PlDoc: Wiki style Literate Programming for Prolog

Jan Wielemaker[1] and Anjo Anjewierden[2]

[1] Human-Computer Studies Laboratory, University of Amsterdam,
Kruislaan 419, 1098 VA Amsterdam, The Netherlands,
`wielemak@science.uva.nl`

[2] Department of Instructional Technology, Faculty of Behavourial Sciences,
University of Twente,
PO Box 217, 7500 AE Enschede, The Netherlands
`a.a.anjewierden@utwente.nl`

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.

1 Introduction

Combining source and documentation in the same file, generally named *literate programming*, is an old idea. Classical examples are the TEX source [7] and the self documenting editor GNU-Emacs [15]. 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. A more recent success story is JavaDoc[3].

There is an overwhelming amount of articles on literate programming, most of which describe an implementation or qualitative experience using a literate programming system [12]. Shum and Cook [14] 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, subjects using literate programming (in this case AOPS, [13]) was considered confusing and harmed debugging the program.

Recent developments in programming environments and methodologies make a case for re-introducing literate programming [11]. The success of systems such

[3] http://java.sun.com/j2se/javadoc/

as Doxygen [16] based on some form of structured comments in the source code, making the literate programming document a valid document for the programming language is evident. Using a source document that is valid for the programming language ensures smooth integration with tools 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" [7]. Embedding documentation source code comments merely produces an *API Reference Manual*.

In the Prolog world we see lpdoc [5], documentation support in the Logtalk [9] language and the ECLiPSe Document Generation Tools[4] 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.

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 5 tells the story of introducing PlDoc in a large open source program, while we compare our work to related projects in Sect. 6.

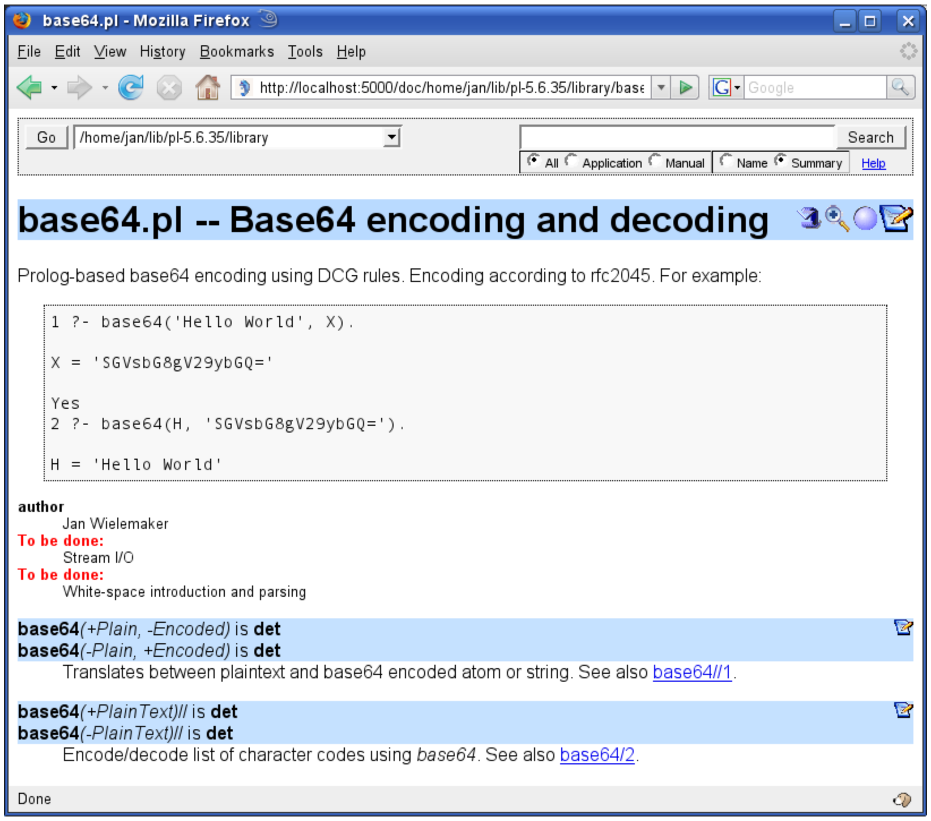
2 An attractive literate programming environment

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 documentation biased grading system in education [14] 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 choicepoints and motivate our primary choices from these starting points.

