# AALBORG UNIVERSITY

# The Vision and the Work in the HyperPro Project

by

Kasper Østerbye and Kurt Nørmark

R-93-2012 ISSN 0106-0791 April 1993

INSTITUTE FOR ELECTRONIC SYSTEMS
DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE
Fredrik Bajers Vej 7 — DK 9220 Aalborg Ø — Denmark
Tel.: +45 98 15 85 22 — Fax.: +45 98 15 81 29

# The Vision and the Work in the HyperPro Project *†

Kasper Østerbye
Kurt Nørmark
(Internet: kasper,normark@iesd.auc.dk)

**Abstract**

This paper is written to give an overview of past, ongoing and future work in the area of hypertext and programming environments at Aalborg University. In the past, the main emphasis of our work has been on data modeling and storage aspects of hypertext. Our current research focuses more directly at programming environments. Our hypothesis is that the node-link representation, inspired by hypertext, together with the basic interaction and working style known from hypertext systems can be taken as the starting point of a generic (language independent) hyperstructure programming environment. In such an environment all the important structural as well as semantical relations, at some chosen level of granularity, should be represented in a uniform way, via the link concept as known from hypertext. In the present paper we describe our visions with this kind of programming environments.

*This paper is a thoroughly updated and extended version of a paper titled "Hyperstructure Programming Environments" presented at the Nordic Workshop on Programming Environment Research, Tampere, January 8-10, 1992

†This work has been partially financed by the Danish Natural Science Research Council, Programme No. 11-0061.

# 1 Introduction and Visions

The work on hyperstructure programming environments is one of the dominant research topics in the TILDA laboratory[1] at Aalborg University. In the current section of this paper we will carefully state why we find our approach to programming environment research interesting and worthwhile. In the following section we will give an overview of earlier relevant work in our group (mostly work on hypertext databases). In section 3.2 we will outline some important properties of the prototype environment, called HyperPro, which is the testbed of our ideas.

**The administration function in the program development process:**
The program development process can be modeled as a number of activities, each of which contributes to at least one program development function [5, 35]. In this context, a "function" characterizes the purpose of the contributing activities. According to our understanding of the program development process, there are three important program development functions, namely the *construction*, the *examination*, and the *administration* functions.

The activities in the construction function all have the purpose of constructing program information from scratch or from some existing information. The program examination activities are concerned with analysis (in a broad sense) of the static as well as the dynamic program information. The activities in the administration function deal with the management of relations between pieces of program information, programmers, machines, etc.

In large scale program development processes it seems to be the case that the activities in the program administration function dominate the other activities. I.e., the difficult problem to deal with in such processes is the maintenance of relations between the pieces of program information and resources involved. What makes the program administration task especially difficult is that the network of program information is continually changing. Consequently, we find that programming environments intended to support large scale program development processes need to offer strong support of the program administration activities.

**Access paths between the pieces of software.**
In order to support the maintenance process during the lifetime of a program, it is essential that the documentation, memos, design rationales, etc. are retained. In other words, it is essential that the important decisions in the process history of the program are kept and made readily available for future maintenance work. Equally important, there should be *access paths* from program fragments to the mentioned kinds of documents (and vice versa). Having the ability to define and handle access paths in a flexible manner in between any two pieces of program information is in our opinion an essential property of a programming environment.

[1] The Laboratory for Design, Implementation, and Application of Languages and Tools in Program Development.

Access paths between program information can be handled and supported in many different ways. In the worst case there is no explicit information about access paths between program information; it is entirely up to the programmer to figure out how to come from one piece of information to another. Somewhat better, one document or program may contain textual references to other relevant documents. However, the environment cannot easily exploit such relations because the relations are not systematically represented. Furthermore, it is not easy for the environment automatically to register a relation on behalf of the programmer. In our opinion the access paths, or more generally, the mutual relations among program information, should be dealt with in a systematic way, and preferably be represented in explicit form.

**Emphasis on understanding.**
A finished piece of software indirectly reflects a substantial amount of understanding of underlying problems and solutions. We believe it is crucial that this understanding is represented and thereby maintained as an integral part of the software. We do not believe that it is feasible or realistic to collect and formulate this understanding posteriorly. One of our main theses is that the programming environment should allow for an interactive, literate program development style, in which the program is created side-by-side with its rationales.

"Literate programming" was coined by Knuth in the early eighties [25], and the most well-known support of this programming style is through his WEB system [24]. However, the elaboration of literate programming in WEB, runs counter to most contemporary ideas on interactive and incremental programming environments. In WEB, a *total* description of both program aspects and explanation aspects is written in a monolithic description, upon which a cascade of two transformations (compilations) produces the external literate program. In the HyperPro project, the internal program structure and the external program presentation should reflect the literate programming style in a more direct way for the author of the program, and not only the reader.

A key problem in the WEB system is that it only addresses the relation between the program and its detailed design rationale. How this rationale is related to analysis and high level design documents is not addressed in the literate paradigm offered by the WEB system. The WEB single file approach works reasonable well for two levels of details, the low level documentation and the program itself. It is, however, not obvious that the single file approach generalizes to multilevels, incorporating analysis and high level design considerations.

**The heritage from structure editing.**
It may be fruitful to understand a hyperstructure programming environment as an environment which applies structure editing [34, 31]. The idea of structure editing is to create and modify a piece of information in terms of the information's natural *inner structure* instead of through the elements of its *outer appearance*. The knowledge of structure may be extended to knowledge of the edited document's syntax, or even knowledge of semantical aspects.

As an example of structure editing related to program development, a program is created by manipulating the lexical or syntactical entities (such as tokens or entire declarations, statements or expressions), instead of the single letters (in the style of ordinary word processing). Another, and perhaps more well-known example of structure editing, is a drawing program that lets the user manipulate the geometrical entities of the drawing as opposed to the "pixels" in its presentation on a screen.

