HyperPro
An integrated documentation environment for CLP

AbdelAli Ed-Dbali
LIFO - Universit d'Orlans
BP 6759 - 45067 Orlans Cedex - France
AbdelAli.ED-DBALI@lifo.unv-orleans.fr

Pierre Deransart
INRIA Rocquencourt - BP 105 - 78153 Le Chesnay - France
Pierre.Deransart@inria.fr

Mariza A. S. Bigonha, José de Siqueira, Roberto da S. Bigonha
DCC - UFMG - Brsil
{mariza,jose,bigonha}@dcc.ufmg.br

October 17, 2001

Abstract

The purpose of this paper is to present some functionalities of the HyperPro System. HyperPro is a hypertext tool which allows to develop *Constraint Logic Programming* (CLP) together with their documentation. The text editing is not new and is based on the free software Thot. A HyperPro program is a Thot document written in a report style. The tool is designed for CLP but it can be adapted to other programming paradigms as well. Thot offers navigation and editing facilities and synchronised static document views. HyperPro has new functionnalities such as document exportations, dynamic views (projections), indexes and version management. Projection is a mechanism for extracting and exporting relevant pieces of code program or of document according to specific criteria. Indexes are necessary to find the references and occurrences of a relation in a document, i.e., where its predicate definition is found and where a relation is used in other programs or document versions and to translate hyper-texts links into paper references. It still lack importation facilities.

Keywords :CLP, literate programming, program version, structured editor, hypertext, documentation, specification.

1 Introduction

HyperPro is a documentation and development tool for Constraint Logic Programming (CLP) systems. HyperPro helps to document CLP programs giving its users the possibility to edit, in a homogeneous and integrated environment, a single program or different versions of a program, comments about them, information for formal verification and debugging purposes, as well as the possibility to execute, debug and test the program or program versions as well. All the tests executed on a CLP program and its history of development can therefore be integrated and consistently documented within an unique environment.

HyperPro has tow basic: version management and documentation. A particular aspect of CLP is that the written programs are short and they are tested on multiple versions which differ only by small changes. In fact, for example, the way constraints are ordered may drastically affect the efficiency and the reachability of a solution. It is thus essential that the user can keep trace of the many versions of a program (there is potentially an exponential number of versions). In this approach of program development, documentation is at least as useful as the code of the program itself. It can be compared with elaboration of formal specification where documentation (or requirements) in natural language plays a role at least as significant as the code.

The HyperPro system, developed by the authors, make use of the Thot editor and its API. Thot is a structured editing tool with hypertext editing facilities. It was developed at INRIA and it is based on the logical aspect of the document ([7], [8], [6], [5]). Thot uses a meta-model that allows the description of several models of documents, with their various presentations, through dedicated languages. Thot is in fact very comparable with XML technology ([12]). Its language $\mathcal{S}$ defining the structure of a document is similar to XML's DTD (*Document Type Definition*). Its languages $\mathcal{P}$ and $\mathcal{T}$ (respectively presentation and translation languages) could be compared with the XSL (*eXtensible Stylesheet Language*) and XSLT (for *Transformation*). We chose Thot because it provides an API which allows the easy creation of an editor such as HyperPro and offers the possibility to integrate new applications to existing ones. It has also a remarkable feature: the possibility to open several views of the same document in synchronized windows.

This paper describes some of the HyperPro functionalities which are: static synchronized views of the document; management of multiple versions of a program within the same document; testing of different program versions; test of these versions using external CLP systems; dynamic views (produced by projections requests); indexes; exports of the document under different formats for processing with other systems. This paper completes the presentation of HyperPro in [1] and presents the main novelties: support for execution, exportations, indexes and projections. Projection is a mechanism for extracting and exporting relevant pieces of code according to specific criteria. Indexes are of three types: the traditional index of words, the index of the relations cross references and the index of the program versions. The two latter relate to the code parts and highlight the "two-dimensional" hypertext structure of a document.

Thanks to the modularity of the implementation, with few changes, HyperPro can be adapted to other languages such as the language C (cf. [4]).

In section 2, we survey basic HyperPro features whose ideas where initialy presented in [1]: static views, versions, program execution and exportations.

Section 3 is dedicated to the indexes, section 4 to dynamic views (or projections) and we conclude this paper by a comparison between HyperPro and other literate programming systems.