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 to-

[4] http://eclipse.crosscoreop.com/doc/userman/umsroot088.html


base64.pl - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/doc/home/jan/lib/pl-5.6.35/library/base
Google
Go
/home/jan/lib/pl-5.6.35/library
Search
All Application Manual Name Summary Help
Done

base64.pl -- Base64 encoding and decoding

Prolog-based base64 encoding using DCG rules. Encoding according to rfc2045. For example:

```
1 ?- base64('Hello World', X).

X = 'SGVsbG8gV29ybGQ='

Yes
2 ?- base64(H, 'SGVsbG8gV29ybGQ=').

H = 'Hello World'
```

author
Jan Wielemaker
To be done:
Stream I/O
To be done:
White-space introduction and parsing

base64(*+Plain, -Encoded*) is **det**
base64(*-Plain, +Encoded*) is **det**
Translates between plaintext and base64 encoded atom or string. See also base64//1.

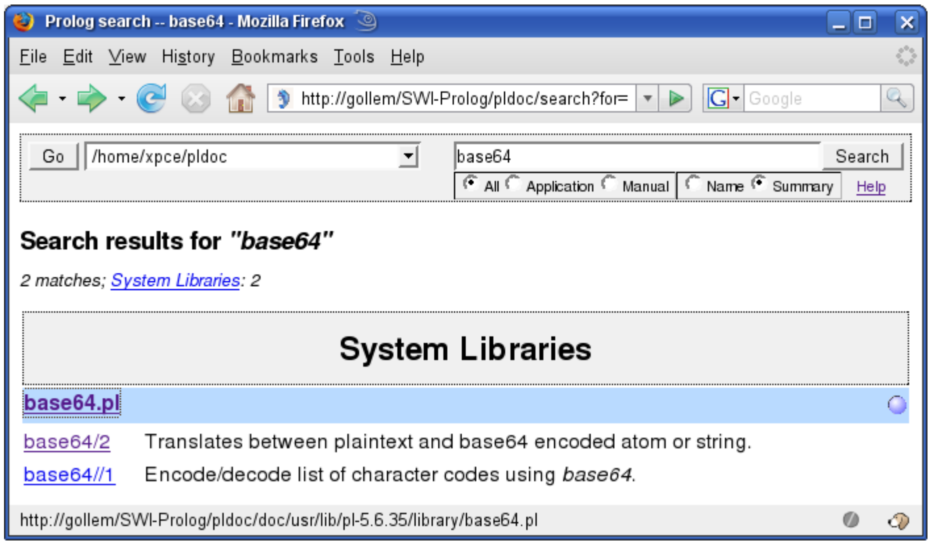
base64(*+PlainText*)// is **det**
base64(*-PlainText*)// is **det**
Encode/decode list of character codes using *base64*. See also base64/2.

Fig. 1. Documentation of library base64.pl. Accessed from ‘localhost’, PlDoc provides edit and reload buttons.

wards 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.

In this search space we aim at a system with little overhead for the programer 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 very 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.



http://gollem/SWI-Prolog/pldoc/search?for=

/home/xpce/pldoc

base64

[✓ All] Application Manual | Name [✓ Summary]

Search results for *"base64"*

2 matches; System Libraries: 2

System Libraries

base64.pl

base64/2	Translates between plaintext and base64 encoded atom or string.
base64//1	Encode/decode list of character codes using *base64*.

http://gollem/SWI-Prolog/pldoc/doc/usr/lib/pl-5.6.35/library/base64.pl

Fig. 2. Searching for "base64"

Minimising impact as well as maximising portability made us decide against the approach of lpdoc, ECLiPSe and Logtalk 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) and use Wiki [8] structuring conventions extended with Prolog conventions such as referencing a predicate using ⟨*name*⟩/⟨*arity*⟩. Wiki is a simple plain-text format designed for collaborative interactive management of web pages. Wikis differ in the details on the text format. We are particularly interested in wiki formats based on common practice simulating font and structuring conventions in plain text such as traditional email, usenet and comments in source code.

