A Documentation Generator for (C)LP Systems

Manuel Hermenegildo
Department of Computer Science
Technical University of Madrid (UPM)
herme@fi.upm.es

Abstract

We describe `lpdoc`, a tool which generates documentation manuals automatically from one or more logic program source files, written in ISO-Prolog, Ciao, and other (C)LP languages. It is particularly useful for documenting library modules, for which it automatically generates a rich description of the module interface. However, it can also be used quite successfully to document full applications. The documentation can be generated in many formats including `texinfo`, `dvi`, `ps`, `pdf`, `info`, `html/css`, Unix `nroff/man`, Windows help, etc., and can include bibliographic citations and images. `lpdoc` can also generate "man" pages (Unix man page format), nicely formatted plain ascii "readme" files, installation scripts useful when the manuals are included in software distributions, brief descriptions in `html/css` or `info` formats suitable for inclusion in on-line indices of manuals, and even complete WWW and `info` sites containing on-line catalogs of documents and software distributions. A fundamental advantage of using `lpdoc` is that it helps maintaining a true correspondence between the program and its documentation, and also identifying precisely to what version of the program a given printed manual corresponds. The quality of the documentation generated can be greatly enhanced by including within the program text *assertions* (declarations with types, modes, etc. ...) for the predicates in the program, and *machine-readable comments*. These assertions and comments are written using the Ciao system *assertion language*. A simple compatibility library allows conventional (C)LP systems to ignore these assertions and comments and treat normally programs documented in this way. The `lpdoc` manual, all other Ciao system manuals, and most of this paper, are generated by `lpdoc`.

1 Introduction

`lpdoc` is an *automatic program documentation generator* for (C)LP systems. Its main functionality is to generate a reference manual automatically from one or more source files of (constraint) logic programming systems. It has been developed as part of the Ciao Prolog [14] program development environment, but it can also be used to document source files of almost any

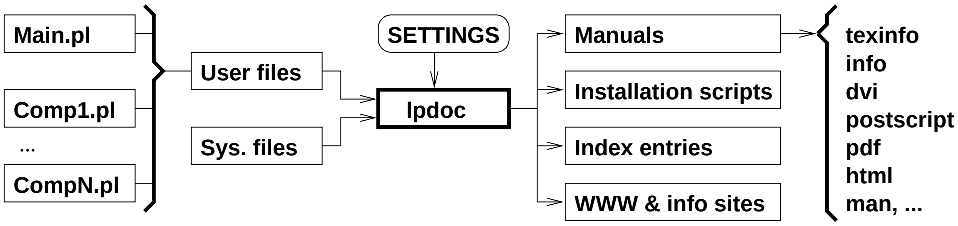

Figure 1: Overall operation

other (ISO-)Prolog-like [5] (C)LP system. `lpdoc` is particularly useful for documenting library modules, for which it automatically generates a rich description of the module interface. However, it can also be used quite successfully to document full applications.

The operation of `lpdoc` is illustrated in Figure 1. `lpdoc` combines the information from a number of user and system files (as specified in a user-provided configuration file –SETTINGS in Figure 1) and produces manuals in a number of formats (dvi, ps, pdf, info, html/css, ascii, Windows help, etc.) which can include bibliographic citations and images (if the target supports them). In addition to full manuals, `lpdoc` can also generate nicely formatted plain ASCII "readme" files, `man` pages (Unix manual page format), as well as brief descriptions in html or emacs info formats suitable for inclusion in an on-line *master index* of applications. Using these index entries, `lpdoc` can create and maintain fully automatically WWW and info sites containing pointers to the on-line versions of the documents it produces. Similarly, it can be used to generate software distribution sites. `lpdoc` also generates installation scripts for the manuals it produces, which simplify the process of creating a distribution of the corresponding software package. Finally, it is also possible to start a number of *viewers* directly from `lpdoc` in order to quickly browse the manuals produced.

The quality of the documentation generated can be greatly enhanced by including within the program text *assertions* (declarations with types, modes, and other properties) for the predicates in the program, and *machine-readable comments* (in the "literate programming" style [10, 3]). The assertions and comments included in the source file need to be written using the Ciao *assertion language* [11]. It also possible to use as part of the input to `lpdoc` files written in the GNU `texinfo` format, which is useful when gradually converting a manual from this popular format to `lpdoc`.

