SWI-Prolog Source Documentation

Jan Wielemaker
HCS,
University of Amsterdam
The Netherlands
E-mail: wielemak@science.uva.nl

January 23, 2009

Abstract

This article presents PlDoc, the SWI-Prolog source-code documentation infrastructure. PlDoc is loosely based on JavaDoc, using structured comments to mix documentation with source-code. SWI-Prolog's PlDoc is entirely written in Prolog and well integrated into the environment. It can create HTML+CSS and LaTeX documentation files as well as act as a web-server for the loaded project during program development.

A demo server, serving all standard libraries from their source as well as the Prolog Reference Manual and package documentation is hosted at http://gollem.science.uva.nl/SWI-Prolog/pldoc/

Contents

1 Introduction

When developing Prolog source that has to be maintained for a longer period or is developed by a —possibly distributed— team some basic quality mechanisms need to be adopted. A shared and well designed codingstyle is one of them. In addition, documentation of source-files and their primary interfaces as well as a testing framework must be established.

In our view, hitherto existing documentation and testing frameworks fell short realising the basic needs in a lightweight and easy to adopt system. To encourage consistent style, well commented code and test-assisted development, we make sure that

- The documentation and testing framework requires a minimum of work and learning.
- The framework is immediately rewarding to the individual programmer as well as the team,

First, we describe the documentation system we developed for SWI-Prolog. In section 11 we motivate our main choices.

2 Overview

Like JavaDoc, the PlDoc infrastructure is based on *structured comments*. Using comments, no changes have to be made to Prolog to load the documented source. If the `pldoc` library is loaded, Prolog will not only load the source, but also parse all structured comments. It processes the mode-declarations inside the comments and stores these as annotations in the Prolog database to support the test framework and other runtime and compiletime analysis tools that may be developed in the future.

Documentation for all or some of the loaded files can be written to file in either HTML+CSS or LaTeX (see section 10.4) format. Each source file is documented in a single file. In addition, the documentation generator will generate an index file that can be used as an index for a browser or input file for LaTeX for producing nicely typeset document.

To support the developer, the documentation system can be asked to start a web-server that can be used to browse the documentation.

3 Structured comments

Structured comments come in two flavours, the line-comment (%) based one, seen mostly in the Prolog community and the block-comment (`/*...*/`) based one, commonly seen in the Java and C domain. As we cannot determine the argument-names, type and modes from following (predicate) code itself, we must supply this in the comment.[1]. The overall structure of the comment therefore is:

- Semi-formal type- and mode-description, see section 5
- Wiki-style documentation body, see section 7
- JavaDoc style tags (`@`keyword value, see section 6)

Using the `/*...*/` style comment, the type and mode declarations are ended by a blank line. Using `%` line-comments, the declaration is ended by the first line that starts with a single %.

The JavaDoc style keyword list starts at the first line starting with @⟨*word*⟩.

[1] See section 11.

4 File (module) comments

An important aspect is documentation of the file or module as a whole, explaining its design, purpose and relation to other modules. In JavaDoc this is the comment that preceeds the class definition. The Prolog equivalent would be to put the module comment in front of the module declaration. The module declaration itself however is an important index to the content of the file and is therefore best kept first.

The general comment-structure for module comments is to use a type identifier between angled brackets, followed by the title of the section. Currently the only type provided is `module`. Other types may be added later.

Example

```
/** <module> Prolog documentation processor

This module processes structured comments and generates both formal
mode declarations from them as well as documentation in the form of
HTML or LaTeX.

@author Jan Wielemaker
@license GPL
*/
```

5 Type and mode declarations

The type and mode declaration header consists of one or more Prolog terms. Each term describes a mode of the predicate. The syntax is informally described below:

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

The *determinism* values originate from Mercury. 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 most general one and implies 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*.

Instantiation patterns are:

+	Argument must be fully instantiated to a term that satisfies the type.
-	Argument must be unbound.
?	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`.

In the current version types are represented by an arbitrary term without formal semantics. In future versions we may adopt a formal type system that allows for runtime verification and static type analysis [Hermenegildo, 2000, Mycroft & O'Keefe, 1984, Jeffery *et al.*, 2000]

Examples

```
%%      length(+List:list, -Length:int) is det.
%%      length(?List:list, -Length:int) is nondet.
%%      length(?List:list, +Length:int) is det.
%
%       True if List is a list of length Length.
%
%       @compat iso
```

6 Tags: @param, etc.

Optionally, the description may be followed by one or more *tags*. Our tag convention is strongly based on the conventions of javaDoc. It is adviced to place tags in the order they are described below.

