AALBORG UNIVERSITY

The Design of a Hyperstructure Programming Environment

by

Kurt Nørmark and Kasper Østerbye

IR 93 2002 ISSN 0908-1216 July 1993

Reference information:

This technical report is available by ftp from host:iesd.auc.dk
as file: pub/reports/tech-reports/IR93-2002.ps.Z

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

The Design of a Hyperstructure Programming Environment

Kurt Nrmark

Kasper sterbye

April 27, 1995

Department of Mathematics and Computer Science
Aalborg University
Fredrik Bajers Vej 7E,
DK-9220 Aalborg, DENMARK

Phone: +45 9815 8522
Fax: +45 9815 8129
E-mail: normark@iesd.auc.dk, kasper@iesd.auc.dk

Contents

Chapter 1

Introduction

We believe that a programming environment based on an internal hyperstructure representation (a graph-like network of nodes connected with links) may enhance the quality of the environment, and ultimately alleviate a number of problems in current program development and maintenance processes. The main reason is that a programming environment based on a hyperstructure representation may provide us with explicit and uniform relations between different software documents. Such relations will make it easier to provide a systematic tool-support when handling these relations.

We see a hyperstructure environment as providing new and better possibilities in the following areas:

- *Modeling of programs in relation to documentation and in relation to the development process.* A careful choice of entity types and relations among these makes it possible to represent a total data model, which describes how program elements are related to a variety of different types of documentation. The model may be rich enough also to capture the temporal development process of program and documentation. The programming environment should have knowledge of the mentioned models such as to support one or more methods that leads to software, which are in accordance with the models that have been decided on.

- *Information retention.* During the construction of the program a lot of information about the program is generated. The information we have in mind is, for instance, documentation, design rationales, working papers, memos, and mail messages. If this information is readily available during maintenance it will be easier to understand the way the program works, and the assumptions behind the program, thus reducing the risk for making changes that break underlying assumptions.

- *Overview and navigation.* Overviews and navigational tools can be build which will work at a general level, rather than being bound to any specific type of relation. This is not plausible in a situation where the relations are represented differently.

- *Version and configuration management.* Many of the problems addressed in modern configuration management are problems of relating information across individual modules. We believe that a hyperstructure representation will make it possible to provide such relationships.

- *Maintaining consistency.* It is often the case that the same topic is addressed in several documents. When changes are made in one document, the related information should be updated too. We believe that hypertext based systems will help by providing explicit interdocumental cross referential links, which will make it easier to find other places that could be affected by a change.

- *Derived artifacts.* Most programming languages support the notion of separate compilation. A problem with most approaches is that there can only be one compiled version for each version group of a module. And most often one can not see which version the compiled version is made from. We believe that hyperstructures can help manage multiple derived artifacts.

This report describes the design and requirements for a hyperstructure programming environment. The name of the environment is *The Aalborg Hyperstructure Programming Environment*. More often, however, we will use the short name *HyperPro*. Our environment will address the issues raised above in different ways.

We have described the HyperPro vision, and the work in the project, in a separate report [18].

We are building a prototype ofHyperPro, which is the test vehicle for most of our ideas. The HyperPro prototype is based on the design decisions in this report. It is hoped that the prototype can be brought to a state during the summer of 1993, where it can be used for realistic, internal experiments with programming supported by a hyperstructure programming environment. Later on the year, or in 1994, we plan to use the prototype in "external experiments" with student groups.

We expect the following near-future outcomes of this project:

- Summer 93: An updated version of this report (the present report).
- Summer 93: A paper on the HyperPro interaction schemes, intended for a hypertext audience.
- Late 93: A paper on a case study, which explores the use of HyperPro as a hyperstructure programming environment for Eiffel.

In addition, the following documentation is planned during the course of the project:

- A paper which describes HyperPro for a programming environment audience. It would be natural to submit such a paper to the programming environment conference (SDE6), to be held i late 94.
- A final version of the design report.
- One or more papers (for appropriate workshops, conferences, and journals) which sum up the outcome and the results of the HyperPro project.

In relation to this work, five students are currently working on topics that are related to HyperPro. The work of the students is only in part related to the design of the current prototype of HyperPro. However, all the work that is carried out by the students contribute to the overall work on hyperstructure programming environments, and it may very well contribute to later prototypes of such an environment. One student group is working on a programming environment based on the traditional file representation of programs and surrounding documents, in which these files can be annotated with links, and other information. Another group is doing research in ways to describe entity types, constraints on networks, and interaction patterns which all are alternative (and maybe simpler) than the ideas reflected in this report. Finally, a group is working on models of the development process and the support of the chosen model in hyperstructure terms.

Structure of this report

First, the report contains a chapter that summaries the overall design ideas, and the direction of our work. Following that there are chapters for each of the three different layers in our design: the storage layer, the kernel layer, and the interaction layer. We start with the chapter that describes the design of the kernel layer. Following that, there is a comprehensive description of design issues relevant for the user interface of a hyperstructure programming environment. The last of the chapters is about the design of the storage layer. At the end of each chapter we have included a collection of notes. These notes contain further elaborations on some detailed design issues as well as descriptions of implementation issues.

Chapter 2

Overall Design and Architecture

In this chapter we will present the overall design decisions of the envisioned programming environment. The more detailed decisions and discussions will take place in the following chapters.

A programming environment is a collection of integrated development tools, which support program construction, examination and administration activities [13]. HyperPro will primarily make contributions to a better support of the administrative activities, but also the program construction activities will be affected. The program examination activities, however, will probably not be affected in any significant way by HyperPro.

The overall idea behind HyperPro is to structure a program as a persistent collection of nodes, connected via a number of persistent links, and to elevate this model to the user interface of the environment. This is, in general, also the idea of hypertext.

Internal representation and granularity
At the internal representation level we want to represent a program as at network of nodes and links. As a matter of terminology, this kind of representation will in the following be called a *program network*. Both nodes and links are typed entities, and these types are arranged in a type hierarchy. The question of granularity in the representation immediately becomes an important issue. At the one extreme, nodes may be atomic. This is the solution known from many syntax-directed editing environments. At the other extreme, the nodes may be entire modules and programs. This resembles the file level representation known from many text-oriented programming environments.

We do not want to take a definitive stand on the question of granularity, because we will design a generic, language independent environment (see below). However, it is our intention to support a medium level of granularity, where, for instance, procedure abstractions and similar language abstractions are the typical constructs in a node. It may be possible to configure and parameterize the generic environment to a finer granularity, but it is not likely that an extreme, such as the fine grained solution just mentioned will be supported very well by the system.

The user interface and interaction schemes
At the user interface level we want the user to feel that the structuring and referencing means of the environment are similar to those used in hypertext systems. So the way to obtain further information from a given node in focus in some window is to follow a link to another node, and the way to construct new elements of a program, or surrounding documentation, is to connect a new node via a link to the existing program network.

At the user interface level, we want to support the possibility of presenting a node with respect to given *presentation scheme*. In a fully developed version of HyperPro, there will be textual as well as number of different kinds of graphical schemes (graphs, tables, structured drawings). The presentation scheme will determine the basic style of the presentation, which of the anchored links to present in a node, and how to present them. It may well be the case that several nodes, and the attached links, are presented in one single window.

The notion of a presentation scheme is actually expanded to that of an *interaction scheme*. Besides presentation aspects, an interaction scheme also determines the style and the concrete elaboration of the user dialogue when working on the presented nodes and links in a window.