When structure editing is used, the editor command repertoire comes closer to the kind of data being edited. When data are specialized, the editing commands on the data become specialized too, much like specialization of classes known from object-oriented programming. Also, some kinds of errors can be prevented, if the editor's awareness of structure includes grammatical relationships. Finally, the presentation of the information on the screen during editing can be controlled by presentation schemes. A presentation scheme describes the layout of each edited data type; this allows for considerable flexibility, primarily with respect to the level of detail being presented and with respect to the actual appearance (decoration, indentation, and line breaking) of the information on the screen.

When large volumes of information are managed in a program development process, it is of special importance to allow for structure editing of the overall, coarse grained structures. (In OS-based development systems, just imagine a file system without "structure editing"). On the other hand, it is less crucial—or perhaps even awkward—if the fine grained structures such as statements, expressions or lexical tokens, are structure edited.

In HyperPro it our intention that medium to coarse grained entities are edited structurally, and leaving it open how to edit the more finely grained structures. This means that the natural abstractions, such as procedures, macros, classes, are structure edited. When natural language documents are dealt with, each (semantical) section should be considered as a structural entity, on top of which other structures may be imposed.

**Multi-user support.**
Large scale program development projects are not carried out by a single programmer. On the contrary, a team of programmers cooperate during the course of such projects. It is therefore essential to support cooperation among programmers in programming environments. As examples of the possible support of such cooperation, we can mention protection against simultaneous updating, mutual orientation on the state of the project, communication of events which are essential for other team members, and the set-up and enforcement of development plans and responsibilities of team members. From this it can be seen that programming support of cooperation deals with a host of problems ranging from simple ones to very complex undertakings.

**Evolution support.**
Large systems are composed of many different modules that evolve over time, both due to simple bug-corrections and because of change-requests from users. Different programmers might have their private sets of modules that are used only by themselves to try out new ideas. The programming environment must be able to support such

evolutionary aspects of program development. It must aid the programmers in tracing bug-reports and change requests, and relate them to the appropriate modules. The system must aid the individual programmer to trace which modules have been changed to suit personal needs, and what the original was, and what changes have been made compared to that. The programming environment must also provide tools that can aid two (or more) programmers to compare and merge their changes to the same (set of) modules.

**Summary**.
From the observations above we find it attractive to model the descriptions and the documents in a program development process as typed *nodes*, whose mutual relationships are captured as typed *links*. This is the basic inspiration from hypertext. Links make up a systematic and explicit representation of access paths between the nodes of program information. A host of administrative relationships can be captured by defining links of specialized types among the nodes. The program network, formed by the links and nodes (a graph), makes up the internal representation of a *hyperstructure programming environment*. Editing at the node and link level will be elaborated as structure editing. Node and links should be stored in a database, which makes multi-user access to the network possible, including some basic coordination, consistency, and evolution support.

The network formed by the nodes and links cannot evolve to arbitrary structures. A number of *topological constraints* will be defined on parts of the network, or if possible, more locally on node as well as on link types. Likewise, there may be a number of *contents restrictions* on the inner data of individual node (and link) types. Checking the topological structure is important, to assist the programmer in keeping track of the constraints on the software. However, not all link types need to be constrained. The unrestricted creation of different types of annotation links is a hallmark of almost any hypertext system, including hyperstructure programming environments.

In the rest of this paper we will first describe the research on hypertext databases, that has been carried out during the last few yeas in the TILDA laboratory. Following that, we describe our current and future work on a generic hyperstructure programming environment on top of a hypertext database component.

## 2 The Hyperbase Work

It has been the experience of several hypertext systems that storage issues of a hypertext system should be separated from the application and interface issues. The two main reasons for this are firstly that it provides for a better internal structuring of a hypertext system, and secondly that it makes it easier to build a system which can support multiple users at the same time. A third argument is that it is possible to develop a general storage facility which can be reused in several different hypertext front-ends.

One way of implementing a hypertext storage layer is to have a central server which provides the basic operations for building and changing a hypertext. We call such a server a *hyperbase*. By having a central server the hypertext network can easily be shared by several clients, and in that way the hyperbase server can be made the platform of a multi-person environment.

The basic services provided by the hyperbase servers, which we have developed are:

- Structural operations: Creating, changing and deleting nodes and links.
- Contents operations: Reading and writing the contents of nodes.
- Locking operations: Locking entities in order to prevent the problem of simultaneous updating from several clients.
- Event operations: The server provides operations allowing clients to subscribe to system- and user-defined events; the clients must be able to respond to events send by the server.
- Versioning operations: The server provides operations to create new versions, both revisions and variants, for nodes (including composite nodes) and links. The server provides operations to extract specific versions, as well as versions satisfying a given predicate.

Since 1988 there has been a number of projects where students together with faculty have done research on the design of different hypertext servers. The main focus of this research has been support for team environments, configuration and version control, and also run-time extensible servers have been explored.

In the next sections we will discuss some of the hypertext servers that have been constructed during these projects, and we will review the most important lessons learned from building these systems.

**Hyperbase**

One hyperbase project that took place focused on support for collaborative work, and has been reported in a couple of papers [52, 47]. The main point of interest was the use of an asynchronous message mechanism that allowed clients of the hyperbase to subscribe to events from the hyperbase. Whenever some other client caused such an event to take place, the clients subscribing to the event would be notified.

The event mechanism of Hyperbase is a synchronization mechanism that allows the client programmer to write applications that respond to actions taken by the other members of the team. In one client, EHTS [48], the event mechanism of Hyperbase was used for a number of purposes:

- Window updating. When a node is displayed on the screen, it is automatically updated when someone changes the contents in the server.

