A Literate Logic Programming System

Pierre Deransart [1]
Roberto da Silva Bigonha
Patrick Parot [2]
Mariza Andrade da Silva Bigonha
José de Siqueira

Universidade Federal de Minas Gerais
Departamento de Ciência da Computação

Abstract

The purpose of this paper is to present an experimental hypertext programming environment for PROLOG based dialects and its application to some logic program development according to a logic programming methodology. The genericity of the tool makes it easily adaptable to other logic programming languages and to other applications in the field of logic programs development, in particular to handle logic programs with constraints. The proposed tool permits to record all the experiences accumulated during the life cycle of a software.

1 The Software Documentation Problem

Software documentation is a perennial problem and a very important issue which still does not have a satisfactory solution. Software systems evolve along their lifetime and thus require continuous maintenance, which is usually expensive and difficult to accomplish because most systems are poorly documented. Indeed, software documentation is often nonexistent, incomplete or out-of-date.

Information contents of software documentation are naturally redundant, since the main purpose of any documented software is to provide at least two alternative views of the same material: the program view for the machine and the text view in a literate style for human consumption. It is also a fact that program documentation tends to become a large collection of files, which is prone to discourage programmers to keep the text part of the documentation up-to-date to the corresponding software. A good documentation system should then provide ways to check automatically inconsistency of this kind, perhaps by creating strong ties between the documentation and its related pieces of programs. The consistency between descriptive texts and the corresponding program code should be always enforced somehow.

Additionally, modern program documentation must be computer processable and *compilable*, so errors in coding can be detected earlier in the design process. All error messages,

[1] Institut National de Recherche en Informatique et en Automatique–France
[2] Service de Cooperation (CSN) France

including those produced by compilers, must refer to the original document files, so that programmers should be encourage to do debugging or testing only on the original document files.

Documentation of software must also be organized in a way to provide different levels of abstraction of the documented software in order to help the understanding of large systems.

2 Characteristics of Logic Programming

Logic Programming is issued from researches in Artificial Intelligence and Logics. Software engineering tradition has been of relatively few influence on discovery and design of programming languages and adapted environments based on the new logical paradigms. Furthermore the axiomatic form of logic programming makes programs similar to executable specifications, and therefore more attention has been given to their use in rapid prototyping rather than their use in very large applications. Logic program documentation and verification have thus received relatively little attention. In contrast implementations have been considerably improved and new logic programming languages compete favorably with commonly used imperative languages. Although a language like Prolog has reached an industrial maturity (there exists now an ISO standard [5, 9]), it is not as extensively used as it could be. People realize now that the lack of well adapted development tools is one of the reasons for its painful growing.

On the other side, the positive qualities of logic programming (highly declarative language, versatility, natural modularity, robustness) recommend various industrial application (see [3] for a survey), also limited by the lack of good development tools. Nowaday the need for such tools is well recognized. Classical tools, developed for imperative or functional programming, cannot be directly re-used. The peculiarities of logic programming demand for well adapted tools which take into account the high level and declarative aspects of the language.

With Constraint Logic Programming (CLP) the situation is even more acute: CLP programs are relatively short pieces of code, but often constitute the sensible kernel of an application. Due to their high level of expressiveness they are closer to a specification rather than to a traditional program. Furthermore, due to commitment to efficiency of implementations, the same CLP program is maintained along the whole life cycle of the application: from its conception until its final uses and further improvements or updating. This has an important consequence: there must be some way to maintain also the whole documentation concerning the program of an application, during its whole life.

3 Logic Programming Tools

Logic program development has been considered by different authors from different points of view. Most of them consider program development as a transformation process from a specification to an efficient Prolog program. Deville [6] starts from first order logical formulas and "mode" declarations. More generally mode declarations can be viewed as type declarations. His ideas have been experimented in the FOLON system [7]. The system

imposes a strict discipline in logic program development and performs automatically some optimizing transformations. Leon Sterling (Case Western Research University, Ohio, USA) and Jan Komorowski (University of Trondheim, Norway) developed pure logic program transformation systems to progressively (and semi automatically) specialize programs by applying transformation rules. A complete survey on logic program transformation can be found in [19].