@**param** *Name Description*
: Defines the parameters. Each parameter has its own param tag. The first word is the name of the parameter. The remainder of the tag is the description. Parameter declarations must appear in the argument order used by the predicate.

@**throws** *Term Description*
: Error condition. First Prolog term is the error term. Remainder is the description.

@**error** *Error Description*
: As @throws, but the exception is embedded in `error`(*Error, Context*).

@**author** *Name*
: Author of the module or predicate. Multiple entries are used if there are multiple authors.

@**version** *Version*
: Version of the module. There is no formal versioning system.

@**see** *Text*
: Point to related material. Often contains links to predicates or files.

`@`**deprecated** *Alternative*
: The predicate or module is deprecated. The description specifies what to use in new code.

`@`**compat** *Standards and systems*
: When implementing libraries or externally defined interfaces this tag describes to which standard the interface is compatible.

`@`**copyright** *Copyright holder*
: Copyright notice.

`@`**license** *License conditions*
: License conditions that apply to the source.

`@`**bug** *Bug description*
: Known problems with the interface or implementation.

`@`**tbd** *Work to be done*
: Not yet realised behaviour that is enticipated in future versions.

7 Wiki notation

Structured comments that provide part of the documentation are written in Wiki notation, based on TWiki, with some Prolog specific additions.

7.1 Structuring conventions

Paragraphs Paragraphs are seperated by a blank line.

General lists The wiki knows three types of lists: *bullet lists* (HTML `ul`), *numbered lists* (HTML `ol`) and *description lists* (HTML `dl`). Each list environment is headed by an empty line and each list-item has a special symbol at the start, followed by a space. Each subsequent item must be indented at exactly the same column. Lists may be nested by starting a new list at a higher level of indentation. The list prefixes are:

`*`	Bulleted list item
`1.`	Numbered list item. Any number from 1..9 is allowed, which allows for proper numbering in the source. Actual numbers in the HTML or LaTeX however are re-generated, starting at 1.
`$ Title : Item`	Description list item.

Term lists Especially when describing option lists or different accepted types, it is common to describe the behaviour on different terms. Such lists must be written as below. ⟨*Term1*⟩, etc. must be valid Prolog terms and end in the newline. The Wiki adds `' . '` to the text and reads it using the operator definitions also used to read the mode terms. See section 5.

```
* Term1
  Description
* Term2
  Description
```

Predicate description lists Especially for processing Wiki files, the Wiki notation allows for including the description of a predicate 'in-line', where the documentation is extracted from a loaded source file. For example:

```
The following predicates are considered Prolog's prime list processing
primitives:

    * [[member/2]]
    * [[append/3]]
```

Tables The Wiki provides only for limited support for tables. A table-row is started by a | sign and the cells are separated by the same character. The last cell must be ended with |. Multiple lines that parse into a table-row together form a table. Example:

```
| Algorithm    | Time (sec) |
| Depth first  | 1.0        |
| Breath first | 0.7        |
| A*           | 0.3        |
```

Section Headers Section headers are creates using one of the constructs below taken from TWiki. Section headers are normally not used in the source-code, but can be useful inside README and TODO files. See section 8.

```
---+ Section level 1
---++ Section level 2
---+++ Section level 3
---++++ Section level 4
```

Code (verbatim) Verbatim is embedded between lines containing only ==, as shown in the example below. The indentation of the == must match and the indentation of the verbatim text is reduced by the indentation of the == marks.

```
==
small(X) :-
        X < 5.
==
```

7.2 Text markup: fonts and links

Wiki text markup to realise fonts is mostly based on old plaintext conventions in Usenet and E-mail. We added Prolog specific conventions to this. For font changing code, the opening symbol must be followed immediately by a word and the closing one must immediately follow a word.

