Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Laboratório de Linguagens de Programação

Functionalities and Utilities of the HyperPro System

por

Mariza A. S. Bigonha, José de Siqueira, Roberto da S. Bigonha AbdelAli Ed-Dbali[1], Pierre Deransart[2] Fabrício M. Schmidt, Flávia Peligrinelli

LLP 010/2000

Av. Antônio Carlos, 6627

31270-010 - Belo Horizonte - MG

[1]University of Orléans(France)
[2]Inria - Institut National de Recherche en Informatique et en Automatique(France)

Abstract

The purpose of this paper is to present the Index Utility and the Index Based Projection of the HyperPro System. HyperPro is a hypertext tool that offers a way to handle two basic aspects: *Constraint Logic Programming* (CPL) documentation and development and text editing. For text editing it is based on the Thot system. A HyperPro program is a Thot document written in a report style. It is designed for logic programming but it can be adapted to other programming paradigms as well. HyperPro offers navigation and editing facilities, such indexes, document views and projections. It also offers document exportation. Projection is a mechanism for extracting and exporting relevant pieces of code according to specific criteria. Indexes are necessary to find the references and occurences 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 hypertexts links into paper references.

Keywords: Constraint Programming, HyperPro, Logic Programming, HyperText

1 Introduction

The HyperPro system [1, 5, 2, 3, 4, 6], developed by the authors, make use of the Thot editor and his API. 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 the possibility to execute, debug and test the program or program versions as well. All the attempts and development history of a CLP program can therefore be integrated and consistently documented within an unique environment gathering together a hypertext editor.

Thot is a structured editing tool with hypertext editing facilites. It was developed at INRIA and it is based in the logical aspect of the document. ([9],[10],[13],[12]). Thot uses a meta-model that allows the description of several models: article, book, technical report, etc. Its editor facility offers the possibility to integrate new applications to existing ones.

This paper describes the HyperPro functionalities which are: different statically views of the document; testing of different program versions; indexes; dynamic views; syntactical verification for different CLP languages; exportations. Among them, our focus will be in the projections and indexes facilities. Projection is a mechanism for extracting and exporting relevant pieces of code according to specific criteria. Index is a mechanism used to find the references and occurences 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.

In Section 3.3 we present the projections and in Section 3 two indexes for the HyperPro system and the associated projections: the Cross Reference Index and the Version Index. Although the Thot editor presents an index facility (see [9, 10] for details), the Thot index is a generic one, appropriate only for text. The two indexes presented are appropriate for our purposes since they reflect the hypertext structure of a Hyperpro document in two dimensions.

The HyperPro document is basically a Thot report document. It must have a title, a sequence of at least one section, a table of contents and a cross-reference index. Optionally, it can contain the date, the authors' names and their affiliations, keywords, bibliographical references, annexes and a versions index. An important aspect of HyperPro is that with few modifications it can be adapted to differents languages paradigms. 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 definiton 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 predicates definitions that define a unique program or different *versions* of the program.

A predicate definition contains : *informal comments*, *assertions* and *packet of clauses*, where, informal comments are a sequence of paragraphs and assertions are a sequence of lines of text and are optional. Packets of clauses can be Prolog or `clp(FD)` clauses. Clauses include directives, goals, facts and rules. The predications in the clauses body may have references or not to the relations which define them. This will actually define the current version of a program in a HyperPro document, as explained below. Every relation definition must have at least one predicate definition. When a predicate definiton is introduced, the fields for informal comments and packet of clauses are initially empty. The optional fields are presented if the user explicitly indicates so. However, the obligatory fields do not need to be filled in immediately after their insertion in the document.

2 HyperPro Functionalities

This section introduces the most important functionalities of HyperPro, which are the views of different parts of the document and indexes associated with different utilities, such as *program testing* and *syn-*

tactic verification for some pre-defined CLP and logic programming languages. These functionalities and utilities are defined in terms of programs versions.