Most of the systems are intended to help the programmer in developing correct programs, or verifying afterwards that the program satisfies some properties. In logic programming different kinds of proof systems have been designed. In [17] one of these system is described. In [4] a systematic approach for logic program validation is presented. Some of the ideas have been implemented in the system LDS2 described in [30] and used to define a methodology for writing specifications in logic programming style [1, 2].

4 Other Systems for Documentation

At the current state-of-the-art, there are no satisfactory tools or widely accepted methodologies for documenting PROLOG programs. Knuth's *Literate Programming* philosophy for documenting Pascal or **C** programs [11, 13, 14, 12] apparently 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 followings are the most important documentation systems: 1) Knuth's Web for Pascal and **C** [11, 12, 14]; 2) Ramsey's Noweb [25, 24, 26]; 3) Thimbleby's Cweb [32], a variant of Knuth's Web; 4) Ramsey's Spider [27], which is a Web generator.

The basic idea behind *Literate Programming* is founded on three languages: a typesetting language, such as LaTeX [15]; a programming language, such as Pascal, and a language which allows flexible combination of the typesetting and the programming texts into a single document. 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. The result is called 'literate programming' because the final document is not only readable, but may, according to Knuth [12], actually be appreciated as literature work.

Donald Knuth introduced literate programming in the form of Web, his tool for writing literate Pascal programs [11]. Since then, many other systems have been designed in order to satisfy particular styles of literate programming. In the mid '80s, Web was adapted to programming languages other than Pascal, including C, Modula-2, Fortran, Ada, and others [32, 27].

Cweb [32] 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 Knuth's Web system mainly in the choice of languages: Web is based on Pascal and TeX instead of C and *troff* (or *nroff*) as in Cweb.

Spider[27] was designed for developing verified Ada programs. The difficulty of using Web directly is that the 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 in fact 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 language definition changes.

Recently, Norman Ramsey has proposed a new literate programming system, called Noweb [25, 24, 26], 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 texts, 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. Noweb tools are implemented as pipelines. Each pipeline begins with the Noweb source file. Successive stages of the pipeline implement simple transformations of the source until the desired results appears in the end of the pipeline. Users change or extend Noweb by inserting or removing pipelines stages rather than recompiling it.

Ramsey claims that Noweb is simpler than Knuth's Web due to its independence on the target programming language, but it also means that Noweb can do less. 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 programs in languages like Miranda and Haskell, in which indentation is significant. In Noweb one can extract more than one program from a single source file. It also generates compiler directives so as to give the underlying compiler the location of lines on the original input files. This facility helps issuing good error messages.

Most of the differences between Web and Noweb come from the fact that Web has language-dependent features which are not present in Noweb. Web works poorly with LaTeX, which cannot be used in Web source, and requires tedious adjustments by hand to get weave output to work in LaTeX documents. At last, Noweb works with both plain TeX and LaTeX. Web takes the monolithic view of literate programming, while Noweb's approach is to compose simple tools that manipulate files in the Noweb format.

Modern tools [26], like Nuweb and Funnelweb, are also language-independent. To users Noweb looks very similar to Nuweb. There are only minor syntactic differences: Nuweb uses markup within the source file instead of command-line options to show things like the names of output files, but both are simple ad easy to master. Funnelweb is a complex tool that includes its own typesetting language and command shell.

Concluding, no system today 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. Moreover the problem of documenting the programs is marginally considered.

Our purpose is thus to offer a tool which permits to record all the experiences accumulated when developing an application based on (constraint) logic programming, and when maintaining it. The high level of expressiveness of constraint logic programming makes

possible to consider a program as an executable specification. It is thus quite clear what such system has to provide: i) an easy way to mixture natural language comments and program with text editing facilities; ii) program debugging functionalities and iii) validation tools with easy interfaces.

Following this idea, we propose to consider a CLP program as a unique document written with a methodology which takes into account the peculiar aspects of logic programming. All information concerning the program development and its maintainance will be recorded in this single document. Obviously such a document will grow very quickly and therefore functions to help writers or readers to handle and to use it must be defined. The experimental system, called HyperPro, we propose is based on the hypertext system Thot [23]. The purpose of HyperPro is to handle such documents and facilitate logic programs development.

