An Overview of the SWI-Prolog Programming Environment

Jan Wielemaker

Social Science Informatics (SWI),
University of Amsterdam,
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands,
jan@swi.psy.uva.nl

Abstract. The Prolog programmer's needs have always been the focus for guiding the development of the SWI-Prolog system. This article accompanies an invited talk about how the SWI-Prolog environment helps the Prolog programmer solve common problems. It describes the central parts of the graphical development environment as well as the command line tools which we see as vital to the success of the system. We hope this comprehensive overview of particularly useful features will both inspire other Prolog developers, and help SWI-Prolog users to make more productive use of the system.

1 Introduction

SWI-Prolog has become a popular Free Software implementation of the Prolog language. Distributed freely through the internet, it is difficult to get a clear picture about its users, how these users use the system and which aspects of the system have contributed most to its popularity. Part of the users claim the programmer's environment described in this article is an important factor.

The majority of the SWI-Prolog users are students using it for their assignments. The community of developers, however, expend effort on large portable Prolog applications where scalability, (user-) interfaces, networking are often important characteristics. Compared to the students, who are mostly short-term novice users, we find many expert software developers in the research and development community.

The material described in this paper is the result of about 18 years experience as a Prolog programmer and developer of the SWI-Prolog system. Many of the described tools are features not unique to SWI-Prolog and can be found in other Prolog implementations or other programming language environments. Experiments are yet to be performed to evaluate the usefulness of features and therefore the opinions presented are strictly based on our own experiences, observations of users, and E-mail reactions.

After describing the SWI-Prolog user community in Sect. 2 we describe some problems Prolog programmers frequently encounter in Sect. 3. In Sect. 4 we describe the command line tools, and in Sect. 5 the graphical tools written in SWI-Prolog's XPCE GUI toolkit [10].

2 User profiles

Students having to complete assignments for a Prolog course have very different needs from professionals developing large systems. They want easy access to common tasks as closely as possible to the conventions they are used to. Scalability of supporting tools is not an important issue as the programs do not require many resources. Visualization of terms and program state can concentrate their contribution to *explanation* and disregard, for example, the issue that most graphical representations scale poorly. The *SWI-Prolog-Editor*[1] shell for MS-Windows by Gerhard Röhner makes SWI-Prolog much more natural to a student who is first of all familar with MS-Windows.

SWI-Prolog comes from the Unix and Emacs tradition and targets the professional programmer who uses it frequently to develop large Prolog-based applications. As many users in this category have their existing habits, and a preferred set of tools to support these, SWI-Prolog avoids presenting a single comprehensive IDE (*Integrated Development Environment*), but instead provides individual components that can be combined and customised at will.

3 Problems

Many problems that apply to programming in Prolog also relate the programming in other languages. Some, however, are Prolog specific. Prolog environments can normally be used interactively and changed dynamically.

3.1 Problem areas

- *Managing sources*
  Besides the normal problems such as locating functions and files, Prolog requires a tool that manages consistency between the sources and running executable during the interactive test-edit cycle. Section 4.1 and Sect. 5.1 describe the SWI-Prolog support to manage sources.
- *Entering and reusing queries*
  Interaction through the Prolog top level is vital for managing the program and testing individual predicates. Command line editing, command completion, do what I mean (DWIM) correction, history, and storing the values of top level variables reduces typing and speed up the development cycle.
- *Program completeness and consistency*
  SWI-Prolog has no tradition in rigid static analysis. It does provide a quick completeness test as described in Sect. 4.6 which runs automatically during the test-edit cycle. A cross-referencer is integrated into the built-in editor (Sect. 5.1) and provides immediate feedback to the programmer about common mistakes while editing a program.