2 HyperPro Functionalities

This section introduces the most important functionalities of HyperPro, which are the static views of different parts of the document, program versions, their testing and document export in various formats.

The HyperPro document looks like a report. It must have a title, a sequence of at least one section, a table of contents and a words index. Optionally, it can contain the date, the authors' names and their affiliations, keywords, bibliographical references, annexes and other indexes. HyperPro defines, also, a special style for logic programming, so a paragraph, may be also a *relation definition.* At least one relation definition must be present in a HyperPro document. A relation definition contains a *relation title* and a list of at least one *predicate definition.* The relation title is defined by a predicate indicator, that is the predicate name and its arity, or just a name, since a packet of goals or directives can be seen as a special case of relations. The relation title contains also a reference to a predicate definition, followed optionally by a list of references to predicate definitions that define a unique program or different *versions* of the program.

2.1 Static views

Thot allows the user to define views of his document, such that, chosen portions of it are presented separately in a view. Views are synchronized in such a way to facilitate selecting, moving and editing consistently and easily in a particular view as much as in the main view. HyperPro provides several different views of the same document: the main view that shows the whole document, the table of contents, the informal comments on relations, the assertions on relations, the relations code which are a collection of packets of clauses corresponding to different versions of the same relation. The main view and the table of contents are displayed automatically, when a document is open. The other views are shown on demand by the user through the appropriate menu bar.

By clicking somewhere in the main view, like in one of the other opened views, all the views are synchronized simultaneously to present the corresponding part of the referred text.

2.2 Program Versions

The possibility of being able to manage various versions of a program during the development and documentation is a significant feature offered by HyperPro. This possibility finds all its interest in the case of the Constraint Logic Programming. Indeed, it is interesting to be able to integrate in the same document several versions which, for example, differ only by one clause or are completely different because they are based on different algorithms.

Every relation is defined by at least one packet of clauses. Among these packets we distinguish a particular one which is the current packet defining the relation. It is called the *current predicate definition* (*c.p.d.*). The c.p.d. is pointed by a hypertext link placed in the relation title. This link is called the *current predicate reference* (*[c.p.r.]*). The [c.p.r.] can be changed to point to any other predicate definition the user wishes, within the same relation definition. The [c.p.r.] are obligatory since they define the current or the default predicate definition for each relation in the document.

A program is defined when it has appropriate references to the relations which compose it. Once all the relations and their c.p.d. for a program are given, the user defines the program by putting a hypertext reference to the relation which defines the predication in every predication in the body of all relation's c.p.d. These references are called *definition references* ([def.]). The user does not have to put [def.] to predications in the body of the c.p.d. which are direct recursive calls.

Figure 1(a) shows some relations defined for a program, with the [c.p.r.] presented in the relation titles composing the program and the [def.] in the predications at the body of the c.p.d. The document has two sections, section `1.1`, which defines relations `a/1`, `b/2`, and `c/1`, and section `1.2`, which defines relation `a/1`. Only relation `a/1` in section `1.1` has two predicate definitions defined for it. The [c.p.r.] for relation `a/1` points to its first predicate definition. The [c.p.r.] are represented by full lines in the figure. The dashed lines represent the definition references, which are presented in the text of the document as [def.]. As it is shown, they point to the relation titles they refer to. Note that there are no [def.] for predications recursively referring to the relation they are defining.

The [def.], together with the version naming utility, allows the user to effectively manage and document different versions of the program in the same document which can have different relations with the same name and arity. Conflicts between relations are therefore solved and managed explicitly and manually by the user with [def.]. Different versions of the program defined in a document are distinguished from each other by their names and are managed with the corresponding naming reference. The naming reference allows the user and the utilities based on it to follow and retrieve all predicate clauses of every relation in a program as defined by the chain of [def.] of every relation linked in it.