We also present a methodology to develop and document logic program based on the methods for a structured elaboration of code and comments that have been investigated by Deransart and Renault [31].

5 Proposed Methodology

The HyperPro system offers a way to handle two basic aspects: text editing and CLP programming. For text editing it uses the Thot system [23]. A HyperPro program is a Thot document written in a report style.

A HyperPro program contains also specific paragraphs which correspond to relation definitions. Their format reflects strictly the methodology required for CLP program development. The methodology is based on the works described in [1, 4, 16, 30]. It uses simple basic principles: in CLP the program unit is a packet of clauses characterizing a relation. Thanks to the declarative aspect of relational programming a relation definition may be understood by just looking at the clauses and the informal definitions of the predicates used in their bodies. The nature of comments is obviously important: it must bring a redundant but different information. For such purpose different kinds of informations must be provided, which are precisely defined in the methodology. On the other side, the text editing system must provide facilities to navigate inside the program and its comments.

We first explain the basic piece of the methodology (the relation definition), then we describe the different functionalities of the HyperPro system.

5.1 The Relation Definitions

Relation definitions (RD) consists of two items: one containing the name and arity of the predicate, say, for example, `safe/1` and the second being a sequence of predicate definitions. Each predicate definition is built on the same model and has two to four items: one informal comment (called **Definition**), optional type statements (called **Types**), optional assertions (called **Assertions**) and the packet of clauses defining the relation. Different predicate definitions in a RD correspond to different versions of the same relation. A relation is uniquely defined by its name and arity. However, the same relation may appear in different sections of a document during program development. In that case it will not be considered as the same relation (see Section 5.2). Here is an example:

safe/1

Definition:
If $safe_1$ is a list of positive integers then all the points of coordinates x/y, where y is the integer of rank x in the list, are not on the same row neither on the same diagonal.

Types:
$safe_1$ is a list of integers.

Assertions:
For all y,v, (y in $safe_1$ and v in $safe_1$ and rank-of(y,x) and rank-of(v,u) and not x = u) implies (not y = v and not x-u = y-v and not x-u = v-y)

```
safe([]).

safe([Queen|Others]) :-
    safe(Others),
    noattack(Queen, Other, 1).
```

Notice that it is possible to give several types and assertions. Arguments of a predicate **pred** of arity n are denoted $\mathbf{pred}_1$ to $\mathbf{pred}_n$ everywhere. Each information plays a specific role and contributes to a clear understanding of a predicate definition:

- **Definition:** it is an informal comment which characterizes the semantics of the defined relation. It must be a partial correctness property, i.e., a property of all possible argument values satisfying the relation.
- **Types:** There are formal or informal expressions. They characterize the form of the arguments either when the predicate is used, or the kind of solutions of interest. Sometimes both informations coincide. It is the case here: $safe_1$ must be a list of integers when calling **safe/1**. But if one considers this predicate alone it has non ground solutions, and only solutions consisting of lists of integers are interesting.
- **Assertions:** Different assertions are possible, depending on the kind of intended use. Here one uses partial correctness assertions [4] written in a formal language: a first order formula. The formula states clearly that the argument denotes points corresponding to reciprocally non-attacking queens.
- **Packet:** The packet of clauses.

To understand a predicate it should be sufficient to read its definition and those used in it. For example safe/1 uses noattack/3 defined as: If $noattack_1$ denotes a point of coordinates (x, $noattack_1$) and $noattack_3$ is a positive integer and $noattack_2$ denotes a list of points at an horizontal distance $noattack_3$ from x, then no point (x+$noattack_3$+r−1,v), where v is the element of rank r in $noattack_2$, is on the same diagonal.

Therefore, an informal "logical" reading of the second clause should be, after obvious simplifications:

If (**Definition of noattack/3:**) if Queen denotes a point of coordinates (x, Queen) and Other denotes a list of points after x, then no point (x + r, v), where v is the element of rank r in Other, is on the same diagonal, and

If (**Definition of safe/2 in the body:**) if Others is a list of positive integers, then all the points of coordinates x/y, where y is the integer of rank x in the list, are not on the same row neither on the same diagonal,

Then (**Definition of safe/2 in the head:**) if [Queen | Others] is a list of positive integers, then all the points of coordinates x/y, where y is the integer of rank x in the list, are not on the same row neither on the same diagonal.