Persistence
In a programming environment it is important to support persistence of the program and information related to the program (documentation, test data, manuals etc). Usually, this has been achieved in a quite simple and obvious way, by storing the necessary, long lived information on files. In our current work, the persistence of nodes and links is achieved by the use of specially developed databases for hyperstructured information.

A number of hypertext backends have been developed [21, 2, 4, 19, 20], and a significant experience has been accumulated with the design and implementation the such databases. (See section 2 of [18] for an overview of this work). A hypertext backend database will be called a *hyperbase*. Some of the fundamental decisions about the anatomy of nodes and node attributes, links and link attributes, and link-arity are taken at this level. HyperPro will be based on an existing hyperbase, and only light modifications of the hyperbase are expected to be necessary during this project.

Collaboration
Collaboration among a number of programmers working in a team is an important objective in the design of HyperPro. Means to avoid simultaneous writing in the same node, and means to communicate events of general interest make up the basic and most elementary support of collaboration. Both locking mechanisms and event mechanisms are well-developed aspects of the hyperbases mentioned above.

Version control and configuration management
As the documents and program to be handled by the system increases in size, it soon becomes necessary for the system to be able to handle different versions of both individual nodes and subparts of the hyperstructure. Hypertext is challenging in that there is a significantly larger number of entities to keep track of, which is also a problem for the user interface to make versioning possible at all, and because the links of hypertext typically represents relations, and must behave consistently across versions. The issue of versioning within the HyperPro project is discussed in [16]. Some of the issues traditionally known as configuration management comes

up in hypertext as versioning of composites.

The administration of different versions of a piece of program (or program related information) is also a serious concern in a programming environment. Version control is planned to be taken care of at the hyperbase level. Configuration management is also a possible issue, which is natural to work with in HyperPro.

Genericity
It is a goal of the project to develop a generic programming environment, which somehow can be parameterized with information about a programming language, a documentation standard, or another formalism. The description of such languages will be structured into two layers:

1. The abstract node and link layer.
2. The interaction layer.

At the abstract node and link layer, it is defined which types of nodes and which types of links there will be supported in a "generic instantiation" of the hyperstructure programming environment. For a link type the possible source and target node types must be determined. Symmetrically for a node type, the possible types of links leaving or entering the node must be described. We may also need non-local constraint mechanisms on the topology of the typed program network. As an extreme in this direction, we could base the environment on graph grammars as in the Ipsen work [7, 10] and use a syntax-directed interaction style on the network defined by a graph. We have some experience with graph grammars [6], but we tend to prefer a simpler, and more "free style" approach, which also is more along the lines of traditional hypertext usage.

The interaction layer is supported by one or more interaction schemes of the supported programming language (see above). It is our hypothesis that some interactions can be described generically. In that way, only interaction patterns, which are specific or special for the given programming language need to be described explicitly for the language.

The means of expression at both of the layers mentioned above are mainly declarative; We express constraints and interaction schemes via declarative rules, which are attached to node and link types.

The client-server architecture
In HyperPro the program network is stored on a server. Each programmer in the programming team, which work on the program network, uses a client. The server as well as each client will typically run on workstations. There must be means on the server to coordinate simultaneous work on the network from different client machines. The lock and event mechanisms are examples of such means.

It is possible for a client to extract the information about nodes and links from the server, each time the information is needed. However, we have decided that we want to have a local representation of part of the program network on each server. Hereby the cost of communication across the network will be lowered. Locally on the server, there may or may not be a cache of nodes and links, in order to minimalize disk access.

There must exist a significant degree of consistency between the server and a client, and following that, also between two given clients. One could call for total consistency at all times, but we do not think that this is realistic.

On a client, we want to describe and manipulate nodes and links as objects in an object-oriented fashion. It is our hypothesis that the genericity aspects (both at the abstract node and link level, and at the interaction level), which we described above, can be realized by specialization of the general classes for nodes and links.

The components of the HyperPro prototype.
We use the Fenris Hyperbase [2] as the bottom layer in HyperPro. The kernel is implemented in Smalltalk. The reason is that Smalltalk provides us with an object-oriented, dynamic language, which is on the one hand is well-suited for prototyping, and on the other hand is reasonable efficient.

The textual interface of HyperPro is realized in Emacs, or more precisely in Epoch, which is a multi-window, multi-fonted Emacs, which can keep track of zones in text and which lets us use colors as a means of expression. Epoch communicates with the Smalltalk kernel via a simple protocol, which we have developed for this project. Using this approach we obtain a decoupling between the kernel and the textual interface (providing for a physically distribution of the kernel and the text editor), an integration of hypertext with a common, widespread text processing tool, and a chance to experiment with an integration of disparate tools.

Chapter 3

The Kernel Layer

The kernel layer provides a framework for defining the types of entities that the hyper document will be made from. This chapter addresses the design of the fundamental entity types like links and nodes.

3.1 Introduction

As mentioned in the in overall description, we have decided that nodes and links are organized in a type hierarchy. The properties of node and link types, and their hierarchical organization is the major topic in this chapter. We will examine what mechanisms are needed to describe the entity types, what operations we want for their manipulation, and how we can represent these properties.

Type descriptions
It has been a fundamental design decision that we organize entity types in a hierarchy which support inheritance. This is seems as an alternative to trying to come up with a totally generic data model, which has been tried out in a number of other backends [5, 17] and our own hyperbases.

There are a number of fundamental properties that must be described for each type of entity:

- Topological constraints. For a node type we want to specify constraints on the number and types of the links relating this node to others. For links, we constrain the type of the source and destination node.
- Contents restrictions. We want to be able to handle structured text within nodes (procedures, class definitions, etc). It is therefore important that we have a way to specify that structure. For composite nodes (see below) the contents restriction becomes restriction on which entities they will contain, and how these are interconnected.
- Interactions schemes. For a specific node type, we will want to be able to specify interaction schemes for that type. Interaction schemes will be discussed in chapter 4.

- Conventional operations. While the above are important properties, we also need more mundane operations to manipulate the structure and attributes of the nodes and links. Some of these operations will be specific to one entity type, while others are general in nature.

- Required attributes. When we create a new entity type we often want to ensure that some attributes are always present in such entities.

3.2 Terminology and model

Before any further elaboration on entity type specification is given, our terminology and underlying model of hypertext will be described.

Entities are organized in a type hierarchy, where each type defines a predicate:

$T(x)$ iff x is of type T

For convenience we will some times write $x \in T$ rather than $T(x)$.

Subtypes are mathematical specializations:

S subtype of $T \Rightarrow \forall x.S(x) \Rightarrow T(x)$

Attributes
An attribute is a partial mapping from entities to values. We will make no general restrictions of what type a value can be, except that entities are *not* values, but entity identifications are [7].

In order to avoid problems with partial mappings, we extend all attributes to be total mappings by introducing the special error value $\perp$.

Instead of writing the mapping application as $A(x)$ we will sometimes write $x.A$.

Required attributes
When we define a new entity type, we are often interested in specifying that an attribute is defined for all entities of this type.

A is a required attribute of entity type T if
$\forall x.x \in T \Rightarrow x.A \neq \perp$

If an attribute is required for a type T, it will be required for all subtypes of T.

In general all entities have at least the following two required attributes: *Type*, which name the entity type this entity is an example of, and *memberIn* which is a list of composite nodes (see below) in which the entity is a member.