- Window removal. If someone removes a node that is displayed on the screen, it will be removed from all clients' screens.
- When a node is locked, its icon is highlighted in client browsers.
- If one tries to edit a node that is locked by someone else, it is possible to be told when the node is later unlocked, so that the editing can actually take place.

Also simple communication facilities, both asynchronous (mail) and synchronous (talk) were developed using the event mechanism. Both mail and talk allowed links to be embedded in the text that was being communicated. It is, for instance, possible to write: "Take a look [->here]", and the receiver could follow the "here" link. These examples of interface features are mentioned to illustrate the power of a hyperbase that supports broadcasting of events.

**MaTilda**

Hyperbase was later extended to support versions of nodes. The server could support both revisions and variants, and allowed different users to have their own private copies that could be evolved in several revisions independent of other users, and independent of the main versions. To test this server a multiuser front end was developed. The front end supported a rather elaborate scheme for configuration control inspired by DSEE [27], and was used extensively for emacs programming, documentation, and the development of a simple C++ program. The MaTilda project is documented in Danish only [43].

**Fenris**

The main focus of the Fenris hypertext server was on examining aspects of an open architecture server. Where Hyperbase had a fixed set of commands, Fenris could be extended on the fly. This was achieved by providing a Scheme interpreter[2] inside the server. It is possible to send definitions of new operations to Fenris, and these operations can then later be called. The vision behind this was twofold:

First, it is difficult to design a hypertext server that is both completely general and at the same time sufficiently powerful for a specific hypertext system. It was often the case that the general operations should behave just a bit differently, in order to implement the clients efficiently. One example, which we came across often, was that it was difficult to find out what kinds of information the servers should be able to provide other than simple hypertext requests. "A list of current users", "a list of all locked nodes", "a list of users that have been inactive for some days" are all examples of useful requests, which cannot easily be foreseen. A solution would be to provide a query mechanism in the server, but if the server can be extended, along the lines in Fenris, this will often be just as good, or better as the kind of information we often found to be lacking was information that did not relate to the information as much as it related to the users.

A second goal for the extensibility is to make the system more efficient. If we need to perform some operation on a number of entities we can either get all entities to the client

[2] Scheme is a Lisp dialect with static scoping and with first class procedures. Fenris as well as HyperForm (the next hyperbase to be described in this paper) use the ELK implementation of Scheme[26].

and perform the operation at the client, or we could send the operation to the server and have the server execute the operation on all the entities. If we ask the server to perform the operation we save the transportation of all the entities over the net.

The Fenris server has been documented in Danish only [3, 4].

**HyperForm**
The latest hyperbase from our group is HyperForm. HyperForm is a general object oriented platform based on the idea of extensibility. Rather than providing a specific data model, HyperForm provides a framework of simple general building blocks that through object-oriented techniques can be tailored to the specific needs of the hypertext application at hand. Hyperform has been documented in Wiil's PhD thesis [49], and in the papers [51, 50].

# 3 The Work on Programming Environments

In this section we are going to present the past and present hyperstructure programming environments, which have been constructed during the course of this project. All the environments have the status of *prototypes*. By that term we mean an environment that exhibits some essential features of a possible later "real" environment, and was made for the purpose of demonstrating an approach or an idea in a concrete elaboration.

First, we briefly describe two of our early systems, together with the most important experience that resulted from the work with them. Second, we more extensively describe our current prototype of a hyperstructure programming environment.

## 3.1 Early Prototypes

### 3.1.1 A hyperstructure programming environment for CLOS

The goals of the work on this prototype are twofold: (1) To do research on programming environment support of the object-oriented extension to Common Lisp, and (2) to gain some preliminary experience with the introduction of hypertext into the domain of programming environments. A description of the work can be found in [36].

The environment is based on a number of node types, which reflect CLOS classes, methods, and generic functions. In addition to these, an organizational node type and a node type for documentation is also supported. The link types of the environment reflect the relations among classes, methods, and generic functions together with organizational relations and cross referencing to and from documentation. Link types and node types are realized as CLOS classes.

The most important contribution to the already existing (Common/Inter) Lisp environment is four different kinds of editors and browsers: (1) Graph overview editors, (2) Link browsers, (3) Lisp structure editors, and (4) (Hyper)text editors. The two former of these are the most interesting. The graph overview editors are used to present and manipulate class hierarchies of a CLOS program. The link browsers are used to provide a palette of link presentations in a "free menu style"[3]. Link browsers are also used as natural places to initiate extensions or contractions at the node and link levels of the program representation. The two latter kinds of editors are more traditional editors augmented with link anchoring capabilities.

It is the experience that hypertext seems be a useful metaphor on which to base a structure-oriented programming environment. One may, however, expect a number of problems in relation to integrating the hyperstructure aspects with the elements of an existing, "native", and non-hyperstructured environment. It is also the experience that the two different kinds of overview editors and browsers, which are based on the hyperstructure representation, are useful tools. However, it would in general be desirable if the bulk of the work in realizing such tools could be carried out once and for all, at a language independent level. Finally, it has turned out that the persistent representation of nodes and links should not be realized in an ad hoc fashion, as in this environment. It is valuable to base the environment on a dedicated server (a hyperbase) which provide persistent storage and multi-user access to the basic entities in the hyperstructure.

### 3.1.2 A hyperstructure programming environment for Smalltalk

Along with the development of the HyperPro prototype we have developed a hyperstructure programming environment for Smalltalk. The prototype builds on the same hyperbase and utilizes the same data model as the HyperPro prototype (see below), but has a completely different user interface.