In order to make the discussion self-contained, an example of source code and the output produced by `lpdoc` is included at the end of the paper. However, since it is difficult to show significant output from the system in the space available, the reader is invited to look at actual manuals generated by `lpdoc` for reference while reading the paper. In particular, the `lpdoc` manual [9] and all other Ciao system manuals (and most of this paper, for that matter) are generated by `lpdoc`. The Ciao manuals and other `lpdoc`-generated manuals can be found on-line at `http://www.clip.dia.fi.upm.es/Software`, `http://www.clip.dia.fi.upm.es/Software/Ciao`, and `http://www.clip.-dia.fi.upm.es/Software/Beta` (registration as a Beta tester is needed for access to the latter). In fact, all these WWW sites are automatically generated and maintained by `lpdoc`.

2 Generating a manual

We now describe, from the user's point of view, the process of generating a manual (semi-)automatically from a set of source files, installing them in a public area, and accessing them on line.

The process starts by creating a directory (e.g., `doc`) in which the documentation will be built.[1] This directory is usually placed in the top directory of the distribution of the application or library to be documented. Typically, almost all files in this directory will be automatically generated by `lpdoc`, which also takes care of cleaning up this directory of intermediate files before distribution of the software, leaving only the manuals in the selected formats. This directory will also contain the necessary information for installation of the manuals during the installation of the software package. This directory should contain the manually maintained configuration file of Figure 1, normally named *SETTINGS*, which is written in Prolog syntax, possibly using Ciao syntactic enhancements (in particular, the functional notation is often useful in this context).

A manual can be generated either from a single source file or from a set of source files. In the latter case, one of these files should be chosen to be the *main file*, and the others will be the *component files*. The main file is the one that will provide the title, author, date, summary, etc. to the entire document. In principle, any set of source files can be documented, even if they contain no assertions or comments. However, the presence of these will greatly improve the documentation (see Section 3).

The name of main file is specified in the SETTINGS file by defining a fact of a predicate `main`. Facts of a (possibly empty) predicate `components` define the component files which will generate the different chapters of the manual. Facts of a predicate `filepaths` are used to define all the directories where the previously mentioned files can be found. Similarly, facts of the predicate `systempaths` are used to list all the *system* directories where system files used by the files being documented can be found. This is needed because on startup `lpdoc` has *no default search paths for files* defined, not even those defined by default in the `Prolog` system under which it was compiled (typically Ciao). This has the important consequence that it allows documenting `Prolog` systems other than that under which `lpdoc` was com-

[1] Actually, documentation for a single file can be generated fully automatically from the Ciao emacs mode, which then also takes care of creating the documentation directory in a temporary area.

piled. The effect of putting a path in `systempaths` instead of in `filepaths` is that the modules and files in those paths are documented as *system modules* (this is useful when documenting an application to distinguish its parts from those which are in the system libraries).

These are the only settings which are strictly needed in order to generate a manual. However, many aspects of the generated manuals can be controlled through additional configuration parameters. For example, it is possible to control what is included in the different files and how: whether to include bug information or not, comments associated to version changes and/or to patches, author info, detailed explanation of predicate argument modes, starting page number, etc. It is also possible to define the set of formats (`dvi`, `ps`, `pdf`, `ascii`, `html`, `info`, `manl`, ...) in which the documentation should be generated by default (however, a manual in any of the supported formats can be generated on demand by typing "`lpdoc` *format*"). In particular, selecting `htmlindex` and/or `infoindex` requests the generation of (parts of) a master index to be placed in an installation directory and which provide pointers to the documents generated.

A predicate `indices` determines a list of indices to be included at the end of the document. These can include indices for defined predicates, modules, properties, types, concepts, files, etc. The contents of these indices are afterwards used for several purposes in on-line documents. In particular, `lpdoc` includes an emacs library for automatically locating any part of the manual related to the symbol (predicate, flag, property, type, etc.) under the cursor ("help for symbol under cursor") and also performing automatic completion of partially typed names of predicates, types, etc. This is very useful when typing the name of a library predicate: it is possible to complete the name and also locate in one step the corresponding page in the on-line manual generated by `lpdoc`.

It is possible to define a predicate `bibfile` containing paths of *.bib files*, i.e., files containing *bibliographic entries* in `bibtex` format. If citations are used in the text (using the `@cite` command) these will be the files in which the citations will be searched for. All the references in all component files will appear together in a *References* appendix at the end of the manual (the `-norefs` option prevents generation of the 'References' appendix). It is also possible to select different levels of verbosity during processing, from pretty silent –more or less only a couple of messages per file–, to quite verbose, documenting the files visited and the predicates being documented on the fly. The latter is obviously quite useful for debugging.