[1] http://www.bildung.hessen.de/abereich/inform/skii/material/swing/indexe.htm

- *Error context*
  If an error occurs, it is extremely important to provide as much context as possible. The SWI-Prolog exception handling differs slightly from the ISO standard to improve such support. See Sect. 4.10.
- *Failure/wrong answer*
  A very common and time consuming problem are programs producing the wrong (unexpected) answer without producing an error. Although research has been carried out to attribute failure and wrong answers to specific procedures [3, 9], none of this is realised in SWI-Prolog.
- *Determinism*
  Although experience and discipline help, controlling determinism in Prolog programs to get all intended solutions quickly is a very common problem. The source-level debugger (Sect. 5.3) displays choicepoints and provides immediate graphical feedback on the effects of the cut, greatly simplifying this task and improving understanding for novices.
- *Performance bottlenecks*
  Being a high level language, the relation between Prolog code and required resources to execute it is not trivial. Profiling tools cannot fix poor overall design, but do provide invaluable insight to programmer. See Sect. 5.4.
- *Porting programs from other systems*
  Porting Prolog programs has been simplified since more Prolog systems have adopted part I of the ISO standard. Different extensions and libraries cause many of the remaining problems. Compiler warnings and static analysis form the most important tools to locate the problem areas quickly. A good debugger providing context on errors together with support for the test-edit cycle improve productivity.

4 Command line Tools

4.1 Supporting the edit cycle

Prolog systems offer the possibility to interactively edit and reload a program even while the program is running. There are two simple but very frequent tasks involved in the edit-reload cycle: finding the proper source, and reloading the modified source files. SWI-Prolog supports these tasks with two predicates:

make
: SWI-Prolog maintains a database of all loaded files with the file last-modified time stamp when it was loaded and —for the sake of modules— the context module(s) from which the file was loaded. The **make/0** predicate checks whether the modification time of any of the loaded files has changed and reload these file into the proper module context. This predicate has proven to be very useful.

edit(*+Specifier***)**
: Find all entities with the given *specifier*. If there are multiple entities related to different source-files ask the user for the desired one and call the

user-defined editor on the given location. *All entities* implies (loaded) files, predicates and modules. Both locating named entities and what is required to call the editor on a specific file and line can be hooked to accomodate extensions (e.g. XPCE classes) and different editors. Furthermore, SWI-Prolog maintains file and line-number information for modules and clauses. Below is an example:

```
?- edit(rdf_tree).
Please select item to edit:

  1 class(rdf_tree)             'rdf_tree.pl':27
  2 module(rdf_tree)            'rules.pl':460

Your choice? 2
```

SWI-Prolog's *completion* and *DWIM* described in Sect. 4.4 and Sect. 4.3 improve the usefulness of these primitives.

4.2 Autoloading and auto import

Programmers tend to be better at remembering the names of library predicates than the exact library they belong to. Similar, programmers of large modular applications often have a set of personal *favourites* and application specific *goodies*. SWI-Prolog supports this style of programming with two mechanisms, both of which require a module system. The SWI-Prolog module system is very close to the Quintus and SICStus Prolog module systems [2].

Auto import tries to import undefined predicates from the module's *import module*. The module `system` contains all built-in predicates, `user` all global predicates and all other modules import from `user` as illustrated in Fig. 1. This setup allows programmers to define or import commonly used predicates into `user` and have them available without further actions from the interactive top level and all modules.

Library *auto loading* avoids the need for explicit **use_module/[1,2]** declarations. Whenever the system encounters an unknown predicate it examines the library index. If the predicate appears in the index the library is loaded using **use_module/2**, only importing the missing predicate.

The combination of auto import, auto loading and a structuring module system has proven to support both *sloppy* programming for rapid prototyping and the use of more maintainable explicit module relations. The predicate **list_autoload/0** as described in Sect. 4.6 supports a smooth transition.