2.1 Program Versions

A program is a set of packets of clauses. A document may contain only one program. It is up to the user to decide how the document will be organized, defining his program throughout subsections, sections or chapters. The user can then define a program by selecting conveniently predicate definitions in the document, and putting references on them. HyperPro provides the means to define, document and correctly manage his program, but it does not give means to solve any conflicts that could appear during these processes. Indeed, since two logic relations can be defined with the same name and the same arity anywhere in a document, this should be avoided by the user within the same program. HyperPro allows the user the possibility to manually define and automatically test, view and export his program.

The possibility of having diferents versions of a program is another important feature when defining, documenting and developping programs. Specially when developping and documenting CLP programs, the user needs to define, test and document different *versions* of his program. For that, he may define for any relation, different versions of its predicate definitions which are documented and managed with the same utilities used to define the program. Therefore, a program may have several different versions which can be defined, named, viewed and tested as much as any program. Indeed, program versions are programs which differ at least in one clause.

2.1.1 Defining Programs and their Versions

Every relation definition has at least a *current predicate definition* (c.p.d.), which is pointed by a hypertext reference placed in the relation title, called the *current predicate reference ([c.p.r.])*. A current predicate reference points to some predicate definition and it is explicitily introduced by the user for every relation defined in a HyperPro document. 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 a 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 *program references* ([p.r.]). The user does not have to put [p.r.] 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 [p.r.] 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 program references, which are presented in the text of the document as [p.r.]. As it is shown, they point to the relation titles they refer to. Note that there are no [p.r.] for predications recursively refering to the relation they are defining.

The [p.r.], together with the version naming utility, allows the user to effectively manage and document different versions of the program in a 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 [p.r.]. 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 [p.r.] of every relation linked in it.