To name a version, once the [def.] for a program are inserted for the predications in the body of the c.p.d. as explained above, the user selects the utility *name a version* in the *Tools* menu. A window opens and the user enters a name for the version. The user then chooses the first relation composing his program and a named reference is put then automatically in the version bar, after the title of every relation composing the version, pointing to the respective c.p.d. of each relation

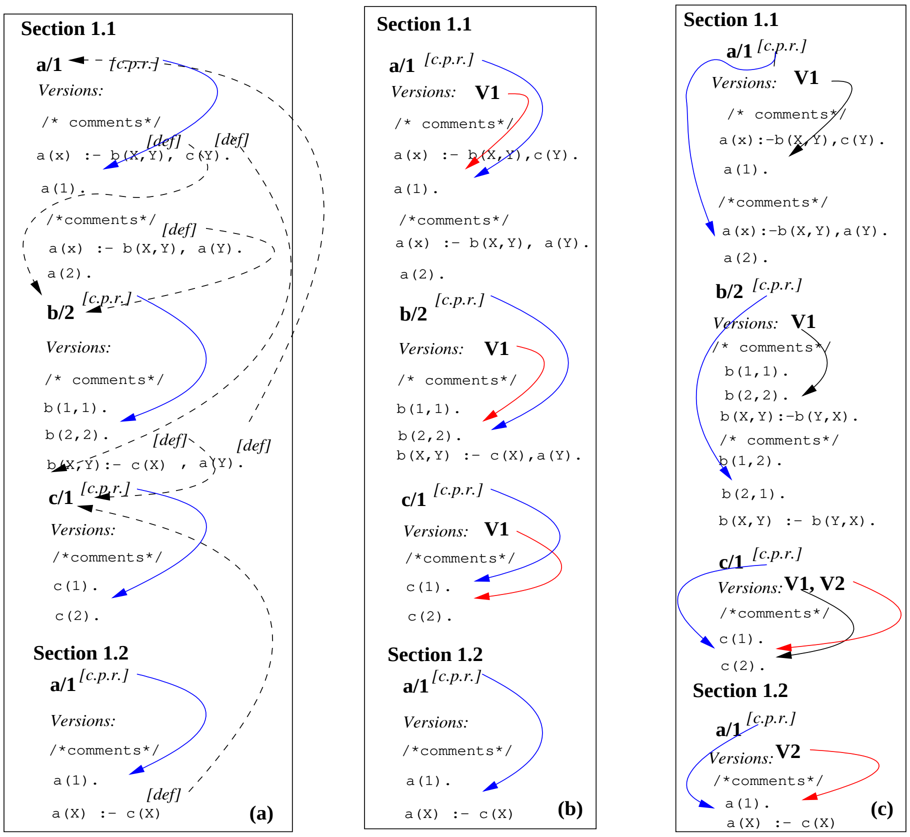

Figure 1: (a) Some Relations, [c.p.r.] and Version References (b) After Naming Version `V1` (c) After naming Version `V2`

in the program.

Figure 1(c) shows two different versions named V1 and V2 defined in a document which has, among others, two different relations with the same name and the same arity. This figure shows the final state after defining both versions. However, the user first named version V1, as presented in Figure 1(b) by choosing the utility *"name a version"* in the *tools* menu, giving the version the name V1, and finally choosing relation `a/1`, which is the first relation in his version. The pointers for V1 point to the c.p.d., i.e. to the same predicate definitions pointed to by the [c.p.r.].

Afterwards, the user entered a new section, numbered 1.2, where he defined relation `a/1`. In its predicate definition, the second clause for `a/1` refers to relation `c/1`, previously defined in section `1.1`. Therefore, he puts a [def.] after the predication `c(X)`, which points to relation `c/1` in section `1.1`. Then the user proceeds to name this new version, choosing for it the name of V2. After choosing relation `a/1` in section `1.2`, the naming reference V2 appears in the version bar after its title, which points to the c.p.d. pointed to by the [c.p.r.], as it appears in the title of relation `c/1` as well, since it also became part of version V2.

The user is able to delete a version by simply indicating its name. The system then removes automatically all the pointers defining it as well as the name of this one accompanying the relation titles in which it appears.

In Figure 1(c), we still consider the same HyperPro document presented in Figure 1(b). However, here the user modified the [c.p.r.] for relation `a/1` in section `1.1` and added a new predicate definition for relation `b/2`, modifying also its [c.p.r.] so as to point to its second predicate definition. However, the naming reference V1 previously defined still points to the previous predicate definition. The [def.] are not shown in this figure so as not to complicate the figure excessively.

Then, the user proceeds to define this version with the utility *name a version*, giving it the name V1 (or other name) and choosing the relation `a/1`. Then, a naming reference V1 is automatically put in the version bar after every title of all the relations of the version, along the [c.p.r.]/[def.] chain.

2.3 Program Testing

Once a version is defined, the user can test it, take a view of it, verify it syntactically or export it. To test a program or a version the user should specify the interpreter for a language he wants to test, from the available ones, in a menu. Then, he can choose one of the following test modes: program/version mode; clause/relation mode; automated recursive mode.