As code comments frequently contain symbols such as =, we —in contrast to normal Wiki conventions— do font font-switches only if a single word is surrounded by =, * or _. Longer sequences must be created using additional |:

```
PceEmacs can be set as default editor using
=|set_prolog_flag(editor, pce_emacs)|=
```

`*bold*`	Typset text in **bold**. Content must be a single word.
`*|bold|*`	Typset text in **bold**. Content can be long.
`_emphasize_`	Typset text as *emphasize*. Content must be a single word.
`_|emphasize|_`	Typset text as *emphasize*. Content can be long.
`=code=`	Typset text `fixed` font. Content must be a single word.
`=|code|=`	Typset text `fixed` font. Content can be long.
`name/arity`	Create a link to a predicate
`name//arity`	Create a link to a DCG rule
`name.pl`	If ⟨*name*⟩.pl is the name of an existing file in the same directory, create a link.
`<url>`	Create a hyperlink to URL.
`Word`	Capitalised words that appear as argument-name are written in *Italic*
$Word : CVS$	CVS expanded keyword. Typeset as the plain keyword value.

7.3 Images

Images can be included in the documentation by referencing an image file using one of the extensions `.gif`, `.png`, `.jpeg` or `.jpg`. By default this creates a link to the image file that must be visited to see the image. Inline images can be created by enclosing the filename in double square brackets. For example

```
The [[open.png]] icon is used open an existing file.
```

8 Directory indices

A directory index consists of the contents of the file `README` (or `README.TXT`), followed by a table holding all currently loaded source-files that appear below the given directory (i.e. traversal is *recursive*) and for each file a list of public predicates and their descriptive summary. Finally, if a file `TODO` or `TODO.TXT` exists, its content is added at the end of the directory index.

9 Documentation files

Sometimes it is desirable to document aspects of a package outside the source-files. For this reason the system creates a link to files using the extension `.txt`. The referenced file is processed as Wiki source. The two fragments below illustrate the relation between an `.pl` file and a `.txt` file.

```
%%      read_setup(+File, -Setup) is det.
%
%       Read application setup information from File.  The details
%       on setup are described in setup.txt.
```

```
---+ Application setup data

If a file =|.myapprc|= exists in the user's home directory the
application will process this data using setup.pl. ...
```

10 Running the documentation system

10.1 During development

To support the developer with an up-to-date version of the documentation of both the application under development and the system libraries the developer can start an HTTP documentation server using the command `doc_server`(*?Port*).

doc_collect(+*Bool*)
: Enable/disable collecting structured comments into the Prolog database.

doc_server(*?Port*)
: Start documentation server at *Port*. Same as `doc_server`(*Port, [allow(localhost), workers(1)]*).

doc_server(*?Port, +Options*)
: Start documentation server at *Port* using *Options*. Provided options are:

  root(+*Path*)
  : Defines the root of all locations served by the HTTP server. Default is `/`. *Path* must be an absolute URL location, starting with `/` and ending in `/`. Intented for public services behind a reverse proxy. See documentation of the HTTP package for details on using reverse proxies.

  edit(+*Bool*)
  : If `false`, do not allow editing, even if the connection comes from localhost. Intended together with the `root` option to make pldoc available from behind a reverse proxy. See the HTTP package for configuring a Prolog server behind an Apache reverse proxy.

  allow(+*HostOrIP*)
  : Allow connections from *HostOrIP*. If *Host* is an atom starting with a '.', suffix matching is preformed. I.e. `allow('.uva.nl')` grants access to all machines in this domain. IP addresses are specified using the library(socket) `ip/4` term. I.e. to allow access from the 10.0.0.X domain, specify `allow(ip(10,0,0,_))`.

  deny(+*HostOrIP*)
  : Deny access from the given location. Matching is equal to the `allow` option.

  Access is granted iff

  - Both *deny* and *allow* match
  - *allow* exists and matches
  - *allow* does not exist and *deny* does not match.

doc_browser
: Open the user's default browser on the running documentation server. Fails if no server is running.

doc_browser(+*Spec*)
: Open the user's default browser on the specified page. *Spec* is handled similar to `edit/1`, resolving anything that relates somehow to the given specification and ask the user to select.[2].

10.2 As a manual server

The library `doc/doc_library` defines `doc_load_library/0` to load the entire library.

doc_load_library
: Load all library files. This is intended to set up a local documentation server. A typical scenario, making the server available at port 4000 of the hosting machine from all locations in a domain is given below.

```
:- doc_server(4000,
              [ allow('.my.org')
              ]).
:- use_module(library('doc/doc_library')).
:- doc_load_library.
```

Example code can be found in `$PLBASE/doc/packages/examples/pldoc`.

