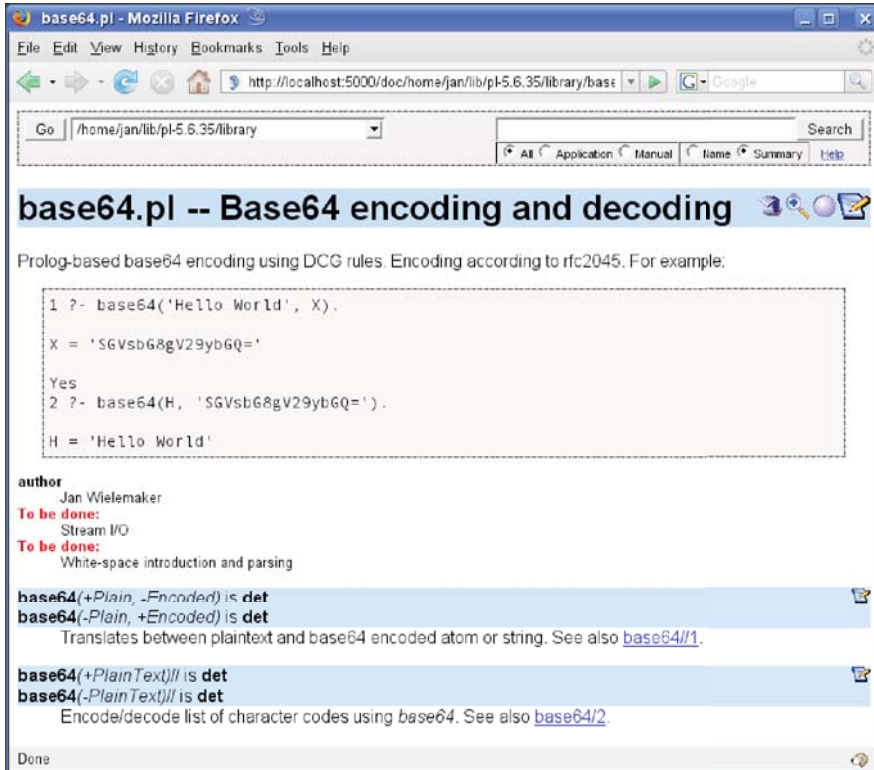


Figure 8.1: Documentation of library `base64.pl`. The example is described in section 8.3. The syntax $\langle name \rangle(\langle arg \rangle \dots) //$ and $\langle name \rangle // \langle arity \rangle$ define and reference grammar rules (DCG). Accessed from 'localhost', PLDoc provides edit (rightmost) 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 today's development environments. Web documents provide both comfortable online browsing and reasonable quality printing.

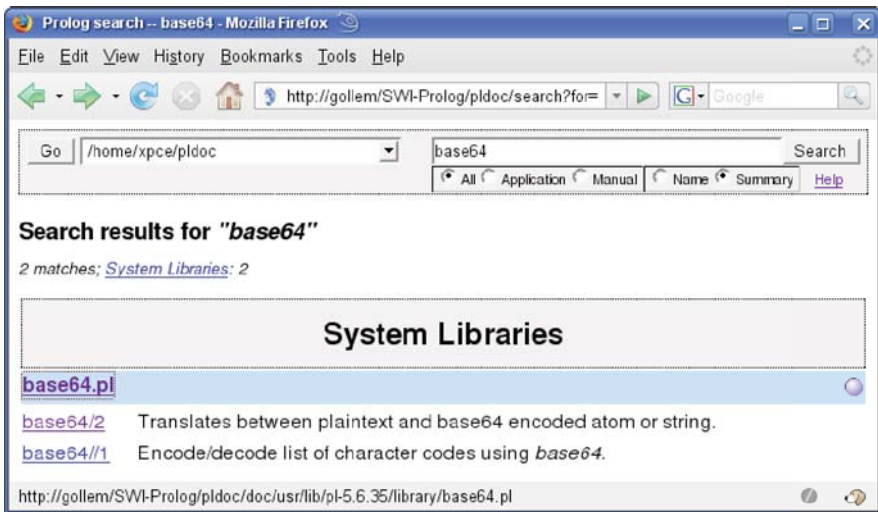


Figure 8.2: Searching for "base64"