In the `program/version` mode, the user selects a named version wherever it appears in the document and then HyperPro opens a test window where the previously chosen interpreter is run and where the version is loaded, starting from the beginning of the program. The `clause/relation` mode allows the user to select any packet of clauses or relation which are loaded in the test window. If a relation is selected, it is loaded from its c.p.d. In the `automated recursive` mode, the user selects one relation in the document and the test utility loads it, following the [c.p.r]/[def.] chain in which the selected predicate definitions are included, starting from the the selected relation. It is the simplest way to test a small program or any set of few predicates, without having to explicitly define a program or a version.

The user can then run and test his program, version or packet of clauses in the test window. Therefore, the only thing that changes when testing a program in each mode is what is loaded from the document in the test window.

2.4 Document Exporting Facilities

HyperPro allows the user to export the whole document in four different formats: ASCII, LaTeX, Postscript and HTML. The ASCII exporting facility simply makes a dump of ASCII codes of the document into a file. It is useful during HyperPro development especially when the structure of the HyperPro document is changed. The LaTeXexporting facility dumps a HyperPro document into a file in atex format. The logical structure of the original document is entirely reflected in the LaTeXfile, except for the hypertext links, for obvious reasons. However, the indexes are mirrored in the LaTeXdocument, so that the hypertext version of the original document is faithfully rendered in paper, as much as possible. The table of contents is maintained in the LaTeXversion of the document. The HTML exporting facility makes the original HyperPro document viewable by any available web browser, where the original hypertext links appear as such in the HTML version of the document. It is also possible to export only part of the document like a version or a packet of clauses using appropriate menus.

3 Indexes

When presenting any hypertext document in two dimensions, as on paper, we obviously loose the hypertext links together with the linked information, and the hypertext facilities as well. The only way to render these informations again in two dimensions is by means of indexes. But this is not the only reason to build indexes. In fact, an index can present the information of all linked information linearly, allowing the user to access any node of the linked hypertext structure instantly, and not only through the linear link sequencing. Another advantage of building indexes, since the HyperPro editor allows the presentation of any part of a document in a separate view, is that we can present, through the index-based projections in HyperPro, parts of the original document by selecting the appropriate information directly from the indexes in a HyperPro document. We present each of the HyperPro indexes in the next subsection and the index-based projections in the following one.

A HyperPro document has two hypertext structures: one linking together information about relations and the other connecting relations together in program versions.There is an index for each of these hypertext structure, called *Cross Reference Index* and *Versions Index*, respectively. Each of these indexes is built independently through the Tools menu presented in the HyperPro editor. Once an index is built, the HyperPro system opens a new view for the corresponding index.

3.1 Cross Reference Index

The Cross Reference Index indicates the page number of every relation appearing in the document, the page number where the relation's current predicate definition is found, and the page numbers where the relation is used in other parts of the relations defined in the document, independently of versions. It is an absolute index of relation definitions. Each of these page entries is presented differently in the HyperPro document, as well as printed: the relation position in the document is shown in italic, the current predicate definition in bold, and each entry where the relations used in normal font. Figure 2 shows a Cross Reference Index built for a HyperPro document.

Since the index page entries are hypertext links, the user can access the corresponding part of the original document by just clicking its index page entry, and the main document view will present synchronously the part of the document corresponding to the entry.