3.2.1 Specialized entities

The general type ENTITY described above is specialized into the types LINK and NODE, which again is further specialized into COMPOSITE-NODE and ATOMIC-NODE (see fig. 3.1).

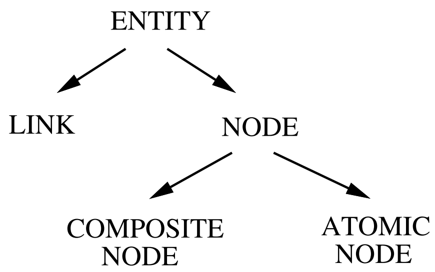


Figure 3.1: *Top of hierarchy*

Links
Links are special entity types with two required attributes *source* and *destination*.

$$\forall l \in LINK.\ source(l) \in NODE \wedge$$

$$\forall l \in LINK.\ destination(l) \in NODE$$

That is, the value of the source and the destination are nodes.

Based on these two required attributes, we define the predicates:

$$source(n, l) : l \in LINK \wedge source(l) = n$$

$$destination(n, l) : l \in LINK \wedge destination(l) = n$$

which reads "n is the source node of the link l" and "n is the destination node of the link l" respectively.

Nodes
Nodes are special entity types with a special required attribute *contents*. No assumption about the value type of contents is made.

In connection with nodes there is a set of predicates, which is derived from the predicates on links, and which is included for the sake of a more natural style of expression:

$$incoming(n, l) \stackrel{\text{def}}{\Leftrightarrow} destination(n, l)$$

$$outgoing(n, l) \stackrel{\text{def}}{\Leftrightarrow} source(n, l)$$

$$attached(n, l) : l \in LINK \wedge (source(l) = n \vee destination(l) = n)$$

We will often talk about the set of incoming links of a node n (the set of links l for which $incoming(n, l)$ is true) and similarly the set of outgoing links of a node and the set of all attached links of a node.

Anchors
In order to be able to have links that start/end within the contents of a node (which is typical in hypertext), we use the notion of anchors. An anchor A is a mapping:

$$A : (NODE, value) \rightarrow LINK$$

We make no general assumptions of the type of *value*, but typical value types of anchors are indices, regions or keywords.

The following assertion must hold for all anchors A:

$$\forall n \in NODE, l \in LINK, v.(A(n, v) = l) \Rightarrow attached(n, l)$$

This is simply the requirement that "all nodes, which are mapped to some link using an anchor, are attached to that link". Notice that the "opposite" is not true, because all links are not required to be anchored.

Composite nodes
A composite node is a specialized node with a contents with is an *ordered* list of entities[1]. Composite nodes are useful in a number of situations:

- As a grouping mechanism. To delimit the nodes and links that make up a specific project, program, paper, etc.
- As answers to queries. If one asks for all the nodes written since yesterday an answer to this query could be represented as a composite node, which contains the relevant set of nodes.
- As a representation of version groups. Rather than making versions groups something special in the system, it can be modeled as a composite node.

The following rule holds between the required attribute *members* of a composite, and the attribute *memberIn* from entity.

$$\forall c \in Composite, e \in Entity.\ e \in c.members \Leftrightarrow c \in e.memberIn$$

This is the rule which reflects the very reasonable requirement that if an entity e is a member of the composite node c then c is a member in the list of composite nodes of e.

It should be noted that composite nodes can have other composite nodes as members, that cycles are allowed, and that a composite node can even have itself as a member.

[1] Strictly speaking it is a list of entity ids.

Some further aspects of composite nodes are discussed in note 4.

Atomic nodes
Atomic nodes are nodes whose contents is not other entities, but something else like text, graphics, object code, etc.

We have thus realized and described the basic datamodel by specializing the attributes made available in ENTITY. Note 8 comments on this.

3.3 Kernel architecture

The basic design of the kernel reflects the three layers in the total architecture of HyperPro:

1. The interaction between the kernel and the storage is fully contained and restricted to entity type objects.
2. The kernel layer is where entity types are specified.
3. The interaction between the kernel and the user is governed by a number of different editor types, each of which can only handle a subset of entity types. Each type of editor requires its own interface in the kernel layer.

This account of the kernel design is illustrated in figure 3.2.

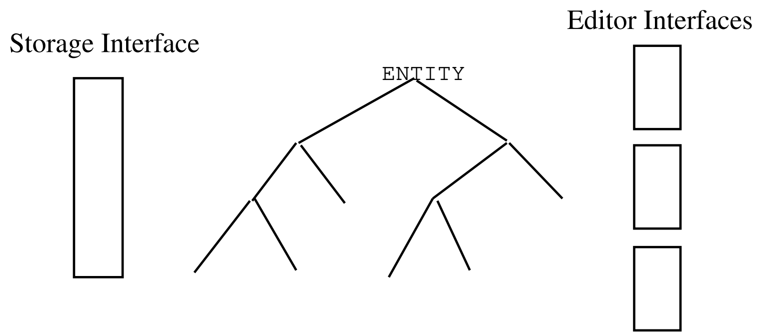


Figure 3.2: *Overview of the kernel architecture.*

Point number 1 is fully addressed in the chapter on the storage layer (see chapter 5). Pointer number 3 is, to some extent addressed in the chapter on the interaction layer (see chapter 4), but we have not yet considered any general design of the interface between the kernel and the different kinds of editors. The rest of this section is devoted to a discussion of point number 2, including design considerations of various kinds of constraints on networks.

Representation of type descriptions
There are a number of different ways one could go about representing the entity descriptions.

1. The entity descriptions could be represented and specified directly in the underlying implementation language. Advantages of this approach are that we are easily started, and we have a full language at our disposal for describing operations. If the language is object-oriented itself, we might even be able to use the inheritance mechanisms of the language directly. Disadvantages are that the syntax of the underlying language will obscure the specifications, making it more difficult to understand, and the direct coupling to the underlying language will fix the semantics of the specifications, and inhibit experimentation.

2. We could create a specialized declarative language which can be transformed into some underlying implementation language. The advantages and disadvantages are basically opposite those mentioned above. The most important disadvantage is that we must develop something close to a full object oriented language to be able to describe the operations.

3. A combination of the two previous, where some aspects of the specification is given using a declarative notation.

We would like to attempt the third solution. The idea being that we use the underlying object oriented language directly for its inheritance hierarchy and for specifying operations. For specifying constraints and interaction schemes, we will extend it with special abstractions which enables us to describe those aspects at an appropriate high level.

Let us examine the attachment constraints for node types as an example. We can make a class in the underlying language that represents such constraints, and this class can then be used when we describe constraints. The problem is then reduced to a problem of attaching a set of constraint objects to a class definition. How that is done most elegant depends on the language.

There are two interesting aspects in using the third alternative. First we can get something up and running incrementally, we need not develop a complete language and runtime environment first, and then start to write entity types. We will be able to have preliminary entity types from the beginning. Second, we can then later work on providing specialized tools that will help in developing the entity types, maybe giving some of the specifications a new surface syntax.

In the next sections we will look at how various forms of constraints can be specified.

3.4 Constraints

We have divided the kinds of constraints in which we are interested into three different different categories.

- *Attachment constraints.* We are interested in the immediate neighborhood of an entity. For nodes we want to constrain its attached links, and for links we constrain it source and destination nodes.