The prototype is fully operational and has been used in a number of experiments regarding literate programming using hypertext. The purpose of the experiments has been to explore to what extent the idea of literate programming can be supported using a simple hypertext interface. Whereas the HyperPro prototype is envisioned to have a rather complex user interface which allows for presenting information gathered from a number of entities in the underlying graph structure, the Smalltalk prototype uses the "one node per window" approach.

The system allows for integration of three different types of nodes: Latex nodes, Smalltalk code, and postscript figures stored on files.

Latex nodes are used for all types of documentation. The latex nodes represent information at the paragraph level and are organized hierarchically. Besides hierarchical organization, cross references are also supported. If certain restrictions are followed, the system is able to linearize the latex part, and produce a typeset version with cross reference links translated into section/page references.

[3] Free Menus is an Interlisp facility described in [15].

Smalltalk nodes are used for representing Smalltalk code. There are two basic types of Smalltalk nodes, class definitions and methods definitions. These are related using links of different types representing such information as protocol names[4] for methods, superclass or metaclass relations etc. The system supports both import and export from the Smalltalk image. It is also possible to execute Smalltalk code from special "workspace" nodes. This is useful for writing and documenting program testing.

Finally we provide some limited support for figures. Rather than integrating with existing drawing programs, or writing one ourselves, we have set the system up with a special kind of node for Postscript images. The node stores the filename and the caption of the picture, and a Smalltalk representation of the figure (screen grab from the drawing program). Browsing the hypertext will bring up the Smalltalk representation, while linearization will use the Postscript file for high quality paper output. The main problem with this approach is to keep the internal representation up to date with the actual drawing.

The results so far indicate that literate programming can be performed using hypertext alone. But the style of literate programming done differs from that originally introduced by Knuth [25] in a couple of ways. The outcome of the literate program is the hypertext itself, intended to be read off the screen, where WEB style programs are intended to be printed on paper. Another observation is that a documentation fragment often refers to more than one piece of code, unlike the WEB style literate program.

## 3.2 The HyperPro Prototype

"HyperPro" is the name of the prototype environment, which currently is under development [37]. In contrast to the environments described in section 3.1, HyperPro is a generic (language-independent) environment, which can be parameterized to support different programming languages. In this section we will first introduce the generic facilities of HyperPro. Following that, the issue of specialization and adaptation of Hyperpro to a given language or a specific need is briefly discussed. Finally, the chosen architecture of the system is outlined.

### 3.2.1 General facilities.

**The general entity types.**
At the general level of HyperPro, a very important structure is the top of the entity type tree, as shown in figure 1. An entity is an abstract concept which covers nodes and links. The common property of all entities is that they possess a set of attribute-value pairs. Nodes can be either atomic nodes or composite nodes. A composite node realizes a "grouping" of other entities. A node has a required attribute which reflects its contents. A link has two required attributes which connect it to a single source node and a single destination node. The actual connection of a link to nodes takes place via so-called

[4] In Smalltalk protocols are named groups of methods, serving as a structuring mechanism.

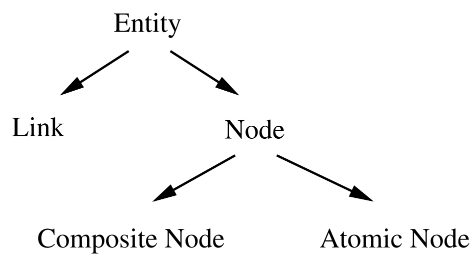



Figure 1: The top of the entity hierarchy.

*anchors*. Although links conceptually are directed, they can be used (followed) from their destination node towards their source node.

All nodetypes can be versioned, and specilized linktypes exist to support flexible linking to versioned nodes. The issues of versioning in Hypertext in general, and in HyperPro in particular are discussed in [39]

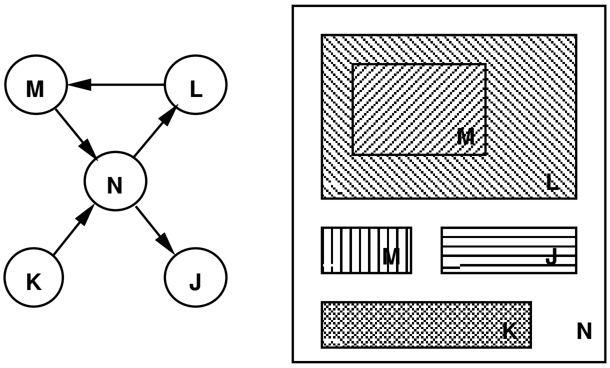
A set of node and link instances form a network, will be called the *program network* or the *hyper structure*. One of the future topics in the HyperPro project will be how to constrain this network, such that a number of overall rules are satisfied (make up an invariant). Some of these rules will reflect syntactic or semantic rules of the programming language to be supported by the programming environment. Earlier in the project, we have considered graph grammars [23] as a means to enforce constraints, but it is our judgement that the rigidity of graph grammars collide with the expected freedom in an environment based on hypertext ideas.

The current entity types are organized in a single inheritance hierarchy. When new entity types are added, we specialize an existing entity. This turns out to be a problem, as the types of nodes and in particular links are not necessarily ordered hierarchically. As an example, the current kernel types include generic version links that perform some computing to retrieve their destination node. The kernel also supports an experimental cross hyperbase link. These are both useful link types, but neither is, or should be, a specialization of the other. For many purposes it is useful to view a link as a hierarchical link or a cross reference link. This is again a type distinction that is completely independent of versioning or remote access. We need to find an appropriate notion of types for hypertext entities. Using a multiple inheritance hierarchy does not solve the problem for two reasons. First, we will see an explosive growth in the number of entity types, as all combinations of types make sense. Second, the type of a node might change dynamically. A link that points to a node must be promoted to a version link if its destination node becomes versioned.