Once the manual has been generated in the desired formats, `lpdoc` can also install them in a different area, specified by a predicate `docdir` in the `SETTINGS` file. As mentioned before, `lpdoc` can generate directly brief descriptions in html or `emacs` info formats suitable for inclusion in an on-line index of applications. In particular, if the `htmlindex` and/or `infoindex` options are selected, then `lpdoc` will create the installation directory, place the documentation in the desired formats in this directory, and produce and place in the same directory suitable `index.html` and/or `dir` files. These files will contain some basic info on the manual (extracted from the summary and title, respectively) and include pointers to the relevant documents which have been installed. The appearance of the actual indices created (e.g., `index.html`) can be controlled via templates and style sheets, specified in the configuration file. Several manuals, coming from different `doc` directories, can be installed in the same `docdir` directory. In this case, the descriptions of and pointers to the different manuals will be automatically combined (appearing in alphabetic order) in the `index.html` and/or `dir` indices, and a *contents area* will appear at the beginning of the *html index page*. In the same way, facilities are provided for de-installation of manuals from the `docdir` area.

3 Enhancing the documentation being generated

`lpdoc` will generate quite useful information from standard program files: e.g., exported predicates with their arity, characteristics of these predicates –dynamic, multifile, ...–, other modules used, required libraries, and, if available, types and other properties, etc. However, the quality of the documentation generated can be greatly enhanced by including within the program text *assertions*, and *machine-readable comments*.

Assertions are declarations which are included in the source and provide information regarding certain characteristics of the program. Typical assertions include type declarations, modes, general properties (such as *does not fail*), etc. For our purposes, we can consider standard compiler directives (such as `dynamic/1`, `op/3`, `meta_predicate/1`...), also as assertions. When documenting a module, `lpdoc` will use the assertions associated with the module interface to construct a textual description of this interface. In principle, only the exported predicates are documented, although any predicate can be included in the documentation by explicitly requesting it (by using a particular `comment/2` declaration –see below). Judicious use of these assertions allows at the same time documenting the program code, documenting the external use of the module, and greatly improving the debugging process. The latter is possible because the assertions provide the compiler with information on the intended meaning or behavior of the program (i.e., the specification) which can be checked at compile-time (by a preprocessor/static analyzer) and/or at run-time (via checks inserted by the same preprocessor) –see [7] for details.

Machine-readable comments are also declarations included in the source program but which contain additional information intended to be read by humans (this is where the connection with the *literate programming* style of Knuth [10, 3] is closest). These declarations are ignored by the compiler in the same way as classical comments. Thus, they can be used to document the program source in place of (or in combination with) the normal comments typically inserted in the code by programmers. However, because they are more structured and they are machine-readable, they can also be used to improve the automatic generation of printed or on-line documentation. Typical such comments include module title, author(s), bugs, changelog, etc. Judicious use of these comments allows enhancing at the same time the documentation of the program text and the manuals generated.

`lpdoc` requires these assertions and comments to be written using the `Ciao` system *assertion language* [11].[2] Comments have the general form:
`:- comment(`*CommentType*`,`*CommentData*`).`
where generally the first argument states the type of comment and the second one the comment itself, written in a particular markup language which is very similar to texinfo and LaTeX (see Section 7). Examples of comments are:

```
:- comment(title,"Complex numbers library").
:- comment(summary,"Provides and ADT for complex numbers.").
:- comment(ctimes(X,Y,Z),"@var{Z} is @var{Y} times @var{X}.").
```

An example of an assertion is:

```
:- pred qsort(X,Y) : list(X) => sorted(Y)
                   # "@var{Y} is a sorted permutation of @var{X}.".
```

which states that in the calls to predicate `qsort/2` the first argument should be a list and, upon exit, the second argument should be sorted. There is also a textual *assertion comment*, written using the same markup language as in `comment/2`. The properties `list/1` and `sorted/1` used in the assertion might be declared as such with the following assertions (we are also including the actual definitions for illustration purposes):

```
:- prop sorted(X) # "@var{X} is sorted.".
sorted([]).
sorted([_]).
sorted([X,Y|R]) :- X < Y, sorted([Y|R]).

:- regtype list(X) # "@var{X} is a list.".
list([]).
list([_|T]) :- list(T).
```

(list is actually a particular case of property: a *regular type*). Space limitations do not allow a description of the assertion language. See the appendices for more examples and [11, 8] for details.

4 Overall structure of the generated documents

If the manual is generated from a single main file (i.e., `components` is empty), then the document generated will be a flat document containing no chapters.

[2] A simple compatibility library is available in order to make it possible to compile programs documented using assertions and comments in traditional (constraint) logic programming systems which lack native support for them. Using this library, such assertions and comments are simply ignored by the compiler.