- *Contents constraints.* For atomic node types, we might want to constrain the type as well as the inner structure of the contents attribute. For composite nodes types we want to constrain what type of entities it can contain and how they are interconnected.

- *Required attributes.* We are interested in a way to specify required attributes.

We will now examine each of these constraints in turn, giving first a general notion, and then discussing issues in realizing the mechanisms in Smalltalk. In the current prototype of HyperPro only required attributes are dealt with.

3.4.1 Attachment constraints

Nodes
Preliminary experience [14] has indicated to us that we are typically interested in constraining the number of links of a given type and direction.

The attachment constraints[2] for a node type could look like:

nodetype constrainedAs:
| $linktype_1$ dir_1 | rel_1 $number_1$;
...
| $linktype_m$ dir_m | rel_m $number_m$

where rel_i is one of $<$, $=$, $>$, $\leq$, $\geq$, $\neq$, and *dir* is one of *in, out, any*. If *dir* is omitted it is interpreted as *out*. The notation | ... | indicates "cardinality".

The above attachment constraints have following semantics:

$$\forall n \in nodetype.$$
$$|\{l \in linktype_1 | attached(n,l) \wedge dir(n,l,dir_1)\}| rel_1 number_1 \wedge$$
$$\ldots$$
$$|\{l \in linktype_m | attached(n,l) \wedge dir(n,l,dir_m)\}| rel_m number_m$$

The notation $dir(n,l,dir)$ is a boolean function returning $incoming(n,l)$ if dir is in, $outgoing(n,l)$ if dir is out, and $attached(n,l)$ if dir is any. Thus, the cardinality of sets of links of certain types and with certain directions (relative to a given node type) must be in some relation to a given quantity.

The following is an example:

N constrainedAs:
| L in | = 1;
| K | > 0

[2] We sometimes refer to this as link constraints, because they constrain the attached links.

The meaning of this constraint is that all nodes of type N have exactly one incoming link of type L, and at least one outgoing link of type K.

The linktype and direction *selects* what links we are constraining whereas the cardinality and the relation make up the actual *restriction*. We could elaborate on the selection part where especially attribute expressions of the link could be an interesting extension. In that way we could consider having the node type "at the other end of the link" be part of the selection in some situations. As an example of that we might formulate that "there must be at most one incoming link of type LT the source of which is of node type NT." Finally, we could also consider to allow for disjunction (an "or" relationship) among clauses.

The effective set of link constraints on a node type are the union of those from its supertypes and those specified in connection with the node type itself.

Links
For link types we want to be able to constrain the source and destination node types of the link. As we operate with one-to-one links, the selection part of the constraints is easy. However, in order to obtain more control of the selection, we might want to restrict the selection to nodes with certain attributes.

For links the neighboring entities are the source and destination node, and the only constraints we want to consider are constraints on these node types.

The attachment constraint[3] for a linktype could look like:

L connects S to D

where L is a link type, and S and D are node types of the source and the destination of the link, respectively.

The semantics of this constraint expression would be the assertion

$$\forall l \in L.\ source(l) \in S \wedge destination(l) \in D$$

In connection with specialization of the link type we can introduce a new attachment constraint. We decide that we are only allowed to strengthen the attachment constraint of the specialized link type. That is, if we write

L' connects S' to D'

were L' is a subtype of L, then S' must be a subtype of S, and D' a subtype of D.

Enforcement of attachment constraints.
While we think it is important to have these constraints, and to be able to check them, we are not yet sure when we want to check them, and what actions should be taken if the constraints are not satisfied. It will be subject to experiments with the prototype to try out different strategies for this. From the outset we believe that the system should be non-intrusive. If there are some attachment constraints that are not satisfied, the user should be told, but it should be allowed to ignore it.

[3]We sometimes refer to this as a node constraint, because it constrains the attached nodes.

3.4.2 Contents constraints

Atomic nodes
It is an important property of HyperPro that the information stored in the nodes can be program fragments at some level of granularity. It would therefore be quite natural to make a specialized node type to hold, say, a Smalltalk class definition or a Pascal procedure. In both cases, there would be obvious attachment constraints (e.g., "at most one super-class", or "exactly one outer scope"). In contrast to attachment constraints, the contents restriction expresses that the text stored in the node must be follow the syntax for the language.

In order to check such contents restriction we can use a parser for the language, of which a fragment appears in the contents of a node. Alternatively, we may use syntax-directed editing technology. Using this approach, the syntactic structure of the fragment is built during the editing dialogue, thereby guaranteeing that the contents constraint is satisfied at any time during editing.

With the syntax-directed editing solution, we have introduced an *inner structure* in nodes (which is a tree structure) that should be contrasted to the primary structure we are working with in between the nodes, the hyperstructured network (which is a graph). A few additional aspects of this is discussed in note 5. The concepts of inner node structure is also discussed in the forthcoming paper on the design of a hyperstructure programming environment for Eiffel [15].

Composite nodes
One of the advantages of composite nodes is that they provide a focus for defining more advanced constraints on a hyperstructure network (such that topological constraints that are of a more global nature as the relatively simple attachment constraints described earlier in this chapter). Rather than having global topological constraints, structural constraints can be attached to composites. This is attractive for several reasons.

First, it is not necessary to recheck the entire structure after changes, but only the composite nodes that are affected. This is a very important optimization. Second, it provides a natural scope for the constraints. One can imagine that a set of entities will be constrained differently in different contexts. Third, it attaches the topological constraints to composite node types, otherwise it was unclear where such constraints would fit in a natural way.

3.4.3 Required Attributes

As we specialize entity types, there will be a number of attributes that we will expect all entities of that type to have. We want to able to ensure that a specific attribute exists, and that it has a value.

- What is needed to enforce this kind of constraints is that attributes are initialized to some meaningful value when entities are created, and that we disallow these attributes to be deleted.

- Their specification is quite simple; For each entity type we just need to enumerate a list of attribute names and their initial values.

- We must prevent that the attribute is deleted. Both at the level where it is specified in the entity hierarchy, as well as at lower (inherited) levels.

Some additional remarks can be found in note 6.

3.5 Notes

Some of the notes below deal with how we can represent various aspects in Smalltalk[4]. These ideas are very preliminary and given in itemized form.

1. **Smalltalk representation of Kernel types**
   - There will be a Smalltalk class named Entity, which will be the root of our entity type hierarchy. This class must implement basic operations to manipulate attributes (get,set, delete) and provide persistence for entities.
   - Standard Smalltalk methods will be used for operations.
   - The metaclass for Entity are likely to be changed to allow better notational support for some of the declarative aspects - especially required attributes (see later) can be given a nice notation.
2. **Smalltalk representation of constraints**
   We will first look at node constraints for links:
   - An attachment constraint for links can be represented as a Smalltalk object with two instance variables, sourceType and destType.
   - Using standard Smalltalk syntax, such an object can be created as:
     ```
     NodeConstraint new
         source: A;
         destination: B
     ```
     where A and B are entity type classes.
   - We could modify the meta class for Link to allow for such an object to be specified in connection with subclass definition.
   - The actual checking mechanism is trivial. Given a constraint object, and a link, we just need to check that the classes match, or are subclasses of the one specified by the constraint object.

   Link constraints for nodes are quite similar.
   - The two first points from above can be carried over.
     ```
     LinkConstraint new
         linkType: A;
         check: [:x| x > 7]
     ```

[4]Actually some things have been designed this way, but we have decided to move the concrete realizations and technical specifics into a document of its own

where the x will be bound to the number of links of type A. The situation is actually a bit more complicated as we need to create a list of link constraints, but it is not more complicated than can be dealt with.