Maximal reward to the programmer A system is more easily accepted if it not only provides reward for the users of the software module, but also to the programmer him/herself. We achieve this by merging the documentation of the loaded Prolog code with the Prolog manuals in a consistent view presented from a web server embedded in the development environment. This relieves the programmer from making separate searches in the manuals and other parts of system under development.

Immediate reward to the programmer Humans love to be rewarded immediately. This implies the results must be accessible directly. 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, maintaining a Prolog database as described in Sect. 4.2. This database is made available to the developer through a web server running in a separate thread (Sect. 4.3). The SWI-Prolog **make/0** comment updates the running Prolog system to the latest version of the loaded sources and updates the web site at the same time.

3 An example

Before going into detail we show the documentation process and access for the SWI-Prolog library `base64.pl`, providing 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 Fig. 3. We see a number of special constructs:

– The `/** <module> Title` comment introduces overall documentation of the module. Inside, the `==` delimited lines start a source code block. The `@`*keyword* section provides JavaDoc inspired keywords from a fixed and well defined set (see end of Sect. 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. This is followed by wiki text, 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. Keyword search on format aspects and a summary line have a long history, for example in the Unix `man` command.

4 Description of PlDoc

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 with a single `%`. Using `/**` style comments the header is ended by a blank line. The header is either "$\langle$*module*$\rangle$ *Title*" or one or more predicate head declarations. The $\langle$*module*$\rangle$ syntax can be extended easily.

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 Fig. 4.

The optional `//`-postfix indicate $\langle$*head*$\rangle$ is a DCG rule. The *determinism* values originate from Mercury [6]. Predicates marked as `det` must succeed exactly once and not leave any choice points. The `semidet` indicator is used for predicates that either fail or succeed deterministically. The `nondet` indicator is the

```
/** <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) --> ...
```

Fig. 3. Commented source code of library base64.pl

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

Fig. 4. BNF for predicate header

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 Fig. 5. Originating from DEC-10 Prolog were the *mode* indicators (`+`,`-`,`?`) had a formal meaning. The ISO standard [4] adds '`@`', meaning "the argument shall remain unaltered". Quintus added '`:`', meaning the argument is module sensitive. Richard O'Keefe proposes[5] '`=`' 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**.

Fig. 5. Defined modes

The body of a description is given to a Prolog defined wiki parser based on Twiki[6] 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 Fig. 3) as such blocks are frequent and not easily supported by Twiki.
- We added automatic links for ⟨*name*⟩/⟨*arity*⟩, ⟨*name*⟩//⟨*arity*⟩, ⟨*file*⟩.pl, ⟨*file*⟩.txt (interpreted as wiki text) and image files using image extensions. Using `[[`file.png`]]`, inline images can be produced.
- Capitalised words appearing in the running text that match exactly one of the arguments are typeset in *italics*.
- We do not process embedded HTML. One of the reasons is that we want the option for other target languages. Opening up the path to unlimited use of

[5] http://gollem.science.uva.nl/SWI-Prolog/mailinglist/archive/2006/q1/0267.html
[6] http://www.twiki.org

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 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.[7].

4.2 Processing the comments

We claimed immediate reward as an important asset. This implies the documentation must be an integral part of the development environment. SWI-Prolog aims at providing IDE modules while allowing the user to use an editor or IDE of choice. An obvious choice is to make the compiler collect comments and present these to the user through a web interface. This is achieved using a hook in 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.