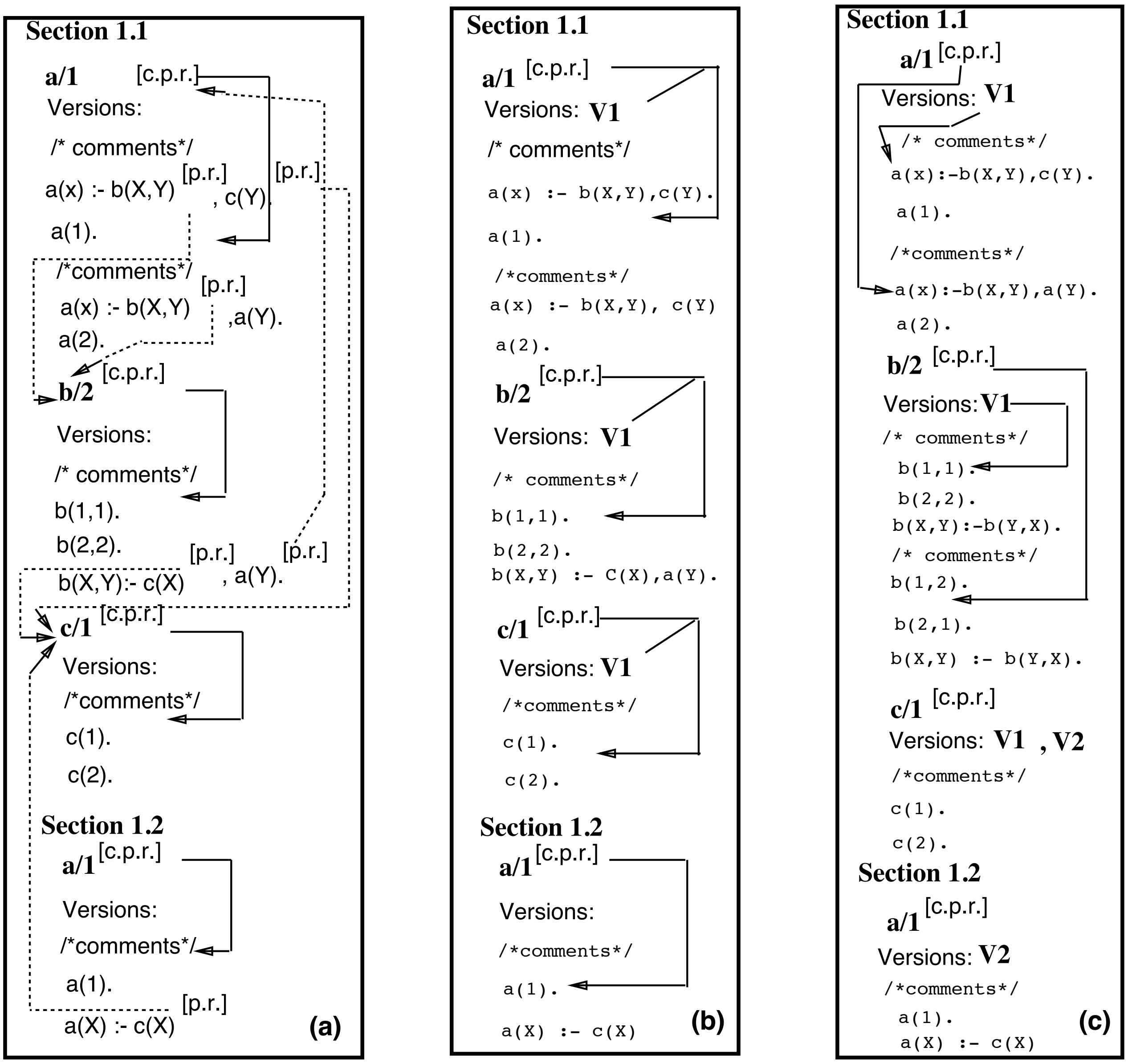


Figure 1: (a) Some Relations, C.P.R. and Version References (b) After Naming Version V1 (c) After naming Version V2

To name a version, once the [p.r.] for a program are manually inserted for the predications in the body of the c.p.d. as explained above, the user selects the utility *name a version* in Thot's Tools menu. A window opens and the user enters a name for the version. The user then chooses the first[3] 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 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 by the [c.p.r.].

Afterwards, the user entered a new section, numbered 1.2, where he defined relation a/1. In its

[3]This constraint in using this utility made easier its implementation.

predicate definition, the second clause for a/1 refers to relation c/1, previously defined in `Section 1.1`. Therefore, he puts a [p.r.] 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 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.

Naming reference, program reference or pointed predicate definition deletion causes the corresponding naming reference in the relation titles composing the program to be automatically deleted, since the version is not defined anymore. The user is able to delete a version by simply indicating its name to the system.

Whenever a program is defined and the user changes the [c.p.r.] of a relation composing it to point to another predicate definition, the user is defining a new program version. This is the simplest way to define a program version. However, the definition is not entirely completed, since the user has to insert [p.r.] in the predications at the body of the c.p.d. and then name it. This is done with the naming utility, as explained above. Note that, although the [c.p.r.] changed for the relation the user modified, all the naming references which pointed to the previous c.p.d. still define the corresponding versions. It is only a new version that is being defined and named, and all previous versions depending on other predicate definitions for this relation are still properly defined.

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 subsection 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 [p.r.] are not shown in this figure so as not to clobler the figure. Then, the user proceeds to define this version with the utility *name a version*, giving it the name V1 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.]/[p.r.] chain.

2.2 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 `version mode`, the user selects a named relation 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 begining 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 or more predicate definition in the document and the test utility loads it, following the [c.p.r]/[p.r.] chain in which the selected predicate definition is included, starting from the begining of the document. 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. The user can also import the program results into the document.

2.3 Syntactic Verification

HyperPro presents the user the possibility to syntactically verify whether any version of his program or predicate definitions is correct or not, according to some pre-defined syntax, chosen from a menu. HyperPro offers three ways to do syntactical verification, in the same way it is done for program testing, according to the selection mode: program/version mode; clause/relation mode; automated recursive mode.

In the program/version mode, a program or version is selected by its naming reference or name; in the clause/relation mode, a predicate definition is selected, either directly or by selecting a relation, and in this case, it is the c.p.d. for the relation which is selected; and in the automated recursive mode, a relation is selected and all the predicate definitions in its [c.p.r.]/[p.r.] chain are selected. In all modes, the selected clauses are verified by a syntactical verifier for the language chosen and its results are displayed in a separate window. The choices for syntactical verifiers include Standard Prolog and `clp(FD)` in HyperPro, and a future version should include syntactical verifiers for Prolog IV and Chip.