**Interaction schemes.**
Given the situation that (1) we have defined a type hierarchy of entities, and (2) there

exists a network of persistent nodes and links instances, we can now discuss the general facilities in HyperPro for presentation and editing of program networks build from these entities.

The starting point of a presentation in HyperPro is a node. However, a presentation of a node N does not need to be limited to some display of N's attributes and the incoming and outgoing links of N. It is one of the basic premisses of HyperPro that a variety of presentation options should be provided for any given node and link. The variety of options should provide for (1) recursive presentation of nodes and links from the neighborhood of N, and (2) selective presentation of the attributes of all involved nodes and links.




(a) (b)

Figure 2: A sample network and its textual presentation.

To make this concrete, assume that the network in figure 2(a) is going to be presented in a HyperPro text editor. The starting point is the node called N. Figure 2(b) outlines one of many possible presentations of N and its context. Each rectangle in figure 2(b) symbolizes a region of text, which stems from the node indicated in the rectangle. It is possible to vary the amount of information in nested textual presentations, ranging from "the empty string", "a short button", to "the whole textual contents of a node and its context". It can be seen that node M and J (and possible contributions from the links in between these nodes and N) are presented as relatively short texts, maybe only as a button with some brief node identification. Notice that one of these nodes is attached to N via an incoming link, and that the other is attached via an outgoing link. Node L and K are presented in more details (as longer texts). In addition, node M is presented inside node L; (Thus, node M is presented twice in the window).

Every entity (node or link) in HyperPro is presented with respect to an *interaction scheme*. With respect to node and link presentation, an interaction scheme determines the level of detail of the presentation together with instructions on how to present the neighbor entities (using which interaction scheme). With respect to input events, an interaction scheme determines how to interpret the mouse and keyboard events issued on the node or link presentation. In that way, every aspect of the external appearance and every aspect of the interactive behavior can be controlled by the interaction schemes in HyperPro. Interaction schemes are organized in a hierarchy, much like the entity types. However, the interaction scheme hierarchy is allowed to have multiple predecessors.

Until now, we have only mentioned the textual interaction style. It is the experience from some of our early prototypes (see section 3.1), and the experience from much research on programming environments, that some graphical styles are well suited to give an overview of a large body of program information. In HyperPro we intend to support a graph-style presentation, in which nodes are presented as labeled vertices and links are presented as labeled edges. In addition we are considering a table style, a picture style and a browser style of interaction schemes.

The "unit of action" in the HyperPro's interaction engine is the interaction rule. The interaction rules are the properties of entity types, and they are also associated with an interaction scheme. Each interaction rule consists of an *enabling condition* and a *consequence*. An interaction rule is either a presentation rule or an event rule. A presentation rule describes the screen presentation characteristics of a node or a link of a certain type in a certain interaction scheme. Similarly, an event rule describes the action to be carried out given some input event on a node or link of the given type, which currently is presented using the given interaction scheme.

Interaction rules are located via a dynamic lookup process in the node or link hierarchy, and the hierarchy of interaction schemes. The lookup process gives an applicable rule on a given node or link instance, in either a presentation situation or an event interpretation situation. Applicability is relative to the enabling condition, which is an integral part of each interaction rule.

In most current hypertext systems, a single node can be presented as text in a single window, with attached links shown as buttons. This is satisfactory for many purposes, but there are situations were it is attractive to show several nodes and links, textually embedded into each other (as outlined in figure 2). This may be used to support literate programming in the WEB style, where a piece of documentation and a program fragment typically are presented together. It may also be used to produce a partial linearization of a program network in order to interface to existing tools such as compilers and text processors. Besides these motivations for the advanced mechanisms in the HyperPro interaction engine, we find it interesting in its own right to extend and generalize the original notion of pretty printing [38] and unparsing schemes [29] from syntax-directed editing environments to hyperstructure environments.

It is not the purpose of this paper to give more detailed information about the interaction schemes in HyperPro. One or more forthcoming papers will be centered around the declarative interaction rules of HyperPro.

### 3.2.2 Specialized facilities

In this section we will take a closer look at how the general mechanisms, which were explained in the previous section, can be tailored to support specific needs; say accommodating specific programming languages, or special documentation methods, such as literate programming or issue based design deliberation.

**The specialization process.**
To accommodate a specific programming language, we must decide how the program should be fragmented into pieces of appropriate size. This is a key problem in specializing the framework for a specific programming language. As mentioned in the introduction, we suggest the granularity of the nodes to be at the level of a procedure, a method, or a similar construct. An instance of this problem is addressed by Bigelow and Riley in [2] for the C language.

When the appropriate fragmentation level has been decided upon, specialized node types are created for each type of program fragment, and a set of link types are defined so the right program structure can be maintained.

Given these new entity types, we can design a set of interaction rules that allow us to manipulate the program network at different levels of abstraction.

**Example of the specialization process: The Smalltalk environment**
As a simple example, consider the specialization process leading from the general HyperPro system to the specialized Smalltalk environment, which we discussed in section 3.1.2. This case is particularly simple, as Smalltalk provides "fragments" which are suitable for hypertext. Two node types are needed, class definitions and method definitions. Specialized link types include *meta-class* and *super-class* which link class definitions, and *method*, that links class-definitions to method-definitions. Part of specifying new entity types is to describe new attributes. As mentioned in section 3.1.2, protocol names are stored as an attribute at the method link. It is part of the method-link specification that it has such an attribute, as well as it is specified that method-links have their source in class definitions, and their destination in method definitions.

At the interface part, we can enforce some of these constraints. Rather than having a general operation for creating a method definition node, we can specify an "add method" operation as part of the interaction schemes for handling class-definitions. This operation creates a new method-definition node, and links it to the class-definition node. In that way, methods are always created with respect to some class definition. The point is, however, that such interface decisions are made as part of the design of interaction rules, and not hard coded into the system.