4.3 DWIM: Do What I Mean

DWIM (*Do What I Mean*) is implemented at the top level to quickly fix mistakes and allow for underspecified queries. It corrects the following errors:

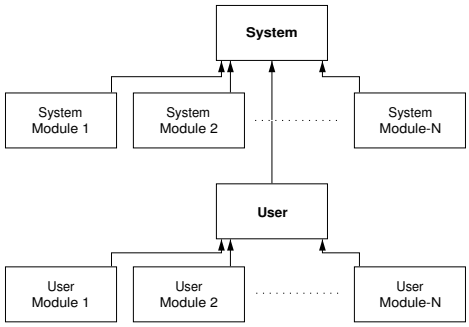

Fig. 1. Modules and their *auto-import* relations

- *Simple spelling errors*
  DWIM checks for missing, extra and transposed characters that result from typing errors.
- *Word breaks and order*
  DWIM checks for multi-word identifiers using different conventions (e.g. *fileExists* vs. *file_exists*) as well as different order (e.g. *exists_file* vs. *file_exists*)
- *Arity mismatch*
  Of course such errors cannot be corrected.
- *Wrong module*
  DWIM adds a module specification to predicate references that lack one or replaces a wrong module specification.

DWIM is used in three areas. Queries typed at the top level are checked and if there is a unique correction the system prompts whether to execute the corrected rather than the typed query. Especially adding the module specifier improves interaction from the top level when using modules. If there is no unique correction the system reports the missing predicates and all close candidates. Queries of the development system such as **edit/1** and **spy/1** provide alternative matches one-by-one. Spy/1 and **trace/1** act on the specified predicate in any module if the module is omitted. Finally, if a predicate existence error reaches the top level the DWIM system is activated to report likely candidates.

4.4 Command line editing

Developers spend a lot of time entering commands for the development system and (test-)queries for (parts of) their application under development. SWI-Prolog provides the following features to support this:

- *Using (GNU-)readline*
  Emacs-style editing is supported in the Unix version based on the GNU *readline* library and in Windows using our own code. This facilitates quick and natural command reuse and editing. In addition, *completion* is extended

with completion on alphanumerical atoms which allow for fast typing of long predicate identifiers and atom arguments as well as inspect the possible alternative (using Alt-?). The completion algorithm uses the builtin completion of files if no atom matches, which ensures that quoted atoms representing a file path is completed as expected.

- *Command line history*
  SWI-Prolog provides a history facility that resembles the Unix csh and bash shells. Especially viewing the list of executed commands is a valuable feature.
- *Top level bindings*
  When working at the Prolog top level, bindings returned by previous queries are normally lost while they are often required for further analysis of the current Prolog state or to test further queries. For this reason SWI-Prolog stores the resulting bindings from top level queries, provided they are not too large (default $\leq$ 1000 tokens) in the database under the name of the used variable. Top level query expansion replaces terms of the form $\$Var$ ($ is a prefix operator) into the last recorded binding for this variable. New bindings do to backtracking or new queries overwrite the old value.
  This feature is particularly useful to query the state of data stored in related dynamic predicates and deal with handles provided by external stores. Here is a typical example using XPCE that avoids typing or copy/paste of the object reference.

```
?- new(X, picture).

X = @12946012
?- send($X, open).
```

4.5 Compiler

An important aspect of the SWI-Prolog compiler is its performance. Loading the 21 Mb sources of WordNet [7] requires 6.6 seconds from the source and 1.4 seconds from precompiled virtual machine code (Multi-threaded SWI-Prolog 5.2.9, SuSE Linux on dual AMD 1600+ using one thread). Fast compilation is very important during the interactive development of large applications.

SWI-Prolog supports the commonly found set of compiler warnings: syntax errors, singleton variables, predicate redefinition, system predicate redefinition and discontiguous predicates. Messages are processed by the hookable **print_message/2** predicate and where possible associated with a file and line number. The graphics system contains a tool that exploits the message hooks to create a window with error messages and warnings that can be selected to open the associated source location.