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 PIDoc

8.4.1 The PIDoc syntax

PIDoc 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 PIDoc 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.

| | | |
|----------------------------|-----|---|
| <i><modedef></i> | ::= | <i><head></i> [<i>//</i>] [*] [<i>'is'</i> <i><determinism></i>] |
| <i><determinism></i> | ::= | <i>'det'</i> <i>'semidet'</i> <i>'nondet'</i> <i>'multi'</i> |
| <i><head></i> | ::= | <i><functor></i> '(<i><argspec></i> { <i>'</i> , <i><argspec></i> }) [*] <i><atom></i> |
| <i><argspec></i> | ::= | [<i><mode></i>] <i><argname></i> [<i>'</i> : <i>'</i> <i><type></i>] |
| <i><mode></i> | ::= | <i>'+</i> <i>'-</i> <i>'?'</i> <i>':</i> ' <i>'@'</i> <i>'!</i> ' |
| <i><type></i> | ::= | <i><term></i> |

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 proposes³ '=' 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 <i>partial term</i> 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 <code>setarg/3</code> or <code>nb_setarg/3</code> . |

Figure 8.5: Defined modes

The body of a description is given to a Prolog defined wiki parser based on Twiki⁴ 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/maillinglist/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 PIDoc manual.⁵

8.4.2 Publishing the documentation

PIDoc 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’, PIDoc 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>

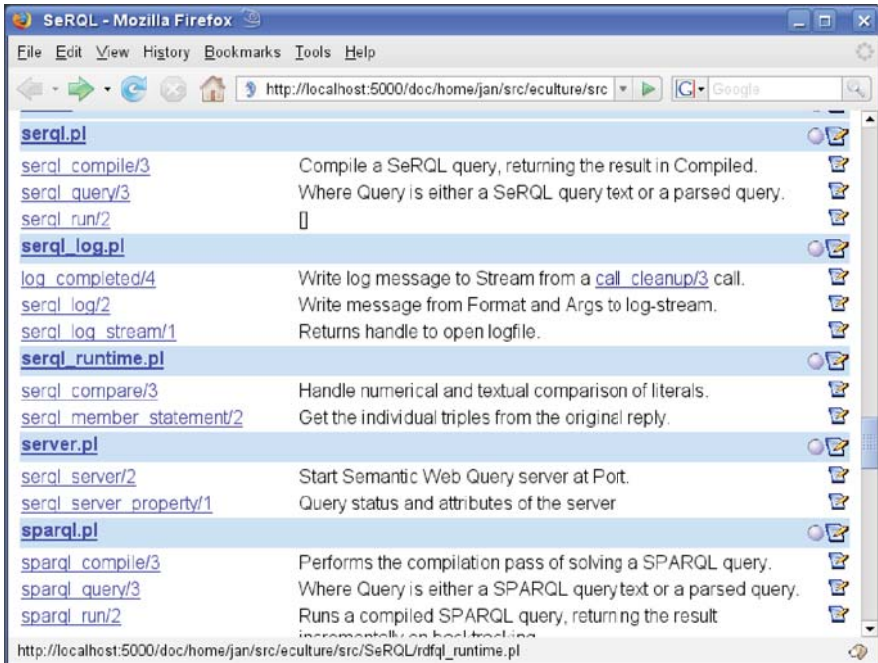


Figure 8.6: PIDoc displaying a directory index with files and their public predicates accessed from 'localhost'. Each predicate has an 'edit' button and each file a pretty print button (blue circle, see section 8.4.4)

In addition the browser interface shows a 'reload' button to run `make/0` and refreshes the current page, reflecting the edited documentation.

Initially, PIDoc 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 PIDoc 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 PIDoc 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 were caused by comments that should not have been turned into structured comments. PIDoc'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, PIDoc 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 XPCE⁷ code for the user interface.