- It might be a bit clumsy to include these in the subclass definition.
- Checking becomes more elaborate, but not intractable.

3. **Smalltalk representation of required attributes**

   - We can extend the subclass mechanism for Entity so that we list required attributes as part of subclass definition.
   - The delete-attribute operation will check this list.

4. **Composites and hierarchy**
   It is not quite clear what the relationship between composite nodes and a relations such as 'hierarchy' is. In a situation where a node contains the text for a chapter, there will be embedded links to each section.

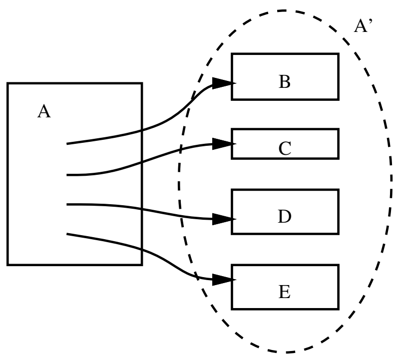


Figure 3.3: *Will a node A with links to B,C,D, and E define an implicit composite A'?*

   In figure 3.3 the problem is illustrated. It is obvious that one can, on the basis of a node with outgoing links always create a composite node by collecting the things the node points to.

   In general it will often be the case that a node in one way or the other defines a composite node. One way of dealing with this is through computed attributes. When we want to see the members, they are calculated on the spot, on the basis of the node.

   Another approach which is maybe more promising is to introduce the notion of *transient entities* in general. We envision that composite nodes, among other things, will be used to represent answers to queries. It is not clear that we want such answers to be kept around in the hyperbase. An easy solution is that transient entities are only created on the client side, and do not have a shared persistent life unless explicitly given one.

5. **Inner structure of nodes.**

   - Anchoring based on syntax rather than just regions. There are some problems in this though. If we do not support structure editing, we must re-parse the text every time it is modified - and then we loose our anchoring - how can we find the construct to which we anchored again? This is strong argument in favor of some kind of structure editing.

- If we do some analysis on identifiers, we can find out which are defined in this node and which are not. Those which are not are fine candidates as anchor points for outgoing links, and if we get further with the development of table nodes, we can hope for (semi)automatic binding to links pointing to their definition.

6. One could consider contents constraints and required attributes as two examples of a more general notion of "attribute constraints". This has some appeal, as we would then have two general kinds of constraints, topological constraints, which state things about the graph structure, and attribute constraints, that state things about the internals of entities.

7. **A note on attribute values**.
   In the actual design we do not store and treat entities as values of attributes. Rather, a new type of values is introduced: an entity identifier. This is described in great length in the chapter on the storage layer, chapter 5 in this report. A main point of the storage layer is to *pretend* that we can actually store entities as values of attributes.

8. As can be seen from the discussion of the top of our hierarchy, we base our model on the availability of attributes in the root of the hierarchy. We represents the relationship between nodes and links using specialized attributes that contains references. This has the advantage that we can describe the model in a rather self contained manner.

   An interesting alternative would be to realize the relationships between nodes and links, as well as the relationship between composites and is member entities, in a more direct manner to provide more efficiency. As it is now it takes at least 5 attribute manipulations to redirect a link: The link should be removed from the list of attached links in the previous node. This costs a read and a write. The link should be added to the list of the new node, this costs yet a read and a write, finally the link should be updated, which costs a write. It seems like there could be room for improvement, and there is really no reason to make it this complex. Perhaps a direct relational approach is actually more efficient. However, the whole issue is mostly one of efficiency.

Chapter 4

The Interaction Layer

The interaction layer is the level in the HyperPro architecture which most directly affects the user of the programming environment. Following a brief introduction we will in this chapter describe the visions and the ideas behind the HyperPro interaction facilities. Following this, some fundamental concepts and models will be presented. Then follows the design description and decisions.

4.1 Introduction

The basic problem we are faced with in this chapter is how to present and work on information, which is represented in a program network. The program network is a *structural representation*, which can be seen as a counterpart to the *textual representation* in many current programming environments.

Textual representations are created and modified via use of text editors or word processors. Such tools can be used in a variety of situations and context. The creation and modification of a structural representations is, in several respects, more challenging and more specific to a particular situation and context. We need to define a specialized editor for the chosen structure. In general, such an editor is called a *structure editor*. A main challenge in the design and realization of a structure editor is to maintain a mapping between the *internal* structural representation and the *external* presentation, as shown on the screen. It is an important invariant that the internal and external structures should be consistent with each other at any time during the interactive dialogue with the user.

Structure editors with a graphical presentation are known from many drawing programs (more specific the drawing programs, which are not pixel-manipulators). Structure editors with a textual presentation are well-known from, e.g., Lisp environments and syntax-directed editing environments. In the field of syntax-directed editors there are much experience with the definition of declarative presentation schemes [1], which can be used by a presentation sub-tool (often called unparsers or pretty-printers) to define a textual view on an abstract syntax tree. There are also some experience with the definition of presentation schemes that define graphical views [12].

In the work on HyperPro we are inspired from the work on presentation schemes in syntax-directed editors. We want to define presentation schemes for nodes and links, which determine their appearance in windows on the screen. Furthermore, we want to define the command structure and the menu setup on node and link presentations via similar means. Instead of presentation schemes we therefore concentrate on a more general concept, which we call *interaction schemes*.

We are not satisfied with the typical hypertext approach, following which one node is presented in one window, with a fixed indication of incoming and outgoing links attached to the node. In section 4.2 we describe the visions behind some much more elaborate presentations of nodes and links.

The orientation towards declarative description of interaction aspects adds to the genericity of the HyperPro environment. Recall that in HyperPro, it is possible to specialize the node and link concepts to specific needs, such as needed when we want to support a new programming language in HyperPro. Hereby it is possible to introduce new types of entities, including new types of relations, into the environment. On top of that it is natural to describe declarative rules which express

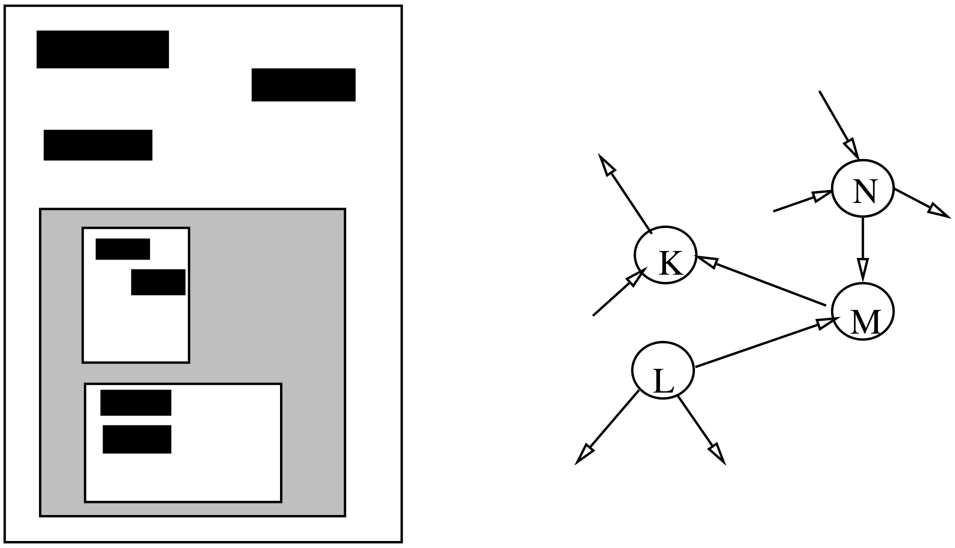
1. constraints on networks,
2. presentations of networks or single nodes, and
3. the interactive editing functionality on such presentations.