4.6 Quick consistency check

The library *check* provides quick tests on the completeness of the loaded program. The predicate **list_undefined/0** searches the internal database for predicate structures that are undefined (i.e. have no clauses and are not defined as

dynamic or multifile). Such structures are created by the compiler for a call to a predicate that is not yet defined. In addition the system provides a primitive that returns the predicates referenced from a clause by examining the compiled code. Figure 2 provides partial output running **list_undefined/0** on the *chat 80* [8] program:

```
1 ?- [library(chat)].
%   ...
%  library('chat/chat') compiled into chat 0.18 sec, 493,688 bytes
% library(chat) compiled into chat 0.18 sec, 494,756 bytes

Yes
2 ?- list_undefined.
% Scanning references for 9 possibly undefined predicates
Warning: The predicates below are not defined. If these are defined
Warning: at runtime using assert/1, use :- dynamic Name/Arity.
Warning:
Warning: chat:ditrans/12, which is referenced by
Warning:         5-th clause of chat:verb_kind/6
```

Fig. 2. Using **list_undefined/0** on chat 80 wrapped into the module `chat`. To save space only the first of the 9 reported warnings is included. The processing requires 0.25 sec. on a 733 Mhz PIII.

The **list_autoload/0** predicate lists undefined predicates that can be autoloaded from one of the libraries. It is illustrated in Fig. 3.

```
3 ?- list_autoload.
% Into module chat (library('chat.pl'))
%     display/1          from library(edinburgh)
%     last/2             from library(lists)
%     time/1             from library(statistics)
% Into module user
%     prolog_ide/1       from library(swi_ide)
```

Fig. 3. Using **list_autoload/0** on chat 80

4.7 Help and explain facility

The help facility uses outdated but still effective technology. The LaTeX maintained source is translated to plain text. A generated Prolog index file provides character ranges for predicate descriptions and sections in the manual. Each

predicate has, besides the full documentation, a ± 40 character summary description used for *apropos* search as well as to provide a summary string in the editor as illustrated in Fig. 4.

The *explain* facility examines the database to gather all information known about an identifier (atom). Information displayed includes predicates with that name and references to the atoms, compound terms and predicates with the given name. Here is an example:

```
explain(setof).
"setof" is an atom
        Referenced from 1-th clause of chat:decomp/3
system:setof/3 is a built-in meta predicate imported from module
               $bags defined in
        /staff/jan/lib/pl-5.2.9/boot/bags.pl:59
        Summary: ``Find all unique solutions to a goal''
        Referenced from 6-th clause of chat:satisfy/1
        Referenced from 7-th clause of chat:satisfy/1
        Referenced from 1-th clause of chat:seto/3
```

The graphical front end is described in Sect. 5.5.

4.8 File commands

Almost too trivial to name, but the predicates **ls/0**, **cd/1** and **pwd/0** are used very frequently.

4.9 Debugging from the terminal

SWI-Prolog comes with two tracers, a traditional 4-port debugger [1] to be used from the terminal and a graphical source level debugger which is described in Sect. 5.3. Less frequently seen features of the trace are:

- *Single keystroke operation*
  If the terminal supports it, commands are entered without waiting for RETURN.
- *List choicepoints*
  The tracer can provide a list of active choicepoints, similar to the goal stack, to facilitate choicepoint tuning and debugging.
- *The 'up' command*
  The 'up' command is like the traditional 'skip' command, but skips to the exit or failure of the *parent* goal rather than the current goal. It is very useful to stop tracing the details of failure driven control structures.
- *Search*
  The system can search for a specific port and goal that unifies with an entered term. The command `/f foo(_, bar)` will go into interactive debugging if **foo/2** where the second argument unifies with `bar` reaches the *fail* (f) port.

In addition to interactive debugging two types of non-interactive debugging are provided. Using **trace**(*Predicate, Ports*), the system prints all passes to the indicated ports of *Predicate*.

The library *debug* is a lightweight infrastructure to handle printing debugging messages (logging) and assertions. The library exploits goal-expansion to avoid runtime overhead when compiled with optimisation turned on. Debug messages are associated to a *Topic*, an arbitrary Prolog term used to group debug messages. Normally the *Topic* is an atom denoting some function or module of the application. Using Prolog unification of the active topics and the topic registered with the message provides opportunity for creativity.

debug(*+Topic, +Format, +Arguments*)
: Prints a message through the system's **print_message/2** message dispatching mechanism if debugging is enabled on *Topic*.

debug/nodebug(*+Topic*)
: Enable/disable messages for which *Topic* unifies. Note that topics are arbitrary Prolog terms, so **debug**(_) enables all debugging messages.

list_debug_topics
: List all registered topics and their current enable/disable setting. All known topics are collected during compilation using goal-expansion.

assume(*:Goal*)
: Assume that *Goal* can be proven. Trap the debugger if *Goal* fails. This facility is derived from the C-language assert() macro defined in `<assert.h>`, renamed for obvious reasons. More formal assertion languages are described in [6, 5].