The implication is obviously satisfied. Such reading helps to understand the clausal predicate definition. It comes as an additional useful information.

The formal assertions may also be used. However, they are intended to be used by a proof system interfaced with the HyperPro system.

It happens usually that such local definition is not sufficient to explain the program. It is the purpose of the report style. At any place (but not inside a predicate definition) more comments may be added.

5.2 Navigation Facilities: index and pointers

Writing a document in a disciplined manner is not easy without navigating facilities. We have considered two ways of navigation: references to definitions and indexes.

References include references to RD's and references to predicate definition inside an RD. Anywhere in the document an RD may be referenced, in particular it is associated with every predication in a clause body. With the name/arity of a relation, an associated reference points to the current version. With such pointers it is possible to identify at any time the current version which is under development or testing.

Indexes offer the possibility to the user to focus an important notion and to point to all the parts of the text where this notion or related notions are used. Index is a true thesaurus and helps considerably to maintain the consistency of the text. Predicate cross references are also part of the index. They are built automatically.

5.3 Editing Facilities: document views and projections

The main originality of the HyperPro system is the possibility to use views and projections instead of the full text. In fact the document grows and becomes intractable very quickly. The HyperPro's synchronized views approach helps the user to write the program. Four basic views are provided corresponding to the items in a predicate definition: the comments view (**Definition**), the type view (**Types**), the assertion view (**Assertions**) and the program view (**packet**).

The program view offers the possibility to program directly, as usual, with the editing help of syntactic menus. Each view corresponds to a window with the pertinent information only, but concerning all the defined relations in the document. All the views are synchronized in such a way that when pointing, for example, to a clause of a predicate definition in the "program view", the corresponding informal definition is selected in the "comment view" and also in the other views.

Projections are views corresponding to selected portions of text which are gathered into a single window. The selection criterion is on the same basis as for a "grep" function. There is also the possibility to open a view of the index.

The combination of all these views offers great possibilities of navigation and facilitate construction, consultation and maintainance of the program/document.

5.4 Document Exportation: interfaces

The purpose of the system is not just to write a program but also to test it or to make some experimentation with it. Therefore there are possibilities of exportation. Two kinds of program exportation have been considered: syntactic one and global one. The first one is used to test the syntax of lines of code: the user may select some lines and a language (Prolog or CLP(fd), for example) and the system runs the corresponding compiler on the considered code and reports syntactic errors.

The global program exportation consists, for example, of selecting a goal or a portion of text and a language, and the system create a file with the corresponding program (built from the current versions) and opens a window to run the corresponding processor and to inspect the program.

Finally Thot also offers the possibility to define sophisticated applications. In particular interfaces with proof systems are possible. In this case, there is the possibility to export assertions with the purpose to run a proof system and, thus, to report in the program documentation text the results of the validation activity. We intend to integrate the HyperPro prototype with the LDS2 system [30] and with the theorem prover SEQUOIA [10].

6 The HyperPro System

6.1 Thot: principles and main concepts

Thot allows the user to create, modify and consult interactively documents that comply with models. These models enable the production of homogeneous documents. Formatting and typography are handled by the editor itself: thus, the user mainly focus his attention on the logic organization and on the contents of documents. Thot performs automatically other operations such as numbering, updating cross references, building index tables, spelling correction, etc (see [23] and [28] for further information).

6.1.1 Generic Structure

Functionalities and services provided by the Thot editor are mainly based on the internal representation of documents in the editor. This representation derives from a document model that can be specified by the user through a description language [22] provided by Thot. Such document model allows the user or an external system to operate on the different logic elements that form the document organization. Those elements, such as chapters, sections, paragraphs and notes are entities that compose the logical structure of a document class. This structure specifies types of usable elements and relationships that can relate them. Specifications of logic entities and their relations describe the generic structure (or generic model) of a document. Each instance of a generic structure is called specific structure. A class of documents is a set of documents having similar structure

(*e.g.* articles, reports, books and thesis); more formally, it means that a class is a set of documents whose specific structure is built according to the same generic structure.