Constraints are discussed in section 3.4 of this report. The presentation and interactive editing capabilities are realized via use of interaction schemes in HyperPro. This is elaborated and described in the current chapter.

4.2 Visions

One of the overall visions of HyperPro is to bring the working style of hypertext to the user interface of the programming environment. Thus, in the programmer's conceptual understanding of the environment, the concepts of nodes, links, and anchors will play an important role. This means that constructive elements at some chosen elementary level of granularity will be directed towards creation and deletions of typed nodes and links; Navigation (which often is called browsing in programming environments) will be carried out by following links; And administrative tasks will be supported by making explicit relations among program elements via the use of links.

Textual presentation play an important role in almost any programming environment, and this is also the case in HyperPro. As mentioned in the previous section, we find it attractive to be able to present text from more than one node in a single window. This makes it possible to form a single presentation with collected information from diverse types of nodes and links. In particular, it makes it possible to present literate programs in the style of the WEB[2] [8] where both a program fragment and its explanation are presented together. In addition, it gives us more freedom to chose the level of granularity in the representation of the program network, compared to the situation where one node always is presented in one window.



(a) (b)

Figure 4.1: Nested textual presentations.

Figure 4.1(a) shows an outline of a textual presentation with nested textual presentations of other nodes.

The outer white box is a presentation of a single node N, as shown on figure 4.1(b). The three black boxes in the upper part of the figure are meant to be presentations of links ("buttons"), which leave or enter N respectively. The gray presentation is a nested presentation of a node M, to which there exist a link from N. Similarly the two inner white boxes are textual regions derived from node L and node K connected via links to M.

We also want to support presentation styles beyond the textual one. Graph-oriented, table-oriented [3], and picture-oriented [4] styles are planed in future versions of the system. In this document, some details will be given about the design of a graph-oriented presentation style. The two other styles have not yet been worked on, and they will not be described as part of the present design.

The contents of a window in HyperPro is determined and controlled by presentation rules on node and link types. These rules are, in addition, attached to an interaction scheme. Besides the window presentation issues, rules of interaction schemes interpret the events (input actions) on the contents of the window [5]. The definition of meaningful menus on presented entities is also controlled by interaction rules. It is intented that there is a coupling between the interaction schemes used during presentation and the possible events and menus on the presented contents. In the following section we define the concepts of interaction schemes more precisely.

4.3 Concepts

The definition of the following concepts is the basis of the design, which is described in the rest of this chapter.

Interaction style
An interaction style is characterized by a repertoire of graphical elements and associated interaction techniques via which to carry out actions on these elements.

Presentation rule
A *presentation rule* describes how to show a node or a link in a window on the screen (or on another output medium, such as paper). The presentation may include other entities, which are connected directly or indirectly to the node or link.

Low level event rule
A *low level event rule* describes how to map an input event (a key stroke or a mouse action) on a node or a link to a logical, *high level event* designation. [6]

High level event rule
A *high level event rule* describes how to map a high level event to an action (a procedure), the execution of which may change the underlying program network [7].

Menu rule
A *menu rule* describes a single menu item, to be presented (possibly together with other menu items) when some action is wanted on a node or a link.

Interaction rule
An *interaction rule* is either a presentation rule, an event rule (high og low level) or a menu rule.

Interaction scheme
An interaction scheme is a plan of how to carry out interaction on the set of entities in a program network. An interaction scheme is defined by a set of interaction rules. These rules—and hereby the scheme—determine "input and output" characteristics of the involved entities.

4.4 Design of the interaction layer

In this section we describe the interaction layer design of HyperPro. First we describe aspects of the design that are common for every interaction style and interaction scheme. This is the contents of section 4.4.1 and 4.4.2. The contents in these sections is rather general. In section 4.4.3 a couple of important consistency issues is discussed and decided on. In section 4.4.4 follows a more concrete description of text-oriented interaction schemes. In the current design, the text-oriented schemes are the most important ones. Graph-oriented schemes are discussed in section 4.4.5, but in less detail.

4.4.1 Interaction Styles and Organization of Interaction Schemes.

An interaction style in HyperPro has associated a unique name, and an interaction scheme has a unique name within an interaction style. [29]

Recall from the previous section, that we define an interaction scheme as a plan of how to present and interact on entities. The interaction scheme is defined by a number of presentation rules, event rules and menu rules. Each such interaction rule is classified as *belonging directly* to exactly one interaction scheme.

In HyperPro a rule of an interaction scheme belongs both to an interaction scheme and to an entity type. In a system with many different entity types, the rules of an interaction schemes may be distributed on a potentially large number of entity types. Thus, that there is no single place where all the rules, which belong to some given interaction scheme, are located or listed. We have a tool (a rule browser) in the environment which allows us to work on sets of rules which belongs to selected entity types and interaction schemes. (The rule browser is not described in further details in this report, but there exists a separate user guide which covers the details of this tool.)

Interaction schemes are, in the same way as entities, organized in inheritance hierarchies. If the interaction scheme S1 inherits from S0 it implies that S0 rules also contribute to the rule set of the S1 scheme. More precisely we say that a rule R *belongs to* S1 (1) if R belongs directly to S1 or (2) if S1 inherits (directly or indirectly) from S0 and R belongs directly to S0. (This is the well-known interpretation of inheritance as known from many object-oriented programming languages). An interaction scheme may inherit directly from two or more other interaction schemes (multiple inheritance). [19]

All interaction schemes within a given interaction style have a common root in the inheritance hierarchy of interaction schemes. For the style S, the *fundamental interaction scheme of* S is the root of all interaction schemes within the S-style. Such fundamental interaction schemes cover the rules, which serve as defaults for the given scheme.

All the fundamental interaction schemes inherit from one scheme called "Interaction Scheme" [8].

In each interaction style there is also an *empty interaction scheme* which suppress the presentation of an entity.

Figure 4.2 shows a node X, the type hierarchy of its node type, some interaction rules which are part of the node types, and a sample interaction scheme hierarchy (including fundamental interaction schemes). As can be seen, the interaction rules are part of the entity type definition (in our example, they are part of the node types), but the rules are also connected to schemes in the interaction scheme hierarchy. One can think of the rules as (name-less) abstractions, which specializes on two parameters: the entity type and the interaction scheme. [30]. Given an entity, such as node X in the figure, and an interaction scheme I, we are interested in locating "the most specific" applicable rule relative to the two hierarchies, in which the type of X and the scheme I are parts.