4.10 Exception context

On exception handling, the ISO standard dictates 'undo' back to the state at entry of a **catch/3** before unifying the *ball* with the *catcher*. SWI-Prolog however uses a different technique. It walks the stack searching for a matching catcher without undoing changes. If it finds a matching **catch/3** call or when reaching a call from foreign code that indicates it is prepared to handle exceptions it performs the required 'undo' and executes the handler. The advantage is that if there is no handler for the exception the entire program state is still intact. The debugger is started immediately and can be used to examine the full context of the exception.[2]

5 Graphical Tools

5.1 Editor

PceEmacs is an Emacs clone written in XPCE/Prolog. It has two features that make it of special interest. It can be programmed in Prolog and therefore has

[2] These issues have been discussed on the comp.lang.prolog newsgroup, April 15-18 2002, subject "ISO catch/throw question".

transparent access to the environment of the application being developed, and the editor's buffer can be opened as a Prolog I/O stream. Based on these features, the level of support for Prolog development is far beyond what can be achieved in a stand-alone editor. Whenever the user pauses for two seconds the system performs a full cross-reference of the editing buffer, categorising and colouring predicates, goals and general Prolog terms. Predicates are categorised as *exported*, *called* and *not called*. Goals are categorised as *builtin*, *imported*, *auto-imported*, *locally defined*, *dynamic*, *(direct-)recursive* and *undefined*. Goals have a menu that allows jumps to the source, documentation (builtin), and listing of clauses (dynamic). Singleton variables are highlighted. If the cursor appears inside a variable all other occurrences of this variable in the clause are underlined. Figure 4 shows a typical screenshot.

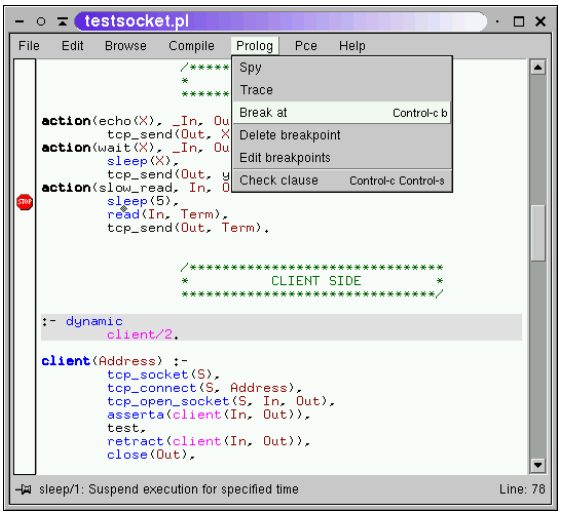

Fig. 4. PceEmacs in action

5.2 Prolog Navigator

The Prolog *Navigator* provides a hierarchical overview of a project directory and its Prolog files. Prolog files are categorised as one of *loaded* or *not loaded* and are expanded to the predicates defined in them. The defined predicates are categorised as one of *exported*, *normal*, *fact* and *unreferenced*. Expanding predicates expands the *call tree*. The Navigator menus provide loading and editing files and predicates as well as the setting of trace- and spy-points. See Fig. 5.