While we envision the interaction schemes as controlling both the presentation of and the interaction with entities in the hyperstructure, the presentation part alone provides us with powerful declarative means for linearization of hypertext. The Latex linearization of the Smalltalk prototype is done using specialized presentation schemes that insert section headings and cross reference commands based on attributes of the nodes and links.

**Documentation support**
Besides providing support for concrete languages, such as Smalltalk, Eiffel, or Latex, the generic facilities are also envisioned to enable us to provide support for various methods.

In the WEB style of literate programming, a number of indexes and cross references play an important role. In the hypertext system described by Marcus Brown [6], different link types are used to represent these different kinds of relations. In HyperPro we can specialize link types to support the same set of relations. Brown suggests a specialized WEB editor, that can handle the three different information types present in a WEB paragraph: documentation, macros, and program text.

In HyperPro a special WEB-paragraph interaction scheme, that would extract the three parts from hyperstructure, and present them in a single window, would give the same result. However, we are faced with a problem regarding the rich set of indexes and cross-references produced by WEB. It will be almost impossible to maintain these manually, and there is currently no support for automating it. In hypertext we have the notion of computed links, but what is really needed is some support for incorporating the semantics of the languages we wish to support.

In the gIBIS system [10] the design rationale is captured through an explicit representation of an argumentation structure. Node types represent different kinds of statements (positions, issues, and arguments), and link types (supports, is-suggested-by etc.) are used to relate these kinds of statements. There are certain rules that constrain which node types can be related through which link types (e.g., an argument supports a position, not an issue).

In HyperPro the general node and link types can be specialized to provide the entity types needed. As part of the specialization we can also specify the constraints of the gIBIS system. We have initiated research on the design of a graph-oriented interaction style. It is our ambition to be able to specify graph interaction schemes that will provide us with a graph-oriented interface in the spirit of the interface provided in gIBIS (and other hypertext systems).

### 3.2.3 System architecture

The architecture of the HyperPro prototype is divided into three layers:

- The storage layer.
- The kernel layer.
- The interaction layer.

The storage layer is made up of a slightly modified Fenris hyperbase, and a client interface that provides the operations of Fenris to the kernel layer. The Fenris hyperbase is described in section 2 of this paper.

The kernel layer realizes the most important part of the system. Besides the communication between the storage layer and the interaction layer, the kernel layer controls the interaction engine, and it is responsible for the management of the topological constraints and the contents restrictions on single entities. In addition, the whole area of integration among the tools in a HyperPro environment falls within the responsibility of the kernel layer. The kernel is implemented in Smalltalk.

The interaction layer consists of a number of editors, which are suitable as front ends of HyperPro. For overview purposes we work on a graph editor, in which HyperPro nodes can be presented as vertices, and links as edges. However, the most central editor is a text editor, through which we are able to present and modify a hierarchical extract of the program network. We could have specialized the Smalltalk text editor to HyperPro purposes, but we have chosen to realize the text editor interface to HyperPro through Emacs [46, 45]. More precisely, the HyperPro text editor is an extension to the Emacs derivative called Epoch [21]. Basing it on Emacs, we hope to be able to present HyperPro as a seamless extension to the editing environment, which very many people are using already.

At the moment the main part of the data model is implemented as part of the kernel layer. This includes composites and versioned entities. This is an abnormality in the system, but something far more sophisticated than the data model of Fenris was needed, and it was decided that we did not want to build yet a hyperbase to serve our specific needs.

# 4 Similar Work

We are aware of several projects that pursue the merging of hypertext ideas and program development.

The software factory project headed by Walt Scacchi is one of the largest projects aiming at merging hypertext with software engineering, and has been widely published [20, 18, 19, 32, 33]. As the name of the project suggests, this system aims at large scale software development, and places great emphasis on project management and on clearly stating the roles of the individual participants. The project manager defines a number of different forms and basic templates, which are the types of nodes to be used in the system. The programmers then work with those types of nodes. The system is based on the Unix file system for storing contents of nodes, and linking information is kept using a relational database.

Another early system was developed at Textronix [14, 1, 2]. The system is most famous for introducing the first dedicated database for hypertext, HAM. A system for manipulating source code and software artifacts in general, Dynamic Design, was build on top of HAM.

Unfortunately the system was never used outside of Textronix[5]. In [2] Bigelow and Riley discuss the representation of C programs in hypertext in great depth. The system is also noticeable for its early work on versioning in hypertext.

HyperWeb from Vista Technologies [16] is a framework which supports the creation of hypertext-based software development environments. At the storage level, HyperWeb is based on PCTE's Object Management System, which in essence already supports a hypertext-like data model. The HyperWeb server is a Lisp based counterpart to the HyperPro kernel. The HyperWeb server emphasizes control integration among tools in the environment. Among the tools, which are clients of the HyperWeb server, are an outline web editor (a browser, which in a rather direct way supports editing of node and link attributes), an adapted version of Epoch, and the Framemaker system. HyperWeb has been used as the basis of a Unix program development environment called Udev. In Udev, files can be imported into a web, edited there, and exported back to the file system such that traditional Unix tools (such as compilers) can manipulate them. The mapping of error messages from compilers and debugger messages into the web has also been addressed.

Marcus Brown et al. propose an interactive programming environment on top of WEB [7, 8, 11]. At least conceptually, this environment is based on hypertext. There is only one node type, namely that corresponding to WEB modules. The link types reflect various cross reference relations already present in WEB. Thus, the set of link types is also fixed. Besides a specialized WEB module editor (which is central in the environment) the environment is intended to support a variety of other editors and views, some of which are non-textual.