If the manual is generated from a main file and one or more components, then the main file will be used to generate the cover and introduction, while each of the component files will be used to generate a separate chapter. The contents of each chapter will reflect the contents of the corresponding component source file.

If a `.pl` file does not define the predicates `main/0` or `main/1`, it is assumed to be a *library* and information on the interface (e.g., the predicates exported by the file, the name of the module and usage if it is a module, etc. –the API), is produced by default. If, on the contrary, the file defines the predicates `main/0` or `main/1`, it is assumed to be an *application* and no description of the interface is generated. Instead, usage information is produced. Any combination of libraries and/or main files of applications can be used arbitrarily as components or main files of a `lpdoc` manual. Several interesting combinations are documented in the `lpdoc` manual [9].

In any case, a cover is generated with the title, authors, summary, version, etc. of the whole manual, which are those of the main file. Then comes the table of contents, whose level of detail can also be controlled via options. This is followed by the sections or chapters corresponding to the file or files being documented. Finally, the manual ends with the selected indices, list of references, etc.

5 Structure of chapters

The structure of the individual chapters depends also on whether they are applications or libraries. In the case of libraries, the structure is as below. Note that inclusion of many of the following items can be turned on or off and can be configured in several ways through options. An example of a source file and the chapter generated for it (under a particular set of options) are listed in appendices A and B, for illustration while reading the following items.

- Chapter title, from a `title` comment, such as the line:
  `:- comment(title,"The classical quick-sort").`
  in the example. If the file is the main file, the title text (a documentation string) will also be used in the cover page and also as the description of the manual in on-line indices. If no such comment exists, then a suitable one is generated from the module or file name. Also, a `subtitle` comment is allowed.

- Authors, which are obtained from `author` comments, such as:
  `:- comment(author,"Alan Robinson").`
  There can be more than one of these declarations per module (normally, one per author). These are followed by copyright info (from `copyright` comments) and version info (from changelog comments).

If the file is part of a bigger package, then both the file version (i.e., when last changed) and the overall system versions are documented.

- Chapter introduction, taken from a `summary` comment or from a `module` comment, if no summary is available (see also the example).
- A usage and interface section, which is typically generated without any need for comment declarations, and includes:
  - Module usage info, stating whether it is a module, a user file, a package [1], etc., and how it is to be loaded. These automatically generated loading instructions can be replaced by more specific ones by means of a `usage` comment.
  - List of exported predicates. These are classified by kind: normal predicates, multifile predicates, regular types, properties, declarations, etc.
  - The list of other modules used. These are separated into *User*, *System* and, optionally, *Engine* libraries[3] (this division is controlled by the paths in `SETTINGS`). It is possible to optionally prevent the information on *System* and/or *Engine* libraries used from being included in the manual. Note that this information is useful because it allows the user of a library to see which other libraries it will load, and thus the impact that it will have on the size of the executable.
- A section with overall information on the library, taken from the `module` comment, if available (and if this comment was not already used before).
- A section documenting any new declarations defined by the library (Ciao specific).
- A section documenting the predicates (including regular types and properties) exported by the library (e.g., `qsort/2`, `list/1`, and `sorted/1` in the example). In principle, all exported predicates are documented. However, it is possible to prevent documentation on a predicate from appearing in the manual by using a `hide` comment.
- A section documenting the multifile predicates defined by the library.
- Possibly a section documenting some internal predicates (or regular types or properties) defined by the library. In principle internal (local) predicates are not documented, but documentation of an internal predicate can be forced by using a `doinclude` comment. This is the case for `partition/4` in the example

[3] In Ciao, engine libraries contain builtins that are always present in any executable, independently of whether they are imported or not from the program.

- Optionally, a section with known bugs, i.e., those present in `bug` comments (see the example).
- Optionally, a section with a list of changes, those present in `version` comments (see the example). It is possible to list only comments associated with major version changes an leave out minor changes ("patches"). This allows writing version comments which are internal, i.e., not meant to appear in the manual. Code is provided for maintaining version numbers automatically with `emacs`, or they can also be maintained with other tools such as standard version control systems.
- Reexported predicates, i.e., predicates which are exported by a module `m1` but defined in another module `m2` which is used by `m1`, are normally not documented in the original module, but instead a simple reference is included to the module in which they are defined. This can be changed, so that the documentation is included in the referring module, by using a `comment/2` declaration with `doinclude` in the first argument. This is often useful when documenting a library made of several components: for a simple user's manual, it is often sufficient to list in the `lpdoc SETTINGS` file the principal module, which is the one which users will do a `use_module/1` of, in the manual. This module typically exports or reexports all the predicates which define the library's user interface.