2.4 Document Exporting Facilities

HyperPro allows the user to export the whole document in three different formats: ASCII; Latex and HTML. The ASCII exporting facility simply makes a dump of ASCII codes of the document into a file. It is useful in case of HyperPro versions clash. The Latex exporting facility dumps a HyperPro document into a file in Latex format. The logical structure of the original document is entirely reflected in the Latex file, except for the hypertext links, for obvious reasons. However, the indexes are mirrored in the Latex document, 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 Latex version of the document. The HTML exporting facility makes the original HyperPro document to be viewed by any available web browser, where the original hypertext links appear as such in the HTML version of the document. Also, the user is able to export only the packet of clauses of a program or version to a file, by simply saving the packet of clauses shown in a view as the result of the program or version view or recursive view utilities.

3 Indexes

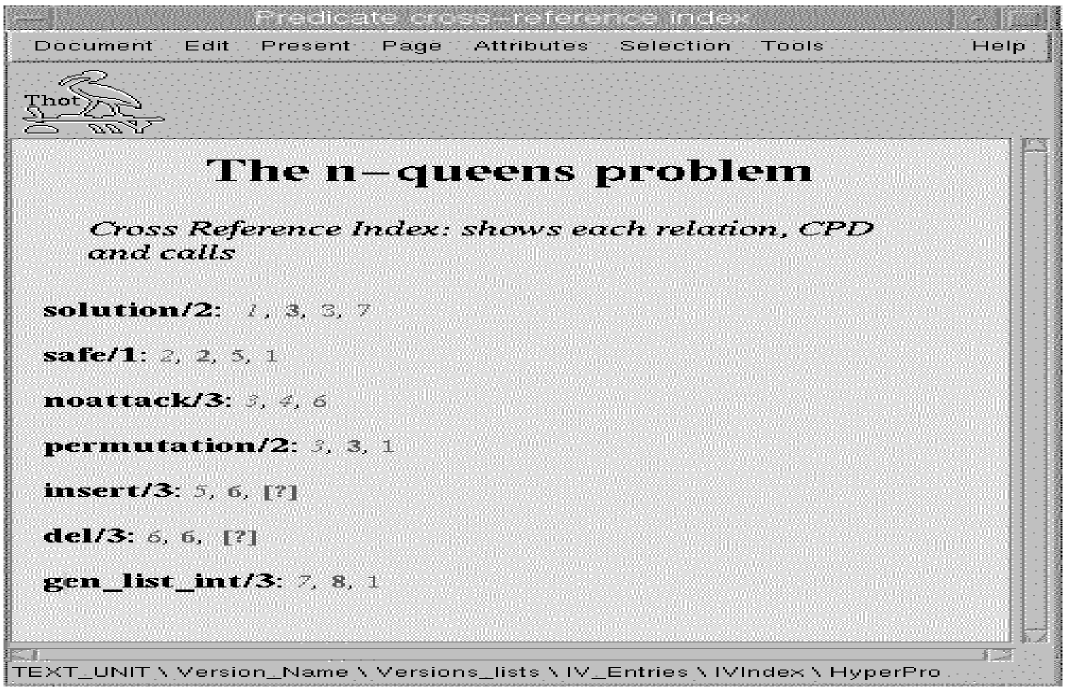
When presenting any hypertext document in two dimensions, as on paper, we obviously loose the hypertext links together with the linked information, and the hypertext facilities as well. The only way to render these informations again in two dimensions is by means of indexes. But this is not the only reason to build indexes. In fact, an index can present the information of all linked information linearly, allowing the user to acess any node of the linked hypertext structure instantly, and not only through the linear link sequencing. Another advantage of building indexes, since the Thot editor allows to present any part of a HyperPro 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 relations information and the other connecting relations together in program versions, as explained in Section 2.1.1. There is an index for each of these hypertext structure, called *Cross Reference Index* and *Versions Index*, respectively. Each of these indexes are built independently through the Tools menu presented in the Thot 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 is found the relation current predicate definition, 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 raltionis 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.