Philip Johnson heads a group who are investigating the CSCW part of software engineering through hypertext. In [22] Johnson and Tjahjono addresses the issue of code review. Seen in relation to HyperPro, this is an interesting aspect of software engineering, the question being if we can specialize our interaction schemes so as to acommodate reviewing properly. The paper presents two interesting aspects in this regard. First, a data model for reviews, which can be seen as an elaboration of the gIBIS model [10], is presented; second, there are different phases of reviewing, one being "private review". HyperPro can support some sort of voluntarily privacy through interaction schemes that mask out annotations done by others. This is an aspect we would like to investigate further.

In addition to the work described above

- the ConversationBuilder by Simon Kaplan et al. [17],
- the Amdahl work on hypertext and CASE [12],
- the Beta/Mjølner work on hypertext in a syntax-directed programming environment [41],
- the work of Alan Dearle [13] on a hyperlinked software development environment,

[5] According to Bigelow.

- the work on design deliberation as seen in gIBIS [10], and NelleN [28]

have some similarities to the work we carry out in the HyperPro project.

Some related research projects on hypertext databases are the hypertext abstract machine [14, 9] and the work carried out by Norman Streitz et al. [42]. The Graph server at UNC [44], and the Sun link service [40] are also examples of backend support for hypertext.

# 5 Conclusion

We will here briefly describe the current state of the HyperPro system, the ongoing work, and future plans.

We have almost finished the prototype of the HyperPro environment, as described in section 3.2. What is missing is some work at the interaction layer. The kernel part is operational, and it has been extensively tested through the Smalltalk experiment mentioned in section 3.1.2. The data model provided by the kernel includes the basic entity types outlined in sec 3.2.1, as well as it supports versioning of atomic nodes and composite nodes as presented in [39].

The presentation part of the interaction schemes are operational for the textual style of interaction, and it has proven to be useful in its own right as a declarative hypertext linearization mechanism.

We are currently concentrating on finishing the textual interaction part. Next, we would like to use the generic mechanisms to specialize the system to support a compiler based language. We are currently doing concrete design work on a hyperstructure programming environment for Eiffel [30]. There are a number of interesting research issues on how to provide good integration between such a language and our hypertext system. Another important issue that we want to address is that of multiuser support.

In the future we plan to address other interaction styles than the textual one.

# References


[1] James Bigelow. Hypertext and CASE. *IEEE Software*, pages 23–27, March 1988.

[2] James Bigelow and Victor Riley. Manipulating source code in dynamic design. In *Hypertext'87 Proceedings*, pages 397–408, 1987.

[3] M. Boel, A. Gregersen, P. Larsen, and F.H. Møller. *Manual og dokumentation til FENRIS version 2.0*. Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, February 1991. In Danish.

[4] Martin Boel and Poul Larsen. Development of programming environments. Master's thesis, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, August 1991. Internal report - IR 91-08. In Danish.

[5] Karen Borup, Kurt Nørmark, and Elmer Sandvad. Ekko—an integrated program development system. Technical Report DAIMI IR-51, Computer Science Department, Aarhus University, Denmark, November 1983.

[6] M. Brown. A hypertext for literate programming. In S. G. Akl, F. Fiala, and W. W. Koczkodaj, editors, *Advances in Computing and Information, ICCI'90. LNCS 468*, pages 250–259, 1990.

[7] M. Brown. A hypertext for literate programming. In S.G. Alk, F. Fiala, and W.W. Koczkodaj, editors, *Advances in Computing and Information*, pages 250–259. Springer Verlag, Lecture Notes in Computer Science, 468, 1990.

[8] Marcus Brown and Bart Childs. An interactive programming environment for literate programming. *Structured Programming*, 11:11–25, 1990.

[9] Brad Campbell and Joseph M. Goodman. HAM: A general purpose hypertext abstract machine. *Communications of the ACM*, 31(7):856–861, July 1988.

[10] Jeff Conklin and Michael L. Begeman. gIBIS, a hypertext tool for exploratory policy discussion. *ACM Transaction on Office Information Systems*, 6(4):303–331, October 1988.

[11] David Curdes and Marcus Brown. The literate-programming paradigm. *IEEE Computer*, pages 52–61, June 1991.

[12] Jacob L. Cybulski and Karl Reed. A hypertext based software engineering environment. *IEEE Software*, 9(2):62–68, March 1992.

[13] Alan Dearle, Chris Marlin, and Philip Dart. A hyperlinked persistent software develepement environment. In *Proceedings of HyperOz'92*. University of South Australia, Level Campus, 1992.

[14] Norman Delisle and Mayer Schwartz. Neptune: a hypertext system for CAD applications. In Carlo Zanilo, editor, *Proceedings of the International Conference on Management of Data (SIGMOD'86)*, pages 132–143. ACM, 1986.

[15] Envos Corporation. *Lisp Release Notes*, September 1988. (Medley Release).

[16] James C. Ferrans et al. Hyperweb: A framework for hypermedia-based environments. In Herbert Weber, editor, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Software Engineering Notes, volume 17, number 5*, pages 1–10, December 1992.

[17] Simon M. Kaplan et al. Supporting collaborative software development with conversationbuilder. In Herbert Weber, editor, *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, Software Engineering Notes, volume 17, number 5*, pages 11–20, December 1992.

[18] P. K. Garg and W. Scacchi. Composition of hypertext nodes. In *Online Information 88. 12th International Online Information Meeting, London 6-8 December, 1988, Proceedings Volume 1*, pages 63–73. Learned Information, Oxford and New Jersey, 1988.

[19] P. K. Garg and W. Scacchi. Ishys - designing an intelligent software hypertext system. *IEEE Expert*, 4(3):52–63, Fall 1989.

[20] Pankaj K. Garg and Walt Scacchi. On designing intelligent hypertext systems for information management in software engineering. In *Hypertext'87 Proceedings*, pages 409–432. ACM, 1987.