If the chapter is documenting an application, then no module interface information is included in the documentation, but it still contains title, authors, version, summary, usage information, body, bugs, changelog, etc.

6 Documentation on individual predicates, properties, etc.

We now describe how individual predicates, declarations, properties, etc. are documented. This is done in essentially the same way, independently of whether they appear in the export list or they are internal predicates. The documentation is obviously more detailed if more information is available on the predicate in the form of assertions and comments.

If the program does not contain any declarations for the predicate, a line is output documenting that this is a predicate of the given name and arity and a simple comment is included saying that there is no further documentation available. Note that this means for example that the predicate will appear in the index, and also that its name will be available for command completion.

If the predicate is declared to be a property or regular type, then this fact is included in the documentation. If there is no textual comment available for it, then its actual definition is included in the documentation (see `list/1` in the example). Otherwise, the comment is used (as with `sorted/1` in the example).

If an overall comment (a `comment/2` declaration) is available for a predicate, it is used as a general explanation (see the general comment for `qsort/2` in the example). If any assertions are present, they are documented in mostly textual form. In particular, if `pred` declarations are present, each of them is considered a possible *usage* and is documented as such (e.g., the two `pred` declarations for `qsort/2` in the example). If a comment appears in the `pred` declaration, it is associated with the usage.

The syntactic sugar which can be used with the assertions can be either kept as is or expanded when documentation is generated. In the example, having chosen the corresponding option, the *modes* (which are "property macros" in the Ciao assertion language) used in `partition/4` have been spelled out in the documentation. Note that the parametric type `list/2` used (e.g., in `list(X,num)`) is assumed to be imported by default.

An interesting point is that if a textual comment is available in the definition of a property or regular type (such as for `sorted` in the example) then this text is used when the property itself is used elsewhere in an assertion. An example is the use of `sorted` in the two usages for `qsort/2`. This also occurs if the property is imported from another module: the comment is read from that module (actually, from the module's `.asr` interface file) [12].[4]

7 Documentation strings

As shown in previous examples, the character strings which can be used in machine readable comments (`comment/2` declarations) and assertions can include certain *formatting commands* ("markup"). The syntax of all the formatting commands is: `@`*command* (followed by either a space or {}), or `@`*command*{*body*} where *command* is the command name and *body* is the (possibly empty) command body. Also, a command may have several bodies, as in: `@`*command*{*body1*}{*body2*}.

In order to make it possible to produce documentation in a wide variety of formats, the command set is kept small. The names of the commands are intended to be reminiscent of the commands used in the LaTeX text formatting system, except that "`@`" is used instead of "\". Note that "\" would need to be escaped in ISO-Prolog strings, which would make the source less readable.[5] Given that space restrictions do not allow a full description of the command set, we provide a general description by categories.

There are a number of *indexing commands* which are used to mark certain words or sentences in the text as concepts, names of predicates, libraries,

[4] This occurs in the example with `list/2`, which is in the lists library.

[5] `@` is familiar to `texinfo` users and, in any case, many ideas in LaTeX were taken from scribe, where the escape character was indeed "`@`"!

files, etc. and which then get indexed and cross-referenced in hypertext formats. There are also *referencing commands* which are used to introduce *bibliographic citations* and *references* to sections, urls, email addresses, etc. A set of *formatting commands* are provided which allow typesetting certain words or sentences in a special fonts/faces, build itemized lists, introduce sections, include verbatim examples, cartouches, etc. There are also special commands for generating *accented* and *special* characters. A number of *inclusion commands* allow inserting code or strings of text as part of the documentation. The latter may reside in external files or in the file being documented. The former must be part of the module being documented. There are also commands for inserting and scaling images.

8 Other issues

8.1 Separating the documentation from the source file

Sometimes one would not like to include long introductory comments in the module itself but would rather have them in a different file. This can be done quite simply by using the `@include` command. For example, the following declaration:

```
:- comment(module,"@include{Intro.lpdoc}").
```

will include the contents of the file `Intro.lpdc` as the module description.

Alternatively, sometimes one may want to generate the documentation from a completely different file. Assuming that the original module is `m1.pl`, this can be done by calling the module containing the documentation `m1_doc.pl`. This `m1_doc.pl` file is the one that will be included the `lpdoc` `SETTINGS` file, instead of `m1.pl`. `lpdoc` recognizes and treats such `_doc` files specially so that the name without the `_doc` part is used in the different parts of the documentation, in the same way as if the documentation were placed in file `m1`.