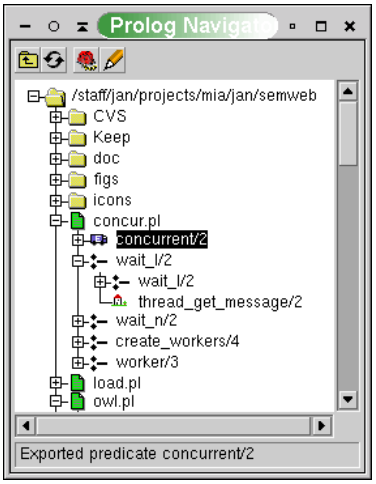

Fig. 5. The Prolog Navigator

5.3 Source-level Debugger

The SWI-Prolog debugger calls a hook (prolog_trace_interception/4) before reverting to the built-in command line debugger. The built in **prolog_frame_attribute/3** provides the infrastructure to analyse the Prolog stacks, providing information on the goal-stack, variable bindings and choice-points. These hooks are used to realise more advanced debuggers such as the source-level debugger described in this section. The source-level debugger provides three views (Fig. 6):

- *The source*
  An embedded *PceEmacs* (see Sect. 5.1) running in *read-only* mode shows the current location, indicating the current port using colour and icons. *PceEmacs* also allows the setting of *breakpoints* at a specific call in specific clause. Breakpoints provide finer and more intuitive control where to start the debugger than traditional spy-points. Breakpoints are realised by replacing a virtual machine instruction with a *break* instruction which traps the debugger, finds the instruction it replaces in a table and executes this instruction.
- *Variables*
  The debugger displays a list of variables appearing in the current frame with their name and current binding in the top-left window. The representation of values can be changed using the familiar **portray/1** hook. Double-clicking a variable-value opens a separate window showing the variable binding. This window uses indentation to make the structure of the term more explicit and has a menu to control the layout.
- *The stack*
  The top-right window shows the stack as well as the recent active choi-

cepoints. Any node can be selected to examine the context of that node. The stack view allows one to quickly examine choicepoints left after a goal succeeded. Besides showing the location of the choicepoint itself, the 'up' command can be used to examine the parent frame context of a choicepoint.

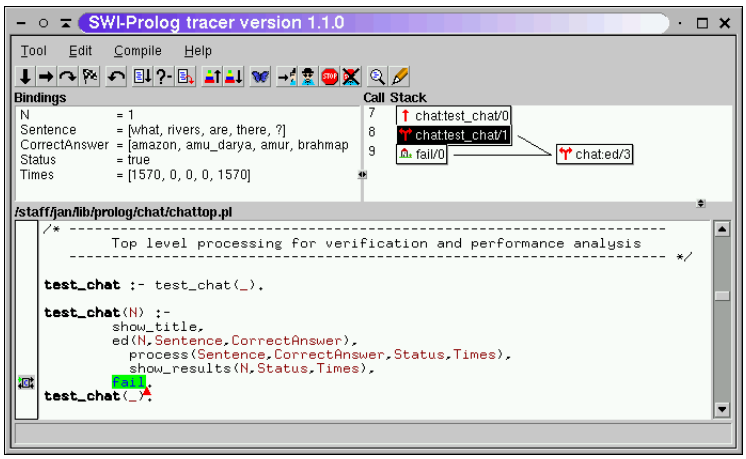

Fig. 6. The Source-level Debugger

5.4 Execution Profiler

The *Execution Profiler* builds a call-tree at runtime and ticks the number of calls and redos to each node in this call-tree. The time spent in each node is established using stochastic sampling.[3] Recording the call-tree is complicated by three factors.

- *Last call optimisation*
  Due to last call optimisation exit ports are missing from the execution model. This problem is solved by storing the call-tree node associated with a goal in the environment stack, providing the exit with a reference to the node exited. Recording an *exit* can now exit all nodes until it reaches the referenced node.
- *Redo*
  Having a reference from each environment frame to the call-tree node also greatly simplifies finding the proper location in the call-tree on a *redo*.
- *Recursion*
  To avoid the uncontrolled expanding of the call-tree the system must record *recursive* calls. The problem lies in the definition of *recursion*. The most naïve

[3] Using SIGPROF on Unix and using a separate thread and a multi-media timer in MS-Windows.

definition is that recursion happens if there is a parent node running the same predicate. In this view meta predicates will often appear as unwanted 'recursive predicates' as will predicates called in a totally different context. The system provides **noprofile/1** to indicate some predicates do not create a new node and their time is included with their parent node. Examples are **call/1**, **catch/3** and **call_cleanup/2**. Calls are now regarded recursive if the parent node runs the same predicate (direct recursion) or somewhere in the parent nodes of the call-tree we can find a node running the same predicate with the same immediate parent.