10.3 Using the browser interface

The documentation system is normally accessed from a web-browser after starting the server using `doc_server/1`. This section briefly explains the user-interface provided from the browser.

10.3.1 Searching

The top-right of the screen provides a search-form. The search string typed is searched as a substring and case-insensitive. Multiple strings seperated by spaces search for the intersection. Searching for objects that do not contain a string is written as $-\langle string \rangle$. A search for adjacent strings is specified as `"`$\langle string \rangle$`"`. Here are some examples:

`load file`	Searches for all objects with the strings `load` and `file`.
`load -file`	Searches for objects with `load`, but *without* `file`.
`"load file"`	Searches for the string `load file`.

The two radio-buttons below the search box can be used to limit the search. All searches both the application and manuals. Searching for Summary also implies Name.

[2] BUG: This flexibility is not yet implemented

10.3.2 Views

The web-browser supports several views, which we briefly summarise here:

- *Directory*
  In directory-view mode, the contents of a directory holding Prolog source-files is shown file-by-file in a summary-table. In addition, the contents of the `README` and `TODO` files is given.

- *Source File*
  When showing a Prolog source-file it displays the module documentation from the `/** <module ... */` comment and the public predicates with their full documentation. Using the zoom button the user can select to view both public and documentated private predicates. Using the source button, the system shows the source with syntax highlighting as in PceEmacs and formatted structured comments.[3]

- *Predicate*
  When selecting a predicate link the system presents a page with the documentation of the predicate. The navigation bar allows switching to the Source File if the documentation comes from source or the containing section if the documentation comes from a manual.

- *Section*
  Section from the manual. The navigation bars allows viewing the enclosing section (*Up*).

10.3.3 Editing

If the browser is accessed from `localhost`, each object that is related to a known source-location has an edit icon at the right side. Clicking this calls `edit/1` on the object, calling the user's default editor in the file. To use the built-in PceEmacs editor, either set the Prolog flag `editor` to `pce_emacs` or run `?- emacs.` before clicking an edit button.

Prolog source-files have a *reload* button attached. Clicking this reloads the source file if it was modified and refreshes the page. This supports a comfortable edit-view loop to maintain the source-code documentation.

10.4 Including PlDoc in a LaTeX document

The LaTeX backend aims at producing quality paper documentation as well as integration of predicate description and Wiki files in LaTeX documents such as articles and technical reports. It is realised by the library `doc_latex.pl`.

The best practice for using the LaTeX backend is yet to be established. For now we anticipate processing a Wiki document saved in a .txt file using `doc_latex/3` to produce either a simple complete LaTeX document or a partial document that is included into the the main document using the LaTeX `\input` command. Typically, this is best established by writing a *Prolog Script* that generates the required LaTeX document and call this from a *Makefile*. We give a simple example from PlDoc, creating this section from the wiki-file `latex.txt` below.

```
:- use_module(library(doc_latex)).
:- [my_program].
```

[3] This mode is still incomplete. It would be nice to add line-numbers and links to documentation and definitions in the sources.

We generate latex.tex from `latex.txt` using this Makefile fragment:

```
.SUFFIXES: .txt .tex

.txt.tex:
        pl -f script.pl -g "doc_latex('$*.txt','$*.tex',[stand_alone(false)]),hal
```

10.4.1 Predicate reference for the LaTeX backend

High-level access is provided by `doc_latex/3`, while more low level access is provided by the remaining predicates. Generated LaTeX depends on the style file `pldoc.sty`, which is a plain copy of `pl.sty` from the SWI-Prolog manual sources. The installation installs `pldoc.sty` in the `pldoc` subdirectory of the Prolog manual.

doc_latex(+*Spec, +OutFile, +Options*) *[det]*

Process one or more objects, writing the LaTeX output to *OutFile*. *Spec* is one of:

Name **/** *Arity*
: Generate documentation for predicate

Name **//** *Arity*
: Generate documentation for DCG rule

File
: If File is a prolog file (as defined by `prolog_file_type/2`), process using `latex_for_file/3`, otherwise process using `latex_for_wiki_file/3`.