8.2 Generating auxiliary files (e.g., READMEs)

Using `lpdoc` it is often possible to use a common source for documentation text which should appear in several places. For example, assume a file `INSTALL.lpdoc` contains text (with `lpdoc` formatting commands) describing an application. This text can be included in a section of the main file documentation as follows:

```
:- comment(module,"...
   @section{Installation instructions}
   @include{INSTALL.lpdoc}
   ...").
```

At the same time, this text can be used to generate a nicely formatted `INSTALL` file in ascii, which can perhaps be included in the top level of the source directory of the application. To this end, an `INSTALL.pl` file as follows can be constructed:

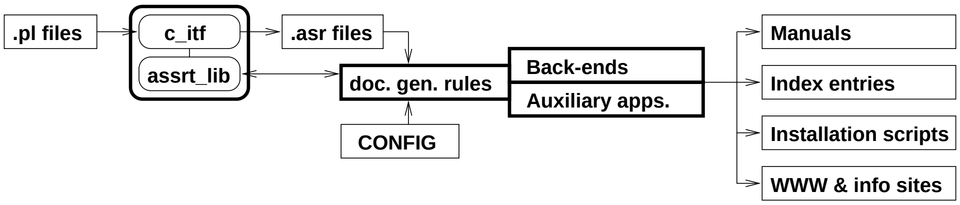

Figure 2: Internal architecture

```
:- use_package([assertions]).
:- comment(title,"Installation instructions").
:- comment(module,"@include{INSTALL.lpdoc}").
main. %% forces file to be documented as an aplication
```

Then, the ascii `INSTALL` file can be generated by simply running `lpdoc ascii` in a directory with a `SETTINGS` file where `MAIN` is set to `INSTALL.pl`.

9 System architecture and implementation

Space limitations only allow us to sketch the architecture of the system.[6] `lpdoc` is implemented in (Ciao-)Prolog and compiled into a standalone Ciao executable. Since the source used by `lpdoc` is not just simple comments but the actual code of the modules (e.g., the assertions, the module declarations, exports, imports, dynamic declarations, syntax extensions, mode definitions, etc., and even the source code) `lpdoc` requires a full reader. This is specially true for the full Ciao system source language, which is designed to be very extensible [1]. Also, because the design objective was to be able to document very large systems in an efficient way, processing of the source files, including module interface information, declarations, comments, assertions, etc. has been made highly incremental.

The objectives are achieved in a straightforward way by using the Ciao assertion processing library (see Figure 2), itself an instance of the `c_itf` low-level generic modular processing library [2], which, for each documented file, and transitively for other files used by the one being documented, reads all the information, normalizes the assertions, and saves them in `.asr` and `.itf` cache files. This process is only repeated on a needed basis when a source file is modified. The generation of documentation files is also partly incremental, in that a documentation cache file (currently in GNU `texinfo` format)[7] is kept for each Prolog file being documented and which only changes as needed by any changes in the source files. Thus, a form of "separate documentation" (in the same sense as "separate compilation") is achieved.

---

[6]Details can be found in the comments within the source files of the system, which, when printed out using `lpdoc` constitute the system's internals manual.

[7]See "The GNU Texinfo Documentation System" manual for more info on this format, widely used in the GNU project and on Linux and other Unix systems.

Given the information on the modules, `lpdoc` uses a number of documentation generation rules (part of which are defined in a configuration file) to implement the documentation actions outlined in previous sections. Documentation is in general first generated in an internal format (basically, the language of Section 7), and then converted by a number of backends (in Prolog) and/or auxiliary (publicly available) applications (TeX, dvi2ps, etc.) into manuals in the different formats, index entries, installation scripts, etc. It is quite easy to add new backends. One of the most complicated issues has been to generate consistent documentation and support as many common features as possible across many different formats (for example, supporting citations using BiBTeX files was tricky because few of the underlying formats were capable of this).

10 Related work

We are not aware of other automatic documentation systems that have all the capabilities of `lpdoc`. There are some systems which allow interspersing TeX and Prolog in a source file in the style of Knuth's original formulation of literate programming.[8] While these systems are quite useful, we believe that `lpdoc` goes beyond them in that a significant part of the documentation is generated essentially automatically by modules of the compiler, and that the assertion language used is shared with other program development tools, which makes them quite useful beyond just documentation. ICON and Perl have some (limited) facilities for merging documentation and programs. Perhaps the closest tool to `lpdoc` is the `javadoc` documentation system for Java [6]. As `lpdoc`, `javadoc` uses information which is typically read and/or derived by the compiler (types, class structure, etc.), allows including textual comments with (HTML a tag-based) markup, and can be extended via *doclets*. Because of the tight integration with the language, `javadoc` cannot be used well for Prolog programs (in the same way as `lpdoc` would certainly not be as effective as `javadoc` for Java programs). Also, we feel that the markup language and, specially, the assertion language and the way properties can be used in documentation, as well as the number of output formats, are richer in `lpdoc`. Also, `lpdoc` is not limited to documenting APIs, i.e., it can also show source code.