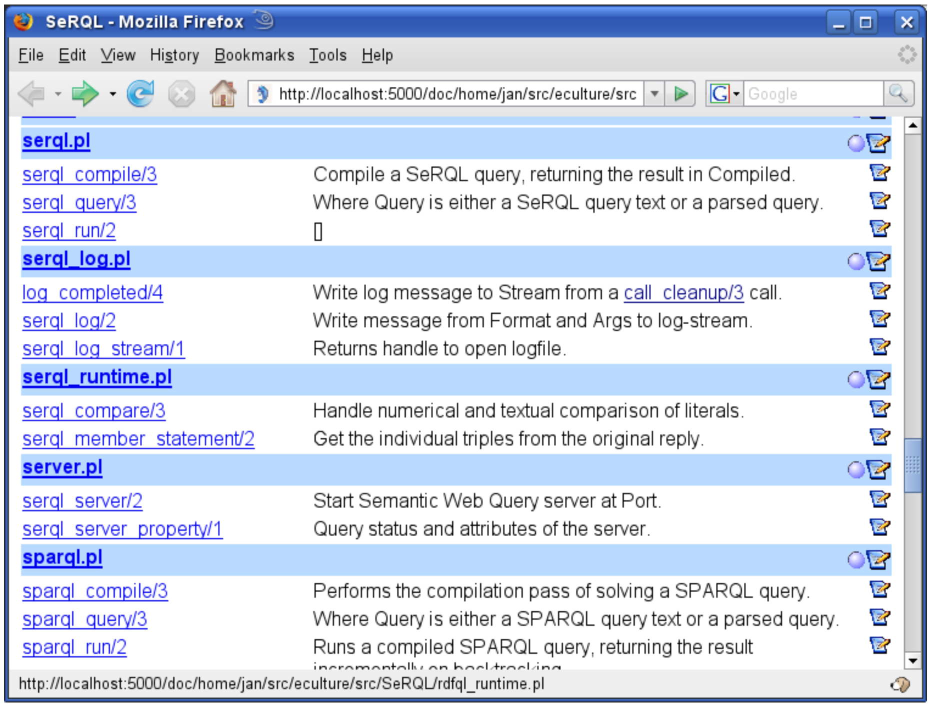
The hook installed by the documentation server extracts structured comments by checking for `%%` or `/**`. For structured comments it extracts the formal comment header and the first line of the comment body which serves, like JavaDoc, as a *summary*. The formal part is processed and the entire structured comment is stored unparsed, but is associated with the parsed formal header and summary which are used for linking the comment with a predicate as well as keyword search. 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.

4.3 Publishing the documentation

PlDoc realises a web application using the SWI-Prolog HTTP infrastructure [17]. Running in a separate thread, the normal interactive Prolog toplevel is not affected. The documentation server can also be used from an embedded Prolog

[7] http://www.swi-prolog.org/packages/pldoc.html

system. By default access is granted to 'localhost' only. Using additional options to **doc_server**(*+Port, +Options*), access can be granted to a wider public. A scenario for exploiting this is to have a central Prolog process with all resources available to a team loaded. Regularly running update from a central repository and **make/0** inside Prolog, it can serve as an up-to-date and searchable central documentation source. Since September 15 2006, we host such a 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 2007), the server handles approximately 100 search requests (1,000 page views) per day.


SeRQL - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/doc/home/jan/src/eculture/src
Google

serql.pl	
serql_compile/3	Compile a SeRQL query, returning the result in Compiled.
serql_query/3	Where Query is either a SeRQL query text or a parsed query.
serql_run/2	[]
serql_log.pl	
log_completed/4	Write log message to Stream from a call_cleanup/3 call.
serql_log/2	Write message from Format and Args to log-stream.
serql_log_stream/1	Returns handle to open logfile.
serql_runtime.pl	
serql_compare/3	Handle numerical and textual comparison of literals.
serql_member_statement/2	Get the individual triples from the original reply.
server.pl	
serql_server/2	Start Semantic Web Query server at Port.
serql_server_property/1	Query status and attributes of the server.
sparql.pl	
sparql_compile/3	Performs the compilation pass of solving a SPARQL query.
sparql_query/3	Where Query is either a SPARQL query text or a parsed query.
sparql_run/2	Runs a compiled SPARQL query, returning the result

http://localhost:5000/doc/home/jan/src/eculture/src/SeRQL/rdfql_runtime.pl

Fig. 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 Sect. 4.5)

4.4 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 [10], 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. 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 README file (if any) and all Prolog files with a summary listing of the public predicates as illustrated in Fig. 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 Fig. 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.