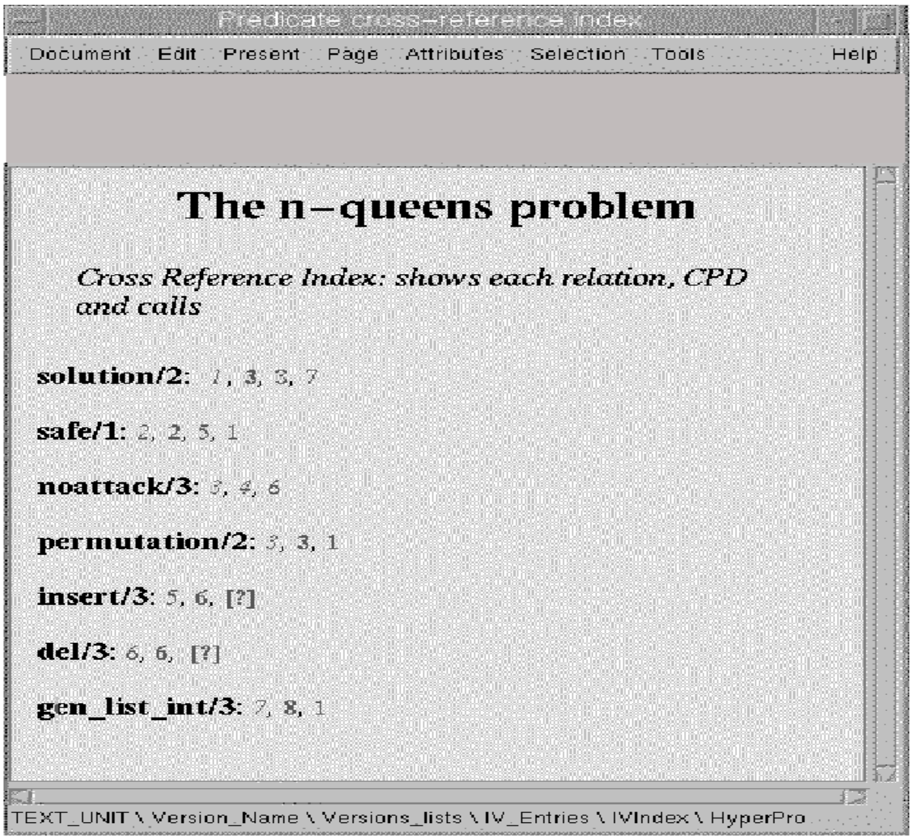

Figure 2: Cross reference index view image

3.2 Versions Index

The Versions Index presents, for each version, all the relations which are part of it, together with the document page number where the relation is defined. Figure 3 shows a Versions Index. The first entry indicates where the version was firstly defined, i.e. in which page of the document is found the predicate definition pointed to by the first named references appearing in the document, as all corresponding predicate definitions pointed to by the named references for each relation included in the version as well.

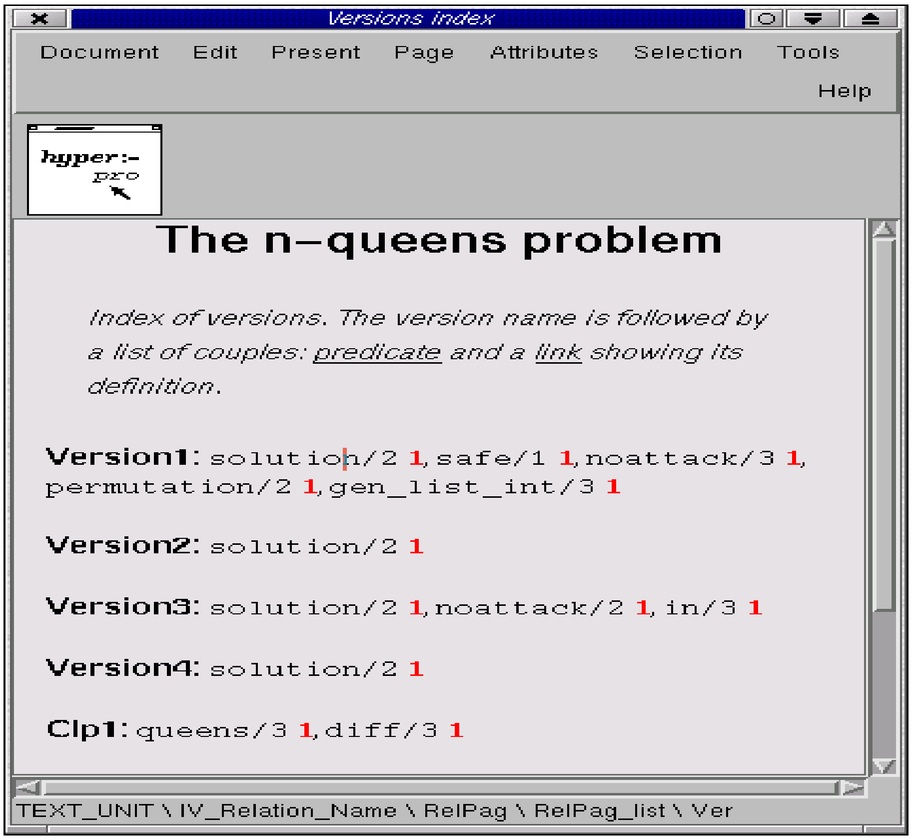

Figure 3: Version Index View Image

As for the Cross Reference Index, each page number of a relation entry in the Versions Index is a hypertext link to the corresponding place where the relation is defined. Even if there are more than one relation with the same name and arity, both defined in the same document page, each participating of a different version, the main view will show only the corresponding entry for each relation page entry link clicked.