11 Conclusions

Since the first "production" versions of the `lpdoc` system became available [9], we have applied it in a number of scenarios. We have used it to document all the components of the Ciao Prolog development environment, libraries for

[8] See `ftp://ftp.dante.de/tex-archive/macros/latex/contrib/other/gene/pl.tar.gz` for a good example.

SICStus and CHIP, standalone Prolog applications, and even applications not written in Prolog. It has certainly proven very useful for documenting library modules. However, we have also found it quite useful for generating "internals" and also user manuals of applications. Because the system can not only generate manuals in many formats, but also maintain documentation and software distribution sites, we have found ourselves using it for documenting and building distribution sites for a number of applications which, as mentioned above, were not even written in Prolog.

We have found that with a bit of practice one can write assertions and comments that at the same time document the program code, document the external use of the library, and greatly improve the debugging and maintenance cycles. One of the fundamental practical advantages observed when using `lpdoc` to document programs is that it is much easier to maintain a true correspondence between the program and its documentation, and to identify precisely to what version of the program a given printed manual corresponds. Furthermore, another fundamental advantage comes from the fact that the assertions are designed to be checkable in part, either statically or dynamically [8, 7], so that the documentation also achieves a certain degree of certification. While in the Ciao system writing assertions is optional (in contrast to, e.g., Mercury [13]), the fact that they will generate a good part of the manual encourages programmers to write them, and this in turn helps developing programs faster, because more errors are detected early on.

`lpdoc` is publicly available.[9] The system is currently going further development in several directions, such as, for example, reducing the need for auxiliary applications (so that it is portable to more platforms) or improving the emacs interface. With a simple compatibility library it is relatively easy to make traditional (constraint) logic programming systems (in which new declarations can be defined) accept programs adorned with Ciao-style assertions and comments, so that they are ignored during compilation but `lpdoc` (and the Ciao preprocessor!) can be used on them. As mentioned above, we have done this for SICStus and CHIP. It should not be too difficult to modify the front end for other type/assertion languages, such as those used in Mercury [13] and HAL [4] (this is under study at least in the case of HAL).

12 Acknowledgements

The authors would like to thank the members of the CLIP group and the anonymous referees for their constructive comments. The development of `lpdoc` has been funded in part by projects DiSCiPl (ESPRIT LTR 22532 and CICYT TIC 97-1640-CE) and ELLA (CICYT TIC 96-1012-C02-01) and ELLA (CICYT TIC 96-1012-C02-01). The design of the `lpdoc` system has benefitted from suggestions made by CLIP group members and users of Ciao Prolog which are too many to mention (acknowledgements are given in

[9] See `http://www.clip.dia.fi.upm.es/Software` .

the reference manual and source files). This document has benefitted from detailed comments from Daniel Cabeza, Per Cederberg, and the anonymous referees.

References

[1] D. Cabeza and M. Hermenegildo. A New Module System for Prolog. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*. N.M. State U., December 1999.

[2] D. Cabeza and M. Hermenegildo. The Ciao Modular Compiler and Its Generic Program Processing Library. In *ICLP'99 WS on Parallelism and Implementation of (C)LP Systems*. N.M. State U., December 1999.

[3] D. Cordes and M. Brown. The Literate Programming Paradigm. *IEEE Computer Magazine*, June 1991.

[4] B. Demoen, M. Garcia de la Banda, W. Harvey, K. Marriott, and P. Stuckey. Herbrand Constraint Solving in HAL. In *Int. Conf. on Logic Programming*. MIT Press, Cambridge, MA, U.S.A., November 1999.

[5] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard*. Springer-Verlag, 1996.

[6] Lisa Friendly. The Design of Distributed Hyperlink Program Documentation. In *Int'l. WS on Hypermedia Design*, Workshops in Computing. Springer, June 1996. Available from `http://java.sun.com/docs/javadoc-paper.html`.

[7] M. Hermenegildo, F. Bueno, G. Puebla, and P. López. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, Cambridge, MA, November 1999. MIT Press.

[8] M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[9] M. Hermenegildo and The CLIP Group. An Automatic Documentation Generator for (C)LP – Reference Manual. The Ciao System Documentation Series–TR CLIP5/97.3, Facultad de Informática, UPM, August 1997.

[10] D. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984.