The editor itself ensures that each document being handled complies with its generic model. It only allows operations which preserve logical structure consistency according to the document generic structure [18]. The editor also uses the generic structure to guide the user during its writing process, and to generate automatically document chunks, potentially empty, to be filled.

6.1.2 Generic Presentation

Since the generic structure describes organization of a document class, it is possible to specify presentation models for the considered class; the user defines presentation rules that will be applied to all documents of the given class. Such a presentation can be very fine-grained because rules may be applied to all kinds of entities that are defined in the generic structure. Thus it is possible to specify different presentations for chapter titles, section titles and similarly to specify, for example, titles for sections according to their level in the section hierarchy. The set of rules which specify the presentation of the elements defined in a generic structure is logically called generic presentation. A whole document model in Thot consists of both a generic structure and a generic presentation.

6.1.3 Hypertext Features

The generic structure previously mentioned is mainly hierarchical. However the Thot system offers possibilities to model non hierarchical organization through hypertext features, notably as hyperlinks and cross-references. Hypertext features emphasize more flexibility of the underlied structure; hyperlinks allow to relate freely any types of data. The reader can refer to [29] for more information about hypertext notions.

6.2 The Prototype

We have designed a well-suited document generic structure for logic programs according to the usually accepted methodology in Logic Programming. With few changes, the prototype could be applied to other programming languages as we will see in the conclusion. The associated generic presentation has been designed according to criteria relative to the nature, importance, expected position of the elements in a program document, etc. Use of the prototype allows us to assess results and to evaluate the impact and the relevance of our technique relative to the proposed methodology for logic program development.

6.2.1 Prototype Architecture

A whole program document is visualized in the integral document view. Other views may be specified in the generic presentation. Different conversion schema may be defined to export a program into different specific formalisms (*e.g.* LaTeX). Document exportation may be achieved upon views to collect information that can be used as input of various external systems such as a Prolog evaluator, a spell checker, a theorem prover system, etc. The Application Program Interfaces (API) provided by Thot allows us to develop our

specific applications that potentially could act on the editing document. The Thot toolkit is a comprehensive set of editing functions (written in C) that can be used for building the previously mentioned applications; such functions perform operations on structured documents in the UNIX X-window environment.

6.2.2 Views

A program (document) can be seen from different perspectives called views: each of them, specified in the generic presentation, is a way to visualize exclusively specific elements of the generic structure that are relevant for the programmer during a given stage of the development process. For instance, the user might want to focus on the clauses part of the program, or on the assertions or comments parts. Automatic synchronization of views allows the programmer to navigate on its document by pointing or selecting some chunks in any views. This aspect may be very relevant for large programs and facilitates "real-time" information retrieval. The user can work (write) in a specific view instead of editing the integral document since the editor itself will achieve real-time up-to-date of all the other views (as well as the integral document view). All the views can be opened simultaneously. Such features enhance flexibility and facilitate the program development process. Four kinds of views have been specified in HyperPro:

- Program view: allows to visualize exclusively the clauses parts (predicate definition) of the integral document.
- Comment view: allows to visualize exclusively the comments parts relative to the predicate definitions.
- Assertion view: allows to visualize exclusively the assertions parts relative to the predicate definitions.
- Typing view: allows to visualize exclusively the typing parts relative to the predicate definitions.

6.2.3 Document Format Conversions and Exports

Thot is an open system; this means that documents may be exchanged with other systems by means of a flexible exporting mechanism.

Specifications of document models enable Thot to produce documents in a high-level abstract form, called canonical form well-suited to handle documents. For each document, a set of translation rules can be defined, specifying how the canonical form has to be transformed into the wished specific formalism. We have defined different export schemes of our program documents: a LaTeX conversion, an ASCII conversion of the whole document or of certain specific parts as the comments part, the clauses part, the assertions part; all of them corresponding to the different views of the program document.

6.2.4 Hypertext Functionalities: links, flexibility

The program document model we defined contains relation definitions (elements of the generic structure) where the same predicate may be defined by several versions. Our prototype offers possibilities to put links in two ways: i)to point the current version of a

given predicate definition in a relation definition block; ii)to relate a use of a predicate to its whole definition, which contains all the information about the considered predicate such as comments, assertions and predicate type.