Within a given entity type, all rules which belong directly to a given interaction

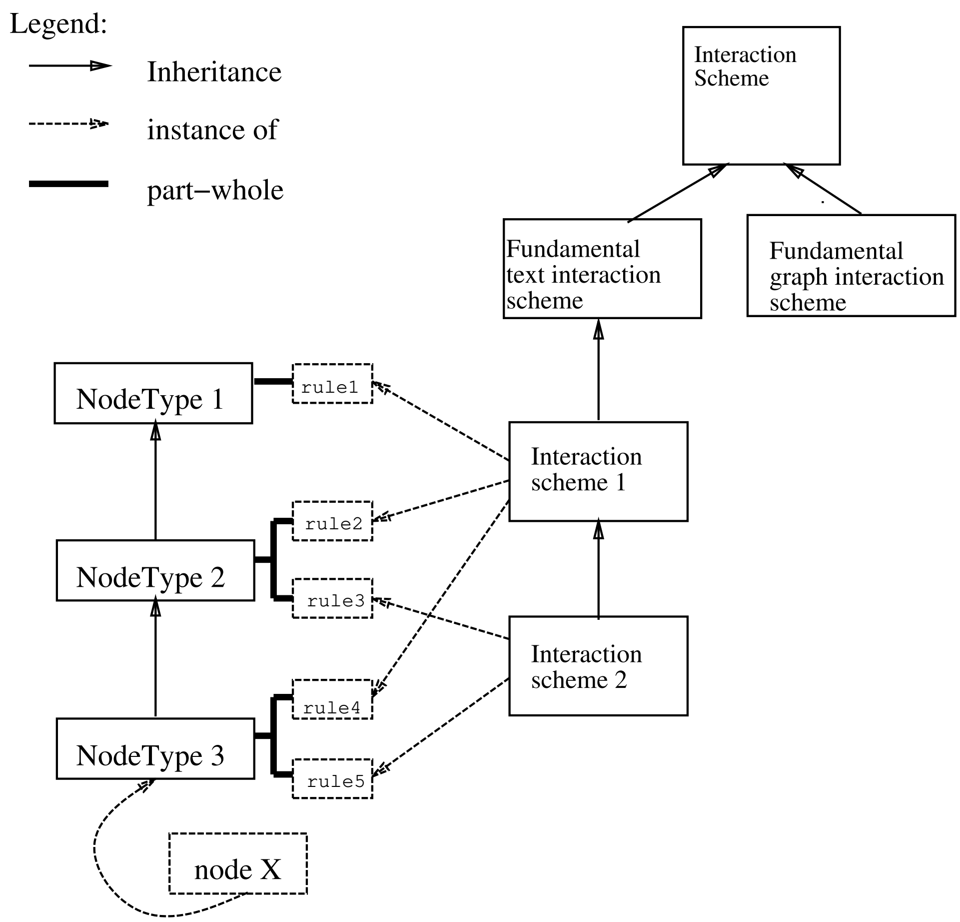


Figure 4.2: Interaction rules and their organization relative to entity types and interaction schemes.

scheme are totally ordered. This order is of significance, because applicability testing of the rules (see below) is done in that order.

The practical use of all this can be explained in this way: An entity X (a node or a link, such as node X in figure 4.2) is to be presented in the way prescribed by some interaction scheme I. In order to carry out the presentation process, the most applicable presentation rule is located given X and I. Roughly formulated, this rule tells which attributes to present in which way and how to present the context (neighboring entities) of X. The details of the rule-lookup-process is not important at this level in the discussions. In note number 31 the details of two different rule-lookup-processes are described.

A similar scenario as the one outlined for node/link presentations can be outlined when an input event is fired on a presentation of the entity E and interpreted under interaction scheme I. As an example, let us assume that we press the middle mouse button in a presentation created via use of the interaction scheme I. We furthermore assume that node X is in focus (that it is somehow selected) in the presentation. Pressing the middle mouse button causes a rule lookup on the type of X and the scheme I. We are only looking for low level event rules (the rules that map primitive keyboard events and mouse events to highlevel events). We find the most applicable rule, which we here assume will sets up a menu of possible actions on node X, when shown in scheme I. Each entry of the menu may be determined by a menu rule. Thus, starting from the type of node X and scheme I we locate all applicable menu rules; each such rule gives us a menu item; if a menu item is selected it causes a high level event to be fired, which again causes rule searching (looking for a high level event rule) on the node type of X and in the scheme I.

4.4.2 Overall Design of Interaction Rules

In this section we will describe the overall design of the various kinds of interactions rules. The design presented in this section is general in the sense that it covers all possible interaction styles. In a following section (section 4.4.4) we will confine ourselves to text-oriented interaction schemes. In that section we will be more concrete on the issues, which are introduced at the general level in the present section.

The classification of interaction rules are shown in figure 4.3. The leaves in the hierarchy are effective rules, whereas the inner rule classifications are abstract ones, of which no instances exist.

Each rule is of the form:

enabling-condition $\rightarrow$ consequence.

The overall meaning of the rule is that *if* the enabling condition is true *then* the consequence of the rule takes effect.

The enabling condition is a boolean expression which may involve the attributes of the entity, of which it is a part. The condition may also involve attributes of "neighboring entities"[35] (such as attributes of the node, which is the destination of a link), the state of input media, or more global information (such as the user

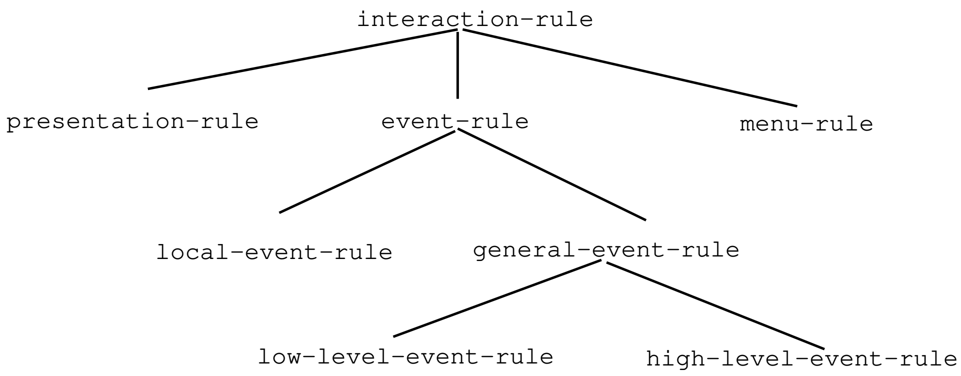


Figure 4.3: The classification of Interaction rules.

name of the current user). The contents of the enabling condition should in general ensure that the consequence part of the rule is meaningful.

We say that a rule is *applicable* if the enabling condition of the rule evaluates to true on a given entity. We have already in the previous section described the overall principles of how to locate and collect the rules which must be tried in an applicability test (and we have given the details in note number 31).

It is a requirement that an applicable rule can be located on each entity with every interaction scheme. In practice, this is not a problem, because we can define generally applicable rules on general entities related to general interaction schemes. Such rules may be thought of as default rules.

In the current prototype of HyperPro, entity types are defined via the Smalltalk system browser.[32] As already mentioned, the interaction rules are defined using a special rule browser, which is implemented for HyperPro. As of now, the interaction scheme hierarchy is defined in programmatic way (by Smalltalk expressions).

We will now describe each of the three main categories of interaction rules, as presented in figure 4.3. Recall that the description at this level is intended to be general an independent of interaction style. In section 4.4.4 we will, in a more concrete way, rephrase the design description for the textual interaction style.

Presentation Rules.
As described in section 3.2, an entity has a set of attributes (name-value pairs). The consequence part of a presentation rule applied on an entity E determines

1. The set of attributes of E, which contributes to the presentation, together with their roles in the presentation [9].

2. The use of presentation-specific elements (constants), which do not depend on the value of any attribute of the presented node.

