SWI-Prolog Source Documention

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

August 30, 2006

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 with 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.

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.

Only a few documentation and testing frameworks exist in the Prolog world. In our view they all fall short realising the basic needs in a lightweight and easy to addopt system. We have noticed in various projects as well as through the codee we receive in the cause of testing and debugging SWI-Prolog that the discipline to come with consistent style, well commented code and a test-suite is not very well established in the Prolog community. If we want to improve this practice, we should 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 10 we motivate our main choices.

2 Overview

The PlDoc infrastructure is based on *structured comments*, just like JavaDoc. 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 format. Each source file is documented in a single file. In addition, the documentation generator will generate an index file that can be used a 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 that seen most 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)

[1] See section 10.

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*⟩.

4 File and section 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 but 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 section comments is to use a section-type identifier between angled brackets, followed by the title of the section. Defined values for *Type* are given in the table below.

	`/**` <*Type*>*Title* `%%` <*Type*>*Title*
⟨*Type*⟩	Description
module	Comment of a module file
section	Section inside the module documentation
subsection	Sub-section inside a section
subsubsection	Sub-Sub-section inside a sub-section

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'
⟨*head*⟩	::=	⟨*functor*⟩'('⟨*argspec*⟩ ',' ⟨*argspec*⟩')'
	::=	⟨*functor*⟩
⟨*argspec*⟩	::=	[⟨*instantiation*⟩]⟨*argname*⟩[':'⟨*type*⟩]
⟨*instantiation*⟩	::=	'+' | '-' | '?' | ':' | '@' | '!'
⟨*type*⟩	::=	⟨*term*⟩

Instantiation patters 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 [Ciao assertion language, O'Keefe and Mycroft, Mercury].

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

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`
: 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`
: Error condition. First Prolog term is the error term. Remainder is the description.

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

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

`@`**version**
: Version of the module.

`@`**see**
: Related maaterial.

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

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

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

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

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

`@`**tbd**
: 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
```

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 for a table. Example:

```
| Author  | Jan Wielemaker |
| Copying | GPL            |
```

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 newsnet and E-mail. We added some 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 comment frequently contains symbols such as = we —in contrast to normal Wiki conventions— do font font-switches onlt 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.
`\bnfmetaurl`	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.

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, use add its contents at the end of the directory index.

9 Running the documentation system

9.1 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:

  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 exit 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].

9.2 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 setup 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.
```

10 Motivation of choices

Literal programming is an old field. The TEX source is one of the oldest and welknown examples of this approach were input files are a mixture of TEX and PASCAL source. External tools are used to untangle the common source, process one branche 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.

[2]BUG: This flexibility is not yet implemented

- 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 and Logtalk. 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.
- 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 purely 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 choosen 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 seperately, we can pick up predicate 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 to 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 virtuall every platform. Output to LaTeX requires more infrastructure for processing, but allows for producing high-quality PDF documents.

Index