Prolog primitives are provided to extract all information from the recorded call-tree. A graphical Prolog profiling tool presents the information interactively similar to the GNU gprof [4] tool (see Fig. 7).

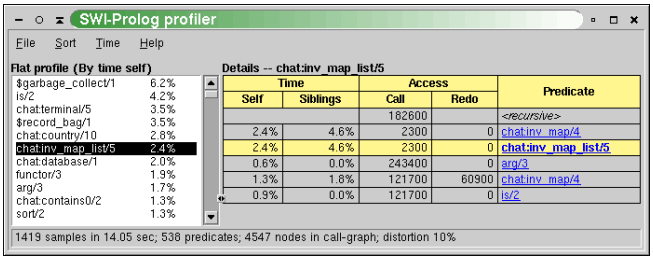

Fig. 7. The Profiler

5.5 Help System

The GUI front end to the help functionality described in Sect. 4.7 adds hyperlinks and hierarchical context to the command line version as illustrated in Fig. 8.

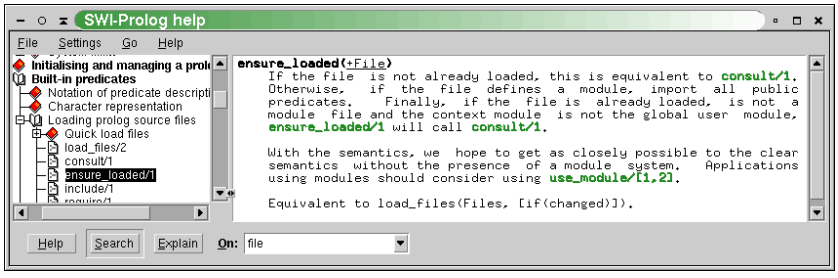

Fig. 8. Graphical front end to the help system

6 Conclusions

In this paper we have described commonly encountered tasks which Prolog programmers spend much of their time on, which tools can help solving them as well as an overview of the programming environment tools provided by SWI-Prolog. Few of these tools are unique to SWI-Prolog or very advanced. The popularity of the environment can possibly be explained by being complete, open, portable, scalable and free.

Acknowledgements

XPCE/SWI-Prolog is a Free Software project which, by its nature, profits heavily from user feedback and participation. We would like to thank Steve Moyle and Anjo Anjewierden for their comments on draft versions of this paper.

References

1. Lawrence Byrd. Understanding the control flow of Prolog programs. In S.-A. Tarnlund, editor, *Proceedings of the Logic Programming Workshop*, pages 127–138, 1980.
2. M. Carlsson, J. Widén, J. Andersson, S. Anderson, K. Boortz, H. Nilson, and T. Sjöland. *SICStus Prolog (v3) Users's Manual*. SICS, PO Box 1263, S-164 28 Kista, Sweden, 1995.
3. Mireille Ducassé. Analysis of failing Prolog executions. In *Workshop on Logic Programming Environments*, pages 2–9, 1991.
4. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
5. M. Hermenegildo, G. Puebla, and F. Bueno. Using global analysis, partial specifications, and an extensible assertion language for program validation and debugging. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 161–192. Springer-Verlag, 1999.
6. Marija Kulas. Debugging Prolog using annotations. In Mireille Ducassé, Anthony Kusalik, and German Puebla, editors, *Electronic Notes in Theoretical Computer Science*, volume 30. Elsevier, 2000.
7. G. Miller. WordNet: A lexical database for English. *Comm. ACM*, 38(11), November 1995.
8. Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. Number 10 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford, California, 1987. Distributed by Chicago University Press.
9. E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
10. Jan Wielemaker and Anjo Anjewierden. An architecture for making object-oriented systems available from Prolog. In Alexandre Tessier, editor, *Computer Science, abstract*, 2002. http://lanl.arxiv.org/abs/cs.SE/0207053.