We have specified many optional elements in the generic structure and elements that can be chosen through menus. For instance, a user who wants to write a clause can choose to insert a fact, a rule or a goal, etc, and the prototype will give him automatically the corresponding template to be filled out accordingly.

6.3 Executable Documentation

In the beginning, a user might think that the writing process provided by the hypertext environment is too rigid, due to the underlying generic structure, and this may be seen as a drawback. But anyone will rapidly change his mind after more practice because this potential rigidity turns to be a great advantage of our approach: it gives orientations to respect a programming methodology during the entire writing process and allows automatic and powerful processing (views, translation,...) that can be performed on the whole document or only on specific chosen elements of the underlied generic structure.

Here we give a few examples of such operations developed with the application program interfaces (API) provided by the editor system:

- Partial tangle: the user selects a chunk of program (packet of clauses) and requires to test it. After clicking on a special menu item the considered chunk is saved on a file, the system opens a window, calls a Prolog evaluator and loads the packet of clauses file. Then the user can perform any tests he wants within the Prolog evaluator. A possibility to load automatically all the current predicate definitions that often are necessary to test a part of code is being studied.
- Global tangle: possibility to extract exclusively the referenced program part of a document in order to load it to an evaluator.

Possibilities to include some results in the program document are being studied.

7 Main Results and Prospectives

One of the main results is that our system enables logic programs writing according to a given methodology which is supported by the editor itself through consistency between a generic structure and the specific structure of an editing program document. This is a great advantage to control efficiently the different stages of the software developments and, notably, during the maintenance stage. Different presentations can be available and the user may carry out some customizations to match its specific needs or tastes.

HyperPro offers possibilities to work in different views: this facilitates the writing process because the user can focus on specific document parts that turn to be more relevant for him during a given stage of the program development.

Hypertext features as hyperlinks allows the user to follow predicate definitions and to retrieve the current version of a predicate definition. This is very useful to perform some

tests when there exists several implementation versions of the same predicate definition in the same document.

HyperPro offers possibilities to export a document into different formalisms: it allows the user to exchange or to transfer piece of documents with other systems such as LaTeX or World Wide Web for instance.

HyperPro is an interactive WYSIWYG[3] system, which is an other advantage [8] in comparison with the "classical" Web family systems described in Sections 3 and 4.

Thot is an integrated and extensible system. It allows to process with the same tool and within the same document not only structured text but also graphics, pictures, complex tables, mathematical formulas, etc. This is not an exhaustive list: users can add other types of information by specifying the appropriate models. In this sense our prototype enhances programs to be documented with hypermedia style comments. Moreover possibilities to add a complete bibliography and some annexes at the end of the programs are available.

An other advantage of the HyperPro prototype is emphasized by the fact that all the future improvements of the Thot editor system will be available in the HyperPro system without a lot of work because of the reciprocal interaction.

We claim that all those previous points are great improvements in comparison with the usual Web systems that are used in the industry. We intend to apply our experiment to imperative programming with languages such as C or Pascal.

8 Conclusion

In any area or programming paradigms, software documentation is a serious and yet not satisfactorily solved problem. Documentation has a direct impact in the cost of software maintainance. Particularly, with constraint logic programming there is a great need for a good documentation system. Due to the high level of expressiveness of CLP, programs are closer to a specification rather than to a traditional program. Usually, as a consequence to commitments to efficiency of implementations, the same CLP program is maintained along the entire life cycle of the application: from its conception to its final uses and further improvements or updating. The role of the HyperPro system is exactly to offer the facilities to maintain also the whole documentation concerning the program of an application, during its whole life. The HyperPro approach allows texts, clauses and assertions to share a single document which helps keeping consistent software documentation throughout their lifetime. Inconsistencies are not automatically checked, but we expect the fact that Prolog clauses and their corresponding explanatory texts are strongly tied together encourages maintenance people to keep software changes well documented.

Documentation of software must also be organized in a way to provide different levels of abstraction of the documented software in order to help the understanding of large systems. The synchronized viewing facilities provided by Thot, and used in the HyperPro system, permit the definition of levels of abstraction, which are the key to build large, understandable and manageable documents. Furthermore, the view windows implemented in HyperPro also accept updating operations of their contents with automatic reflection in all other view windows.