4.5 Presentation options

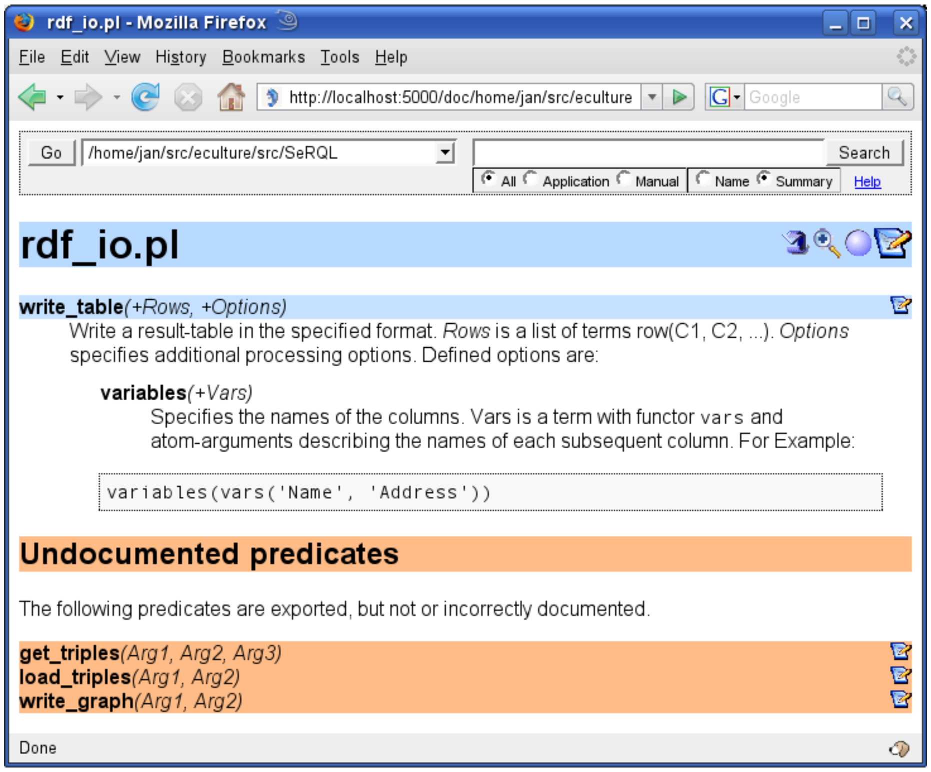
By default, PlDoc only shows public predicates when displaying a file or directory index. This can be changed using the 'zoom' button displayed with every page. Showing documentation on internal predicates proves helpful for better understanding of a module and helps finding opportunities for code reuse. 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[8]. The colouring engine uses **read_term/3** with options 'subterm_positions' to get the term layout compatible to Quintus Prolog [1] and 'comments' to get comments and their positions in the source file.

5 User experiences

tOKo [2] 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.

[8] http://www.swi-prolog.org/emacs.html


rdf_io.pl - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://localhost:5000/doc/home/jan/src/eculture
Google
Go
/home/jan/src/eculture/src/SeRQL
Search
All Application Manual Name Summary Help
Done

rdf_io.pl

write_table(*+Rows, +Options*)

Write a result-table in the specified format. *Rows* is a list of terms row(C1, C2, ...). *Options* specifies additional processing options. Defined options are:

variables(*+Vars*)

Specifies the names of the columns. Vars is a term with functor `vars` and atom-arguments describing the names of each subsequent column. For Example:

```
variables(vars('Name', 'Address'))
```

Undocumented predicates

The following predicates are exported, but not or incorrectly documented.

get_triples(*Arg1, Arg2, Arg3*)
load_triples(*Arg1, Arg2*)
write_graph(*Arg1, Arg2*)

Fig. 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.

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[9] 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.

[9] http://www.swi-prolog.org/packages/xpce/

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, 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[10] automatically reformats the documentation, such that mixing code and documentation becomes natural after a very 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**, 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.