[11] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *ILPS'97 WS on Tools and Environments for (C)LP*, October 1997. `ftp://clip.dia.fi.upm.es/pub/papers-/assert_lang_tr_discipldeliv.ps.gz`.

[12] G. Puebla and M. Hermenegildo. Some Issues in Analysis and Specialization of Modular Ciao-Prolog Programs. In *ICLP'99 WS on Optimization and Implementation of Declarative Languages*. N.M. State U., Las Cruces, NM, November 1999.

[13] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.

[14] The CLIP Group. CIAO System Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, Facultad de Informática, UPM, August 1997.

A An example: source

```
:- module(sort,[qsort/2,list/1,sorted/1],[assertions,regtypes,isomodes]).
:- use_module(library(lists),[append/3]).

:- comment(title, "The classical quick-sort").
:- comment(module,"This library provides a naive implementation of
   quick-sort and some associated types and properties.").

:- comment(qsort(X,Y),"@var{Y} is a sorted permutation of @var{X}.").
:- pred qsort(X,Y) : list(X) => sorted(Y)
                   # "This is the normal use.".
:- pred qsort(X,Y) : (list(X), sorted(Y))
                   # "Checking that @var{Y} is a sorted permutation of @var{X}.".
qsort([],[]).
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).

:- pred partition(+list(num),+num,-list(num),-list(num)).
:- comment(doinclude,partition/4).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E < C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E >= C, !,
        partition(R,C,Left,Right1).

:- prop sorted(X) # "@var{X} is sorted.".

sorted([]).
sorted([_]).
sorted([X,Y|R]) :- X < Y, sorted([Y|R]).

:- regtype list/1.

list([]).
list([_|T]) :- list(T).
```

```
:- comment(bug, "Code uses @pred{append/3}, which is inefficient.").

:- comment(version_maintenance,on).
:- comment(version(0*1+1,1999/10/11,03:19*00+'CEST'), "Already made
   the first change...  (Manuel Hermenegildo)").
:- comment(version(0*1+0,1999/10/11,03:18*29+'CEST'), "File created.
   (Manuel Hermenegildo)").
```

B The classical quick-sort

Version: 0.1#1 (1999/10/11, 3:19:0 CEST)

This library provides a naive implementation of quick-sort and some associated types and properties.

B.1 Usage and interface (sort)

- **Library usage:**
  `:- use_module(library(sort)).`
- **Exports:**
  - *Predicates:*
    `qsort/2`.
  - *Properties:*
    `sorted/1`.
  - *Regular Types:*
    `list/1`.
- **Other modules used:**
  - *System library modules:*
    `lists`.
  - *Internal (engine) modules:*
    `arithmetic`, `atomic_basic`, `attributes`, `basic_props`, `basiccontrol`, `concurrency`, `data_facts`, `exceptions`, `io_aux`, `io_basic`, `prolog_flags`, `streams_basic`, `system_info`, `term_basic`, `term_compare`, `term_typing`.

B.2 Documentation on exports (sort)

qsort/2: PREDICATE

`qsort(X,Y)`

`Y` is a sorted permutation of `X`.

Usage 1: `qsort(X,Y)`

- *Description:* This is the normal use.
- *Should hold at call time:*
  `list(X)` (`list/1`)

- *Should hold upon exit:*

  `Y` is sorted. (`sorted/1`)

Usage 2: `qsort(X,Y)`

- *Description:* Checking that `Y` is a sorted permutation of `X`.
- *Should hold at call time:*

  `list(X)` (`list/1`)

  `Y` is sorted. (`sorted/1`)

list/1: REGTYPE

A regular type, defined as follows:

```
list([]).
list([_1|T]) :-
        list(T).
```

sorted/1: PROPERTY

Usage: `sorted(X)`

- *Description:* `X` is sorted.

B.3 Documentation on internals (`sort`)

partition/4: PREDICATE

Usage:

- *Should hold at call time:*

  `Arg1` is a list of `nums`. (`regtype/2`)

  `Arg2` is a number. (`regtype/2`)

  `Arg3` is a free variable. (`var/1`)

  `Arg4` is a free variable. (`var/1`)

- *Should hold upon exit:*

  `Arg3` is is a list of `nums`. (`regtype/2`)

  `Arg4` is is a list of `nums`. (`regtype/2`)

B.4 Known bugs and planned improvements (`sort`)

- Code uses `append/3`, which is inefficient.

B.5 Version/Change Log (`sort`)

- **Version 0.1#1 (1999/10/11, 3:19:0 CEST)**

  Already made the first change... (Manuel Hermenegildo)

- **Version 0.1 (1999/10/11, 3:18:29 CEST)**

  File created. (Manuel Hermenegildo)