Predicate cross-reference index
Document
Edit
Present
Page
Attributes
Selection
Tools
Help
Thot
TEXT_UNIT \ Version_Name \ Versions_lists \ IV_Entries \ IVIndex \ HyperPro

The n–queens problem

Cross Reference Index: shows each relation, CPD and calls

solution/2: *1*, **3**, 3, 7

safe/1: *2*, **2**, 5, 1

noattack/3: *3*, *4*, *6*

permutation/2: *3*, **3**, 1

insert/3: *5*, **6**, **[?]**

del/3: *6*, **6**, **[?]**

gen_list_int/3: *7*, **8**, 1

Figure 2: Cross reference index view image

3.2 Versions Index

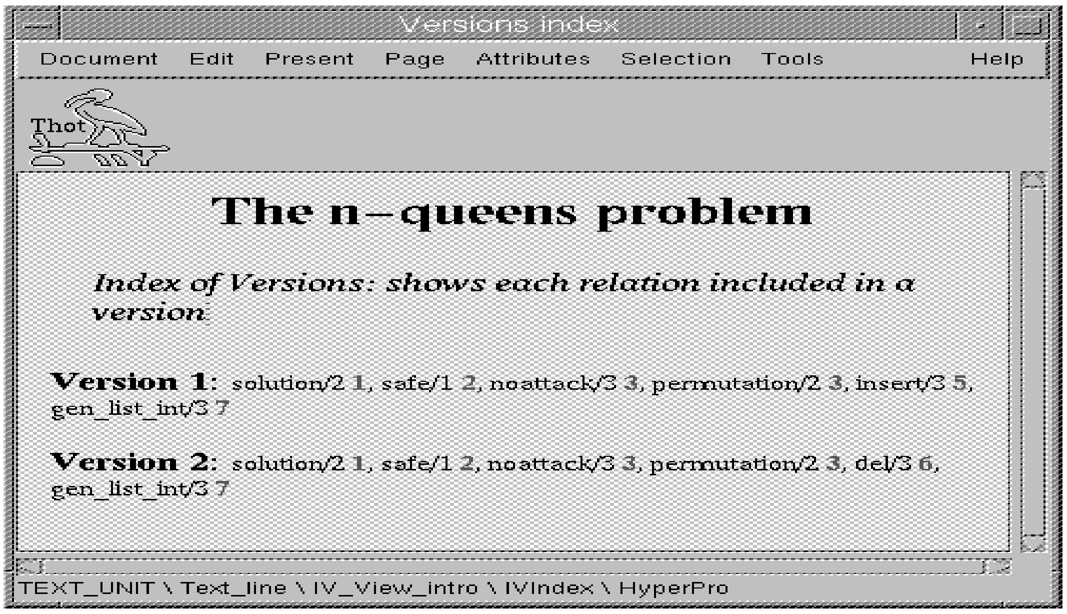
The Versions Index presents, for each version, all the relations which are part of it, together with the document page number where the relation is defined. Figure 3 shows a Versions Index. The first entry indicates where the version was firstly defined, i.e. in which page of the document is found the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

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.

3.3 Views and Projections

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

HyperPro provides several different views of the same document: the main view of the whole document; the table of contents; view of the comments; view of the assertions; view of the packets. The main view and the table of contents are displayed automatically, when a document is open. The



The n−queens problem

Index of Versions: shows each relation included in a version:

Version 1: solution/2 1, safe/1 2, noattack/3 3, permutation/2 3, insert/3 5, gen_list_int/3 7

Version 2: solution/2 1, safe/1 2, noattack/3 3, permutation/2 3, del/3 6, gen_list_int/3 7

TEXT_UNIT \ Text_line \ IV_View_intro \ IVIndex \ HyperPro

Figure 3: Version Index View Image

main view shows all the document as it is defined by the user, and while he defines it. It contains all the subsections, sections and chapters defined, with all the relations definitons and versions references. The table of contents displays all the subsections, sections and chapters defined in the document and their references, such as their ordering number and page. The views of comments, of assertions, and of packets are shown on demand by the user through the appropriate Thot menu bar. Each view displays all and only the comments, or assertions or packets defined in the document.