4 Dynamic views

HyperPro allows the possibility to project and view separately some parts of the document selected by the user following some criteria.

The dynamic views (called also *projection views*) HyperPro offers are: manual projection view; automated projection view; version view; recursive projection view; index-based projection view.

A projection view displays selected portions of the document in a separate view. The selection processes depends on the projection wanted. Projections differ from static views on the selection process, which are already incorporated in Thot's machinery in the case of static views, but have to be provided dynamically by the HyperPro implementors in case of projections.

In the *Manual Projection View Mode* the user is free to select any part of the document, called elements, to be viewed. The granularity of the selected element is defined to be paragraphs and relation definitions. Therefore, when the user clicks in any of these elements or in their descendants, the projection will show the whole paragraph or relation definition.

In the *Recursive Projection View* the user selects one or more relations and HyperPro shows in a separate view all the packets of clauses in the [c.p.r.]/[def.] chain in which the relation is inserted, including obviously the packet pointed to by its own [c.p.r.]. This projection view is rather useful to help the user to detect which predications have not yet its [def.] set, or to find out where it is set to.

In the *Automated Projection View Mode*, the user gives some regular expression and HyperPro shows in a separate view all the portions of the document where the regular expression appears. The smallest granularity for this projection is a word.

The *Version Projection View* shows the user, in a separate view, all the packets of clauses composing a program or version chosen by the user in the document.

Index-Based Projection View: HyperPro offers a projection associated to indexes. Once an index view is built, as explained in Section 3, the user can choose in the Tools menu the `Projections` entry which will show a sub-menu where the different projections are presented. If the user chooses the `Index based` entry, clicking on an index entry, either `Versions` or `Cross Reference`, HyperPro will automatically show in another separate view what is related to the entry chosen in the index view. For instance, if the user clicks on a `Version Index view` entry, the projection will present in its view all the relations which are part of that version entry, and only them. Clicking on any part of the projection view will synchronously present the corresponding part of the document on the main view. In the case of the Cross Reference Index, the projection will show the relation corresponding to the entry the user clicked on. Figure 4 presents the image of a projection view for an entry of the Version Index shown in Figure 3.

5 HyperPro Interfaces

In this section we present the user's schematic interfaces such as menus and sub-menus for the functionalities and utilities that should be present in HyperPro. Some functionalities and utilities appear in specific Thot menus, like the export, views and index facilities. However, the manual and automated projection view, the version or program view, the recursive view and the index-based projection views facilities appear in the Tools menu, as the others utilities as well. The reason is that Thot allows anyone to include his own facilities using the Thot toolbox.

Views are selected in the menu *Views*. There the user chooses the entry point to the sub-menu *Open a view...* and in there, he chooses the view he wants to open, including the indexes views. The views the user can open are, therefore: the main view of the whole document; the table of contents; view of the comments; view of the assertions; view of the clauses; view of indexes.

The exporting facilities are accessed by the *Document* menu, and the *Save as...* sub-menu. A window opens where the user can then choose the exporting facility he wants to use. We will not present here the index facility interface. Tools menu offers the user access to the following utilities: Make indexes, Versions, Projections, Tests, Syntax verification and HyperPro preferences.

All facilities interface is done through *communications windows*, where the user controls the utility and where the data input is done, and some utilities may open a specific window to work as their output interface, as explained above.

6 HyperPro Versus Other Systems for Documentation

At the current state-of-the-art, there are no satisfactory tools or widely accepted methodologies for documenting `PROLOG` programs. Donald Knuth introduced literate programming in the form of Web, his tool for writing literate Pascal and C programs [2, 3]. The philosophy for documenting Pascal or **C** programs apparently offers the basis to establishing a methodology to document programs in the logic programming paradigm, but it seems not sufficient as we will see.