Before the introduction of the PIDoc 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 PIDoc all new tOKo functionality is being documented using the PIDoc 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/>

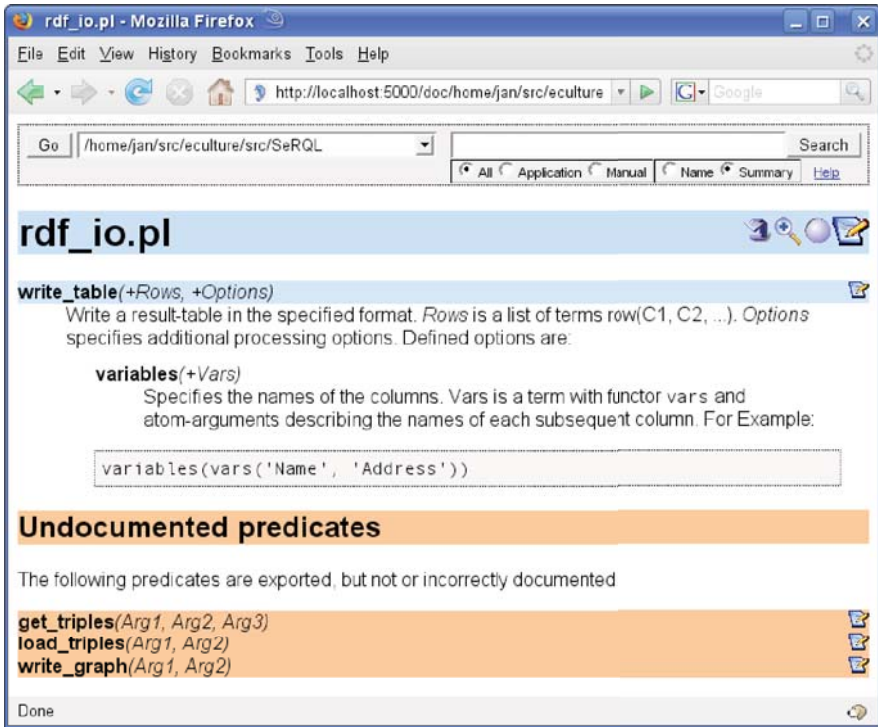


Figure 8.7: Undocumented public predicates are added at the bottom. When accessed from 'localhost', the developer can click the *edit* icon, add or fix documentation and click the *reload* icon at the top of the page to view the updated documentation.

in particular the immediate reward for the programmer. The intuitive notation of PIDoc also makes it relatively easy to add the documentation. The Emacs Prolog mode developed for SWI-Prolog⁸ 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 PIDoc 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 PIDoc 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 Ipdoc 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 T_EX-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 ECLiPSe⁹ 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.

PrologDoc¹⁰ 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 website¹¹ 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 L^AT_EX backend

Although the embedded HTTP server backend is our primary target, PLDoc is capable of producing L^AT_EX files from both Prolog files and plain text files using Wiki style markup as described in section 8.4.1. The L^AT_EX backend can be used to create a standalone L^AT_EX 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 L^AT_EX and the generated fragments are included using L^AT_EX `\input` or `\include` statements. This approach provides flexibility by distributing documentation source at will over pure L^AT_EX, 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 L^AT_EX 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 concatenation 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 PIDoc 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 PIDoc

PIDoc 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 PIDoc 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. PIDoc 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. PIDoc can be integrated in web applications, creating a self-documenting web server. We use this in ClioPatria (chapter 10). The secondary backend is L^AT_EX, which can be used to include documentation extracted from Prolog source files into L^AT_EX documents.

PIDoc 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, PIDoc cannot be called *knowledge-intensive*. In part, this was a deliberate choice to keep the footprint of PIDoc small. In the future we may consider more advanced search features that could be based on vocabularies from computer science such as FOLDOC¹² 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 PIDoc with support for knowledge-based search in the system and application documentation.

Acknowledgements

Although the development of PIDoc 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>