The 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 processess depends on the projection wanted. Projections differ from views on the selection process, which are already incorporated in Thot's machinery in the case of views, but have to be provided dynamically by the HyperPro implementors in case of projections.

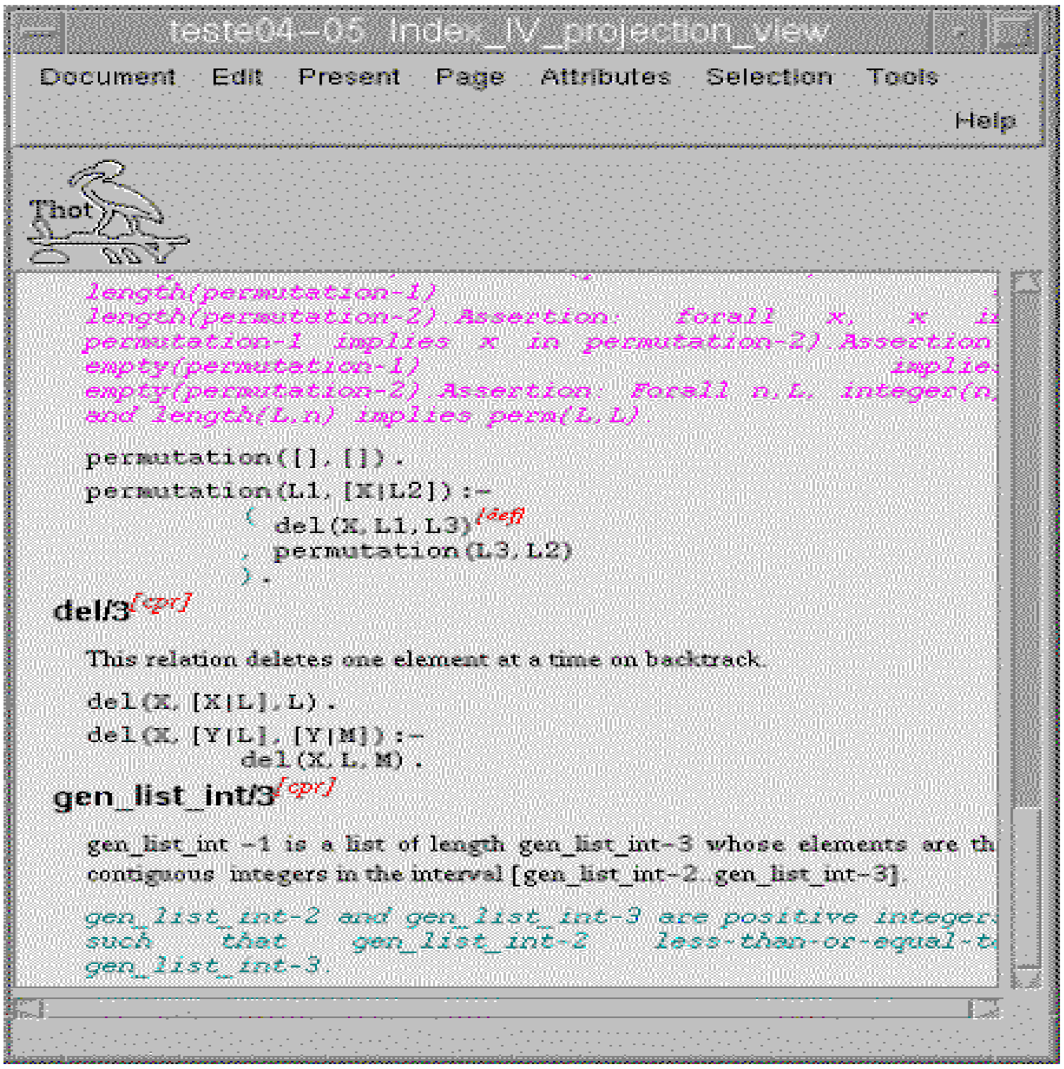
In the *Manual Projection View Mode* the user is free to select any part of the document, called elements, to be viewed. The granularity of the selected element is defined to be paragraphs and relation definitions. Therefore, when the user clicks in any of these elements or in their descendants, the 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.]/[p.r.] chain in which the relation is inserted, including obviously the packet pointed by its own [c.p.r.]. This projection view is rather useful to help the user to detect which predications have not yet its [p.r.] 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 submenu 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.