In the context of Web-like literate programming systems developed since 1984, the following are the most important documentation systems: 1) Knuth's Web for Pascal and **C** [2]; 2) Ramsey's Noweb [10]; 3) Thimbleby's Cweb [11], a variant of Knuth's Web; 4) Ramsey's Spider [9] which

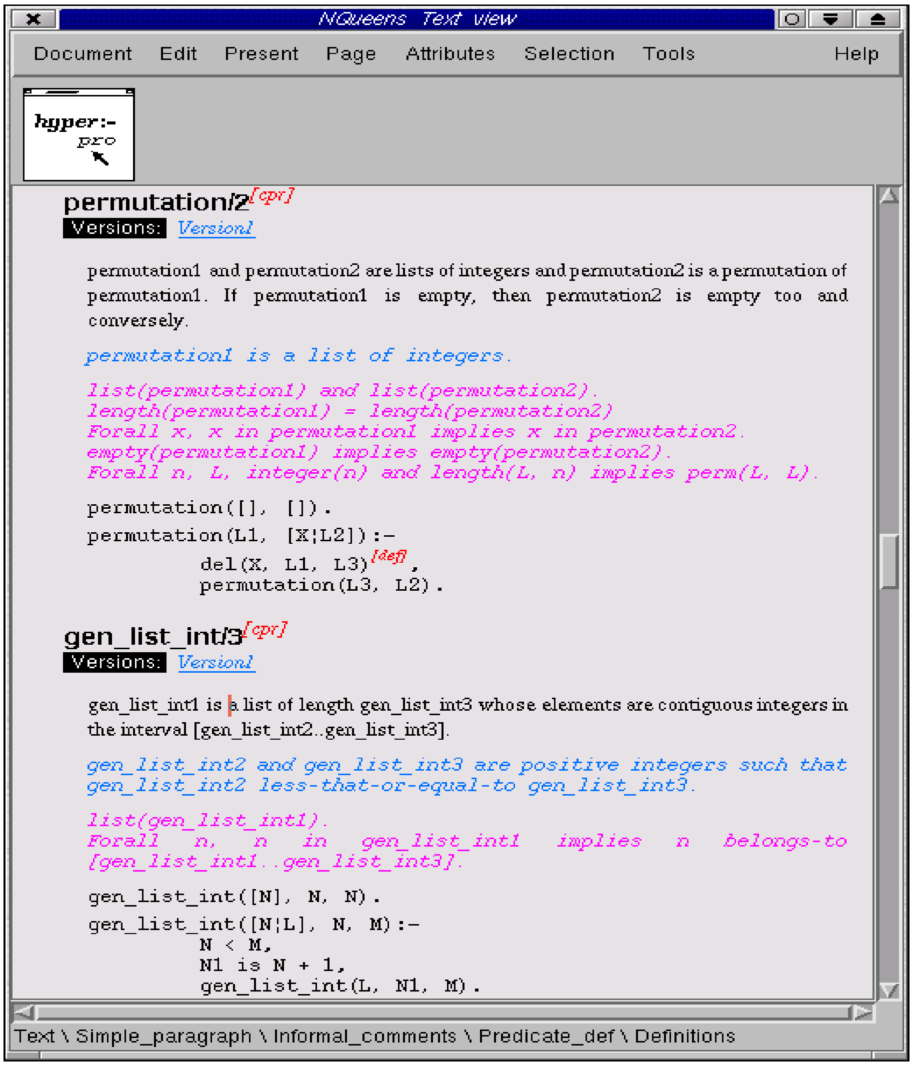

Figure 4: Version Index Projection View Image

is a Web generator. The basic idea behind *Literate Programming* is that programmers should use three languages: a typesetting language, such as LaTeX ; a programming language, such as Pascal, and a language which allows flexible combination of the typesetting and the programming texts into a single document. Thus, a literate program contains pieces of programs interleaved with descriptive texts. A literate programming system integrates these languages by providing tools to extract and process, from the input files, program texts and to generate documents containing summaries, index tables, cross-references, etc.

Cweb [11] is a tool to produce program documentation in a combination of C, the programming language, and *troff*, a text-formating language. The combined code and documentation can be processed and possibly typeset to result in a high-quality presentation including a table of contents, index, cross-referencing information, and related typographical conventions. Cweb differs from Web mainly in the choice of languages: Web is based on Pascal and TeX instead of C and *troff* or *nroff*.

Spider[9] was designed for developing verified Ada programs. The difficulty of using Web directly is that the intended target programming language is SSL (language for specifying structured editors), and the only languages for which Web implementations were available were Pascal and

C. Spider is a Web generator, akin to parser generators. Using Spider the user can build a Web without understanding the details of web's implementation, and can easily adjust that Web to change as a language definition changes.