[3]WYSIWYG stands for *what you see is what you get.*

Additionally, HyperPro views program documentations as being computer processable, *compilable*, and executable, so errors in coding can be detected earlier in the design process. All error messages, including those produced by compilers, refer to the original document files, so that programmers are encouraged to do debugging or testing only on the original document files.

At the present stage of development, HyperPro does not check automatically inconsistency among clause definitions, assertions and their informal descriptions presented in the documents. A solution for this hard problem still has to be envisioned, perhaps by creating strong ties between the documentation and its related pieces of programs. Thus, the consistency between descriptive texts and the corresponding program code clauses could be always enforced somehow.

References

[1] Pierre Deransart and Gérard Ferrand, *An Operational Formal Definition of Prolog: a Specification Method and its Application* , New Generation Computing 10 (1992), 121-171, 1992.

[2] 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, September.

[3] Ciancarini, Paulo and Levi, Giorgio, *Applications of Logic Programming in Software Engineering*, University of Bologna, 1995, cianca@cs.unibo.it, 38.

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

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

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

[7] Henrard, J. and Le Charlier, B., *FOLON: an Environment for Declarative Construction of Logic Programs*, PLILP'92, Leuven, Belgium, 217–231, 1992, August 26–28.

[8] Furuta, R. Quint, V. and André, J., *Interactively Editing Structured Documents*, Electronic Publishing, 1988, 1, 1, 19–44, April.

[9] ISO, *Programming Languages - Prolog - Part 1: General core*, Information Technology, 1995, ISO/IEC 13211-1, May.

[10] Siqueira, J. de, *SEQUOIA: a theorem prover for counter model construction*, XVth conference of the Chilean Computer Science Society, Arica, Chile, 1995, August.

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

[12] Knuth, Donald D., *Literate Programming*, The Computer Journal, Vol. 27, No. 2, 1984, pp. 97-111.

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

[14] Knuth, Donald, and Levy, Silvio, *Cweb System of Structured Documentation*, Version 3.0, Addison-Wesley Publishing Company, 1994.

[15] Lamport, L., *Latex: A Document Preparation System*, Addison-Wesley Publishing Company, 1986.

[16] M. Bergère and G. Ferrand and F. Le Berre and B. Malfon and A. Tessier, *La Programmation Logique avec Contraintes Revisitée en Termes d'Arbre de Preuve et de Squelettes*, LIFO, Orléans, 1995, LIFO 96-06, February.

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

[18] Parot, P., *Construction incrémentale et modulaire de modèles de documents*, Mémoire de DEA, Université d'Orléans, 1992.

[19] Pettorossi, Alberto and Proietti, Maurizio, *Transformation of Logic Programs*, Complog II Deliverables of Year 3, D13.3.1, Apt, K.R. and Marchiori, E., CWI, Amsterdam, NL, 91, 1995, August.

[20] 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.

[21] Quint, V. and Vatton, I., *Hypertext aspects of the Grif structured editor: design and applications*, Rapports de Recherche #1734, INRIA Rocquencourt, 1992, July.

[22] Quint, V., *Les langages de Grif*, Internal report (in french), INRIA-CNRS, 1992, May.

[23] Quint, V., *The Thot user manual*, Internal report, INRIA-CNRS,
1995.

[24] Ramsey Norman, *The noweb Hacker's Guide*, Departament of Computer Science, Princeton University, September 1992 (Revised August 1994).

[25] Ramsey Norman, *Literate-Programming Tools Can be Simple and Extensible*, Departament of Computer Science, Princeton University, November 1993.

[26] Ramsey Norman, *Literate Programming Simplified*, IEEE Software, V.11(5), 97-105, September 1994.

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

[28] Richy, H., *Grif et les index électroniques*, INRIA Rocquencourt, 1992, October .

[29] Rizk, A. Streitz, N. and André, J., *Hypertext : concepts, systems and applications*, Proceedings of the European Conference on Hypertext, 1990, November, University Press.

[30] 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.

[31] Renault, Sophie and Deransart, Pierre, *Design of Redundant Formal Specifications by Logic Programming: Merging Formal Text and Good Comments* , International Journal of Software Engineering and Knowledge Engineering, vol 4, No. 3, 1994, 369–390.

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