length(permutation-1)
length(permutation-2).Assertion: forall x, x i
permutation-1 implies x in permutation-2).Assertion
empty(permutation-1) implie
empty(permutation-2).Assertion: Forall n,L, integer(n,
and length(L,n) implies perm(L,L).

```
permutation([],[]).
permutation(L1,[X|L2]):-
            ( del(X,L1,L3)[def]
            , permutation(L3,L2)
            ).
```

del/3[cpr]

This relation deletes one element at a time on backtrack.

```
del(X,[X|L],L).
del(X,[Y|L],[Y|M]):-
            del(X,L,M).
```

gen_list_int/3[cpr]

gen_list_int -1 is a list of length gen_list_int-3 whose elements are th
contiguous integers in the interval [gen_list_int-2..gen_list_int-3].

gen_list_int-2 and gen_list_int-3 are positive integer
such that gen_list_int-2 less-than-or-equal-t
gen_list_int-3.

Figure 4: Version Index Projection View Image

4 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 Thot's Tools menu, as the others utilities as well. The reason is that Thot allows anyone to include his own facilities using the Thot toolbox, which are acessible through the Tools menu.

Views are selected in the Thot 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 acessed 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. The reader can find about it and how to use this facility in [2, 3]. For projection consult [5, 4] to see for each projection its dialog box. Thot's tools menu offers the user access to the following utilities: Make indexes, Versions, Projections, Tests, Sintax 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.

5 HyperPro Versus Other Systems for Documentation

Logic program development has been considered by different authors from different point of views. Most of them consider program development as a transformation process from a specification to an efficient Prolog program. Deville [17] starts from first order logical formulas and "mode" declarations. More generally a mode declarations can be viewed as type declarations. Most of the systems intend to help the programmer in developing correct programs, or verifying afterwards that the program satisfies some properties. In logic programming different kind of proof systems have been designed. In [20] one of these system is described. In [15] a systematic approach of logic program validation is presented. Some of the ideas have been implemented in the system LDS2 described in [22] and used to define a methodology for writing specifications in logic programing style [14].

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 [18, 19]. The philosophy for documenting Pascal or **C** programs apparentlty offers the basis to establishing a methodology to document programs in the logic programming paradigm, but it seems not sufficient as we shall 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** [18]; 2) Ramsey's Noweb [21]; 3) Thimbleby's Cweb [23], a variant of Knuth's Web; 4) Ramsey's Spider [24] which 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 [23] 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[24] 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 [21], 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 developping the programs is marginally considered.

Our purpose while developping HyperPro was to offer a tool which permits to record all the experiences accumulated when developping 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 informations concerning the program development and its maintainance is recorded in this single document.

6 Conclusion

We presented here an system based on the hypertext system Thot [12], called HyperPro, whose purpose is to handle such documents and 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 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.

References

[1] Bigonha, Mariza A.S., Siqueira, José de , Bigonha, Roberto S., Ed-Dbali, AbdelAli, Deransart, Pierre, Schmidt, Fabrício, Peligrinelli, Flávia, *Sistema de Indexação e Projeções de HyperPro*, Technical report of the Programming Language Laboratory, DCC/UFMG, LLP006/2000.

[2] Siqueira, José de, Schmidt, Fabrício, *Manual do Usuário para Índices e Projeções Baseadas em Índices para o HyperPro Básico*, UFMG, RT DCC 013/1999.