6 Related work

The lpdoc system [5] 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 [3]. 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 in the Logic Programming domain and is not easily expanded. The required $\TeX$ based infrastructure and way of thinking no longer is a given.

In Logtalk [9], 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 struc-

[10] http://turing.ubishops.ca/home/bruda/emacs-prolog

ture information embedded in the XML can be used by other tools to reason about Logtalk programs.

The ECLiPSe[11] 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[12] is a Prolog version of JavaDoc. It stays close to JavaDoc, heavily relying on '@'-keywords and using HTML for additional markup. Figure 8 gives an example.

```
/**
    @form member(Value,List)
    @constraints
    @ground Value
    @unrestricted List
    @constraints
        @unrestricted Value
        @ground List
    @descr True if Value is a member of List
*/
```

Fig. 8. An example using PrologDoc

Outside the Logic Programming domain there is a large set of literate programming tools. A good example, the website of which contains a lot of information on related systems, is Doxygen [16]. Most of the referenced systems use structured comments.

7 Extending and porting PlDoc

Although to us the embedded HTTP server backend is the primary target, PlDoc will be extended with backends for static HTML files (partially realised). PlDoc is primarily an API documentation system. It is currently not very suitable for generating a book. Such functionality is highly desirable for dealing with the SWI-Prolog system documentation, maintained in LaTeX. We will investigate the possibility to introduce a LaTeX macro that will extract the documentation of a file or single predicate and insert it into the LaTeX text. For example:

```
\begin{description}
    \pldoc{member}{2}
    \pldoc{length}{2}
\end{description}
```

[11] http://eclipse.crosscoreop.com/doc/userman/umsroot088.html
[12] http://prologdoc.sourceforge.net/

PlDoc is Open Source and can 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[13], 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 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.

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.

References

1. AI International ltd., Berkhamsted, UK. *Quintus Prolog, User Guide and Reference Manual*, 1997.
2. Anjo Anjewierden, Bob Wielinga, and Robert de Hoog. Task and domain ontologies for knowledge mapping in operational processes. Metis Deliverable 4.2/2003, University of Amsterdam, 2004. tOKo home: http://www.toko-sigmund.org/.
3. Robert J. Chassell and Richard M. Stallman. *Texinfo: The GNU Documentation Format.* Reiters.com, 1999.
4. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.

[13] http://www.sju.edu/ jhodgson/wg17/projects.html

5. Manuel V. Hermenegildo. A documentation generator for (c)lp systems. In John W. Lloyd, Verónica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic*, volume 1861 of *Lecture Notes in Computer Science*, pages 1345–1361. Springer, 2000.
6. David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in mercury. In *ACSC*, pages 128–135. IEEE Computer Society, 2000.
7. Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984.
8. B. Leuf and W. Cunningham. *The Wiki Way: Collaboration and Sharing on the Internet*. Addison-Wesley, 2001.
9. Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.
10. Linda Dailey Paulson. Building Rich Web Applications with Ajax. *IEEE Computer*, 38(10):14–17, 2005.
11. Vreda Pieterse, Derrick G. Kourie, and Andrew Boake. 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*, pages 2–9, , Republic of South Africa, 2004. South African Institute for Computer Scientists and Information Technologists.
12. Norman Ramsey and Carla Marceau. Literate programming on a team project. *Software - Practice and Experience*, 21(7):677–683, 1991.
13. A. Shum and C. Cook. Aops: an abstraction-oriented programming system for literateprogramming. *Software Engineering Journal*, 8(3):113–120, 1993.
14. Stephen Shum and Curtis Cook. Using literate programming to teach good programming practices. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pages 66–70, New York, NY, USA, 1994. ACM Press.
15. Richard M. Stallman. Emacs the extensible, customizable self-documenting display editor. *SIGPLAN Not.*, 16(6):147–156, 1981.
16. D van Heesch. *Doxygen, a documentation system for C++*, 2007. http://www.stack.nl/ dimitri/doxygen/.
17. Jan Wielemaker, Zhisheng Huang, and Lourens van der Mey. SWI-Prolog and the Web. Paper submitted to tplp, HCS, University of Amsterdam, 2006.