[21] Human-Computer Interaction Laboratory, University of Illinois. *Epoch – GNU Emacs for the X Windowing System*, September 1991. System manual, also present as info pages in Emacs.

[22] Philip Johnson and Danu Tjahjono. Improving software quality through computer supported collaborative review. Obtaind from Author, January 1993.

[23] Cristopher V. Jones. An introduction to graph-based modelling systems, part i: An overview. Technical Report 89-11-03, University of Pennsylvania, Department of Decision Sciences, The Wharton School, University of Pennsylvania, Philadelphia, PA 19104, January 1990. To be published in ORSA Jurnal on Computing.

[24] Donald E. Knuth. The WEB system of structured documentation. Technical Report STAN-CS-83-980, Department of Computer Science, Stanford University, September 1983.

[25] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, May 1984.

[26] Oliver Laumann. Reference manual for the elk extension language interpreter. This document is part of the ELK system avaiable via anonymous ftp from mcsun.eu.net, May 1990.

[27] D. B. Leblang and Jr. G.D. McLean. Configuration management for large-scale software development efforts. In *GTE Workshop on software engineering environments for programming in the large*, pages 122–127, 1985.

[28] Michel Leonard and Ian Prince. NelleN: a framework for literate data modelling. In *Advanced Information System Engineering, CAiSE'92*, pages 239–256. Springer-Verlag, 1992. LNCS, vol 593.

[29] R. Medina-Mora. *Syntax-Directed Editing: Towards Integrated Programming Environments*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, March, 1982.

[30] Bertrand Meyer. *Eiffel the Language*. Prentice Hall, 1992.

[31] Steen Minor. *On Structure-Oriented Editing*. PhD thesis, Department of Computer Science, Lund University, 1990.

[32] K. Narayanaswamy and W. Scacchi. A database foundation to software system evolution. *The Journal of Systems and Software*, 7:27–49, 1987. Elsevier Science Publishing Co. Inc. 1987.

[33] K. Narayanaswamy and W. Scacchi. Maintaining configurations of evolving software systems. *IEEE Transactions on Software Engeneering*, 1987.

[34] Kurt Nørmark. Transformations and abstract presentations in a language development environment. Technical Report IN-CSLI-87-9, Stanford University, CSLI, 1987.

[35] Kurt Nørmark. Programming environments—concepts, architectures, and tools. Technical Report R-89-5, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, March 1989.

[36] Kurt Nørmark. A hyperstructure programming environment for CLOS. In Jean Bézivin and Betrand Meyer, editors, *Technology of Object-Oriented Languages and Systems – TOOLS4*, pages 127–140. Prentice Hall, 1991.

[37] Kurt Nørmark and Kasper Østerbye. The design of a hyperstructure programming environment. Working paper in the HyperPro project, 1993. Forthcoming technical report.

[38] D.C. Oppen. Prettyprinting. *ACM Transactions on Programming Languages and Systems*, 2(4):465–483, October 1980.

[39] Kasper Østerbye. Structural and cognitive problems in providing version control for hypertext. In D. Lucarella, J. Nanard, N. Nanard, and P. Paolini, editors, *ECHT'92, Proceedings of the ACM Conference on hypertext*, pages 33–42, 1992.

[40] Amy Pearl. Sun's link service: A protocol for open linking. In *Hypertext '89 Proceedings*, pages 137–146, November 1989.

[41] Elmer Sandvad. Hypertext in an object-oriented programming environment. Technical Report DAIMI-PB 280, Computer Science Department, Aarhus University, May 1989.

[42] Helge A. Schütt and Norbert A Streitz. Hyperbase: A hypermedia engine based on a relational database management system. In A. Rizk, N. Streitz, and J. André, editors, *Hypertext: Concepts, Systems and Applications*, pages 95–108. INRIA, Cambridge University Press, November 1990.

[43] Morten O. N. Simoni and Poul Sloth. Hypertext based configuration management and version control. Master's thesis, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, June 1991. Report number: IR 91-06.

[44] John B. Smith and F. Donelson Smith. ABC: A hypertext system for artifact-based collaboration. Technical Report TR-91-021, University of North Carolina at Chapel Hill, April 1991.

[45] Richard Stallman. *GNU Emacs manual*. The Free Software Fundation Inc, 1985.

[46] R.M. Stallman. Emacs: The extensible, customizable, self-documenting display editor. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[47] Uffe Kock Wiil. Using events as support for data sharing in collaborative work. In *Proceedings of the International Workshop on CSCW*, 1991. April 9-11, 1991, Berlin.

[48] Uffe Kock Wiil. Issues in the design of EHTS: A multiuser hypertext system for collaboration. In *Proceedings for the Hawaii International Conference on System Sciences (HICSS-25)*, pages 629–639, Kauai, Hawaii, January 1992.

[49] Uffe Kock Wiil. *Extensibility in open, distributed hypertext systems*. PhD thesis, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, 1993.

[50] Uffe Kock Wiil and John. J. Leggett. Hyperform: An extensible hyperbase management system. Technical Report TAMU-HRL 92-003, Department of Computer Science, Texas A&M University, 1992.

[51] Uffe Kock Wiil and John. J. Leggett. Hyperform: Using extensibility to develop dynamic, open and distributed hypertext systems. In D. Lucarella, J. Nanard, M. Nanard, and P. Paolini, editors, *Proceedings of the Fourth ACM Conference on Hypertext (ECHT '92)*, pages 251–261, 1992.

[52] Uffe Kock Wiil and Kasper Østerbye. Experiences with hyperbase - a multi-user back-end for hypertext applications with emphasis on collaboration support. Technical Report R90-38, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, October 1990.