[3] Siqueira, José de, Schmidt, Fabrício, *Manual de Sistema para Índices e Projeções Baseadas em Índices para o HyperPro Básico*, RT DCC 012/1999, UFMG.

[4] Peligrinelli, Flavia, Bigonha, Mariza, *HyperPro: Sistema de Programa e Documentação em um Ambiente de Programação Baseado no Paradigma Literário*, Technical report of the Programming Language Laboratory, DCC- UFMG, LLP003/1999, 03/1999.

[5] Bigonha, Mariza A.S., Ed-Dbali, AbdelAli, Bigonha, Roberto S., Peligrinelli, Flavia, Deransart, Pierre, Siqueira, J. de, *Projection of HyperPro Document*, III ISBLP, 1999, pages:171-183.

[6] Pierre Deransart AbdelAli Ed-Dbali Mariza A. S. Bigonha Roberto S. Bigonha Jose de Siqueira, "Basic HyperPro Functionalities and Utilities", *RT 023/97, DCC-UFMG*, 12/1997.

[7] Quint, V. and Vatton, I., *The Thot Kit API*, INRIA Rocquencourt, technical report, 07/10/1997.

[8] Deransart, P., Bigonha, R., Parot, P., Bigonha, M., Siqueira, J. de, *A Hypertext Based Environment to Write Literate Logic Programs*, I SBLP, 1996, pages:1-16.

[9] Quint, V. and Vatton, I., *Grif: an interactive System for structured Document Manipulation*, Proceedings of the International Conference on Text Processing and document Manipulation, 1986, November, 200-213, Cambridge University Press.

[10] Quint, Vicent & Vatton, Iréne, *Hypertext Aspects of the GRIF Structured Editor: Design and Applications*, Rappports de Recherche # 1734, Julliet 1992.

[11] M. Bergère and G. Ferrand et alii, *La Programmation Logique avec Contraintes Revisitée en Termes d'Arbre de Preuve et de Squelettes*, Orléans, 1995, LIFO 96-06.

[12] Quint, V., *The Thot User Manual*, Internal report, INRIA-CNRS, 1995.

[13] Quint, V., *The Languages of Thot*, Internal report, INRIA-CNRS, translated by Ethan Munson, version of 06/25/1996.

[14] Abdelali Ed-Dbali and Pierre Deransart, *Software Formal Specification by Logic programming*, Logic Programming Summer School, Zurich, N. E. Fuchs and G. Comyn, 1992, Zurich, Suisse, Springer Verlag, LNAI, 636, 278–289.

[15] Deransart, Pierre and Małuszyński, Jan, The MIT Press, *A Grammatical View of Logic Programming*, 11/1993.

[16] Deransart, Pierre and Ed-Dbali, Abdelali and Cervoni, Laurent, Springer Verlag, *Prolog, The Standard; Reference Manual*, 1996.

[17] Deville, Yves, Addison Wesley, *Logic Programming: Systematic Program Development*, 1990.

[18] Knuth, Donald, *The Web System of Structured Documentation*, Technical Report 980, Stanford Computer Science, California, 09/1983.

[19] Knuth, Donald, *Literate Programming*, CSLI lecture notes, Stanford, CA, Center for the study of language and information, 1992, 27, 349–358.

[20] Loveland, D., *Near-Horn Prolog and Beyond*, Journal of automated Reasoning, 1991, 1, 1–26.

[21] Ramsey Norman, *The noweb Hacker's Guide*, CSD, Princeton University, 09/1992 (Revised 08/1994).

[22] S. Renault and P. Deransart, *Design of Redundant Formal Specifications by Logic Programming: Merging Formal Text and Good Comments*, International Journal of Software Engineering and Knowledge Engineering, 1994, 4, 3.

[23] Thimbleby, H., *Experiences of 'Literate Programming' using Cweb(a variant of Knuth's Web)*, The Computer Journal, Vol. 29, No. 3, 201-211, 1986.

[24] Ramsey Norman, *Literate Programming: Weaving a language-independent Web* , Communications of the ACM, 32(9): 1051-1055, 09/1989.