Norman Ramsey has proposed a new literate programming system, called Noweb [10], which is intended to be a simple and extensible tool. It was developed on Unix and can be ported to non-Unix systems provided that they can simulate pipelines and support both ANSI-C and either awk or icon. Noweb can also work with HTML, the hypertext markup language for Netscape and the World-Wide Web. A Noweb file is a sequence of *chunks*. A chunk may contain code or documentation and may appear in any order. Code chunks contain program source code and references to other code chunks. Several code chunks may have the same name. Ramsey claims that Noweb is simpler than Knuth's Web due to its independence of the target programming language, but it also means that Noweb can do less. At last, Noweb works with both plain TeX and LaTeX. The system is extensible in the sense that new tools can be easily added to it, requiring no reprogramming. Its Weave tool preserves white spaces and program indentation when expanding chunks. Theses features are necessary to document program in languages like Miranda and Haskell, in which indentation is significant.

To conclude, no present system has gone beyond the experimental stage or beyond the capacity to handle small programs. Programming in the large with such systems is an objective still far to be reached. The problem of documenting while developing the programs is marginally considered.

Our purpose while developing HyperPro was to offer a tool which permits to record all the experiences accumulated when developing an application based on constraint logic programming, and maintaining it: programming, debugging, performing verifications, testing. The high level of expressiveness of constraint logic programming makes possible to consider a program as an executable specification. HyperPro presented in this paper consider that a CLP program is a unique document written with a methodology which takes into account the peculiar aspects of logic programming on one side. On the other side, it has the flexibility of a textual document. All the information concerning the program development and its maintainance is recorded in this single document.

7 Conclusion

We have presented a system based on the hypertext system Thot [5], called HyperPro, whose purpose is to facilitate logic programs development. It offers several facilities to view and handle documents at different levels of abstraction and from different point of view. Particularly, HyperPro aims to document CLP programs giving its users the possibility to edit, in a homogeneous and integrated environment, different versions of programs, comments about them, information for formal verification and debugging purposes, as well as the possibility to execute, debug and test the programs as well. All the attempts and development history of CLP programs can therefore be integrated and consistently documented within a unique environment gathering together a hypertext editor, different CLP interpreters and syntactical verifiers, as different debugging and verification tools as well. It also possesses several functions for exporting the document in different formats such as: html, latex, ascii, and producing projections, which are especial excerpts of the document, according to different criteria, such as, the program goal, pieces of code directly marked by the user, program versions, etc. HyperPro is convenient to create and maintain new programs. However, it still need program importation facilities in order to be able to incorporate and handle existing programs.

References

[1] P. Deransart, R. Bigonha, P. Parot, M. Bigonha, and J. de Siqueira. A hypertext based environment to write literate logic programs. In *I SBLP*, pages 1–16, 1996.

[2] Donald Knuth. The `web` system of structured documentation. Technical Report 980, Stanford Computer Science, 9 1983.

[3] Donald Knuth. *Literate Programming*. Number 27 in CSLI lecture notes. Center for the Study of Language and Information, 1992. pages 349–358.

[4] S. Montagne. Hyperpro(c) : un environnement pour la programmation en c. Rapport de stage de fin d'tudes, 1998.

[5] V. Quint. The thot user manual. Technical report, INRIA, 1995.

[6] V. Quint. The languages of thot. Technical report, INRIA, 6 1996. translated by Ethan Munson.

[7] V. Quint and I. Vatton. Grif: an interactive system for structured document manipulation. In *International Conference on Text Processing and document Manipulation*, pages 200–213. Cambridge University Press, 11 1986.

[8] V. Quint and I. Vatton. Hypertext aspects of the grif structured editor: Design and applications. Technical report, INRIA, 7 1992.

[9] Norman Ramsey. Literate programming: Weaving a language-independent `web`. *Communications of the ACM*, 32(9):1051–1055, 9 1989.

[10] Norman Ramsey. *The noweb Hacker's Guide*. Princeton University, 9 1992. Revised 08/1994.

[11] H. Thimbleby. Experiences of 'literate programming' using Cweb (a variant of knuth's `web`). *The Computer Journal*, 29(3):201–211, 1986.

[12] W3C. Xml w3c recommendation. http://www.w3c.org/XML.