Typically *Spec* is either a list of filenames or a list of predicate indicators. Defined options are:

stand_alone(+*Bool*)
: If `true` (default), create a document that can be run through LaTeX. If `false`, produce a document to be included in another LaTeX document.

public_only(+*Bool*)
: If `true` (default), only emit documentation for exported predicates.

section_level(+*Level*)
: Outermost section level produced. Level is the name of a LaTeX section command. Default is `section`.

summary(+*File*)
: Write summary declarations to the named File.

latex_for_file(+*File, +Out, +Options*) *[det]*

Generate a LaTeX description of all commented predicates in *File*, writing the LaTeX text to the stream *Out*. Supports the options `stand_alone`, `public_only` and `section_level`. See `doc_latex/3` for a description of the options.

latex_for_wiki_file(+*File, +Out, +Options*) *[det]*

Write a LaTeX translation of a Wiki file to the steam *Out*. Supports the options `stand_alone`, `public_only` and `section_level`. See `doc_latex/3` for a description of the options.

latex_for_predicates(+*PI:list, +Out, +Options*) *[det]*

Generate LaTeX for a list of predicate indicators. This does **not** produce the \begin{description}...\end{description} environment, just a plain list of \predicate, etc. statements. The current implementation ignores *Options*.

11 Motivation of choices

Literal programming is an established field. The TeX source is one of the first and best known examples of this approach, where input files are a mixture of TeX and PASCAL source. External tools are used to untangle the common source and process one branch to produce the documentation, while the other is compiled to produce the program.

A program and its documentation consists of various different parts:

- The program text itself. This is the minimum that must be handed to the compiler to create an executable (module).
- Meta information about the program: author, modifications, license, etc.
- Documentation about the overall structure and purpose of the source.
- Description of the interface: public predicates, their types, modes and whether or not they are deterministic as wel as an informative text on each public predicate.
- Description of key private predicates necessary to understand how the public interface is realised.

Structured comments or directives

Comments can be added through Prolog directives, a route taken by Ciao Prolog with lpdoc [Hermenegildo, 2000] and Logtalk [Moura, 2003]. We feel structured comments are a better alternative for the following reasons:

- Prolog programmers are used to writing comments as Prolog comments.
- Using Prolog strings requires unnatural escape sequences for string quotes and long literal values tend to result in hard to find quote-mismatches. Python uses comments in long strings, fixing this problem using a three double quotes to open and close long comments.
- Comments should not look like code, as that makes it more difficult to find the actual code.

We are aware that the above problems can be dealt with using syntax-aware editors. Only a few editors are sufficiently powerful to support this correctly though and we do not expect the required advanced modes to be widely available. Using comments we do not need to force users into using a particular editor.

Wiki or HTML

JavaDoc uses HTML as markup inside the structured comments. Although HTML is more widely known than —for example— LaTeX or TeXinfo, we think the Wiki approach is sufficiently widely known today. Using text with minimal layout conventions taken largely from plaintext newsnet and E-mail, Wiki input is much easier to read in the source-file than HTML without syntax support from an editor.

Types and modes

Types and modes are not a formal part of the Prolog language. Nevertheless, their role goes beyond pure documentation. The test-system can use information about non-determinism to validate that deterministic calls are indeed deterministic. Type information can be used to analyse coverage from the test-suite, to generate runtime type verification or to perform static type-analysis. We have chosen to use a structured comment with formal syntax for the following reasons:

- As a comment, they stay together with the comment block of a predicate. We feel it is best to keep documentation as close as possible to the source.
- As we parse them separately, we can pick up argument names and create a readable syntax without introducing possibly conflicting operators.
- As a comment they do not introduce incompatibilities with other Prolog systems.

Few requirements

SWI-Prolog aims at platform independency. We want tools to rely as much as possible on Prolog itself. Therefore, the entire infrastructure is written in Prolog. Output as HTML is suitable for browsing and not very high quality printing on virtually all platforms. Output to LaTeX requires more infrastructure for processing and allows for producing high-quality PDF documents.

References

[Hermenegildo, 2000] 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.

[Jeffery *et al.*, 2000] David Jeffery, Fergus Henderson, and Zoltan Somogyi. Type classes in mercury. In *ACSC*, pages 128–135. IEEE Computer Society, 2000.

[Moura, 2003] Paulo Moura. *Logtalk - Design of an Object-Oriented Logic Programming Language*. PhD thesis, Department of Informatics, University of Beira Interior, Portugal, September 2003.

[Mycroft & O'Keefe, 1984] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.

Index