3. The graphical appearance (size and decoration) of the presentation.

4. An indication of the protection (in the simple case, read/write allowance) of the elements in the presentation of E.

5. The composition of the individual attribute values and constant elements to the combined node presentation.
6. Which interaction scheme to use for the presentation of neighbor entities of E. [21]

If, in item 6, the entity E is a node, the "neighbor" entities are links that are anchored in the node E. If we are engaged with the presentation of a link there is only one neighbor entity, which is either the source node or the destination node of the link [33].

Event Rules.
In general, an event rule maps an input event at some abstraction level to the meaning of the event.

Let us recall the situations, where event rules come into play. We will assume that we have presented one or more nodes or links in a window, and that we now want to carry out some actions on the presentation of these, typically in order to induce some changes on the underlying program network. These actions are under control of event rules. Before an action is carried out, some node or link has to be selected in the window. The node or link, which is selected, is in the following called the *focus*. In order to apply an event rule, we need information about an entity type and an interaction scheme. The entity type comes from the type of the focus. The interaction scheme is determined as the scheme that was used for presentation of the focus. When acting on the entity in focus, we locate the most specific event rule relative to the entity type and relative to the interaction scheme. (The location of the event rules follows the exact same pattern as described for presentation rules in note 31). This event rule determines the action which should take place on the entity.

As can be seen in figure 4.3, an event rule may be either a local event rule or a general event rule. The most interesting rules are the general event rules, which come in two variants: the low level event rules and the high level event rules.

As already explained, a low level event rule maps keystrokes or mouse events to a high level event. This may be formulated in a low level event rule as follows:

> some-enabling-condition **and** guard-on-low-level-event →
> high-level-event-designation

The constituent 'some-enabling-condition' may, for instance, be some condition on the attributes of the entity in focus. The 'guard-on-low-level-event' could, for instance, formulate that "the middle button of the mouse is activated while the control button on the keyboard is held down". In case the conjunction of the conditions in the enabling condition is true, the high level event given in the consequence of the rule should be carried out.

A high level event rule determines how to map the high level event designation to an actual description of the action or procedure (or in our case, a Smalltalk message to the entity, on which we operate). The formulation of a high level event rule may be the following in HyperPro:

> some-enabling-condition **and** high-level-event-designation →

Smalltalk-message-indicator

The two level handling of events, as described above, contributes with a high degree of flexibility in the determination of the actual consequence of acting on entities presented in HyperPro. Let us give an example, where the action is “to add a note to a Pascal routine, which we will assume we have just written”. The low level event rules make us control which actual, primitive user action should initiate the dialogue that creates the note of the routine. We might chose the “C-n” (control n) keyboard event, or some special combination of mouse buttoning and keyboard modifiers. (Actually, the most realistic possibility is to activate an entry in a menu. Later in this section, we describe how that is done and controlled in HyperPro). An event rule placed on Pascal routine nodes and on a specialized Pascal interaction scheme may give us a shortcut for adding notes to Pascal routines. In contrast, we may fall back on a low level event rule defined at a general level in the entity type hierarchy and on a general interaction scheme. Following this approach we can achieve that “C-n” always is mapped to the same high level event.

The execution of a low level event rules gives us a high level event, such as 'add-note'. The high level event rules allows us to control the underlying semantics of “adding a note”. There may be a general, default interpretation, which add a standard note node via a standard non-anchored link type, and which present the note in a separate window using a general interaction scheme. Alternatively, there may be a specialized interpretation, which add a special Pascal comment note, via a specialized, anchored pascal-annotation-link type, and which present the comment in-line in the existing windows, using a Pascal-note interaction scheme.

As can be seen, there are many parameters which can be varied in the above scenarios. The HyperPro event rule framework allows us to describe simple, specialized rules, at two different abstraction levels (low and high level), and to place these rule at the entity types and interaction schemes, for which the rules are of relevance. All this need not necessarily be a burden for the user of the environment, because appropriate default event rules (both at high level and at low level) can be defined at general places in the hierarchies. However, users that whish to change the activation conditions or the semantical meaning of actions can do so by adding more specific rules to specialized entities and/or interaction schemes.

Let us finally describe the *local event rules*. An editor-local interaction rule describes that upon some input event from the mouse or the keyboard, a named procedure in one of the relevant hyperstructure editors will be run. Thus a collection of local interaction rules describes a mapping from input events to procedures, which must be defined in the hyperstructure editors. It is the intention that all possible local interaction rules should be collected and sent to the hyperstructure editor at entity-presentation time. In that way, it is *not necessary to consult the HyperPro kernel upon each local interaction*. This is the pragmatic advantage. Nevertheless, the interaction is under the control of interaction schemes. As a limitation, the changes in a local interaction procedure should only affect matters, which are presentation specific. (It is, for instance, not attractive to change the underlying structure of the program network through a local interaction rule.) The parameters of a local interaction rules are some standard ones. [23] [34] As an example of local interaction we can mention contraction and expansion (holophrasting) of presented attributes, which is an editor-local action which we, nevertheless, may want to control from the kernel.

Besides interaction, which are controlled by local or general interaction rules, there is interaction which is entirely local to a hyperstructure editor. This kind of interaction will be called *editor specific interaction.* Editor specific interaction cannot be controlled via rules in the HyperPro kernel, and it cannot affect the program network in any way. The movement of the cursor may be an example of such interaction.

Menu Rules.
Menu rules make it possible to control the contents of the menus that are brought up during a dialogue on a presentation in HyperPro. The menus, which we want to use in HyperPro, depend on the entity type in focus and the interaction scheme that was deployed during the presentation of the focus.

A menu rule may be of the following form:

some-enabling-condition → menu-description

A menu description consists of a menu item, which appears as an entry in the actual menu, a helping text which explains what happens if the entry is selected, and finally a high level event designation, which is interpreted along the lines described above.

A menu is brought up as a consequence of the execution of some low level event rule. Thus, we may in a low level event rule (conceivably placed at a general entity type and a general interaction scheme) determine that the middle mouse button activates a menu. Each applicable menu rule, relative to the type of the focus and the applied interaction rule, contribute with menu entry. [37]

It can be noticed that menu rules are quite independent of the interaction style. This is in contrast to the presentation rules, which depend heavily on the presentation style (e.g., whether text-oriented or graph oriented).

4.4.3 Consistency Issues

In this section we will describe a couple of important consistency issues.

The first consistency case
Let us assume that a node N has been presented in a window, using an interaction scheme IS. Now a number of commands are issued on the presentation, some of which add and/or delete (anchored) links of N. What does the presentation of N look like when these commands are carried out, compared with a brand new re-presentation of N

There are at least two possibilities:

1. The presentation of N is incrementally updated and kept consistent with a presentation-from-scratch during the command execution process.
2. The initial presentation of N is only a *starting point* for the interaction on the program network. The sequence of command may alter the presentation is such a way that the final presentation cannot automatically be reestablished using the interaction scheme IS.

Following the second solution, it may be attractive to allow selective user adjustment of elements of a presentation. Font changing, in line expansion of anchor presentations (as an alternative to creation of a presentation in a new window), similar contraction of nested presentations to anchor presentations are examples of such adjustments. The disadvantage of the second solution is that incremental changes of the presentation cannot be reestablished automatically, from the set of the presentation rules. The user of the environment may be forced to adapt the presentation via manual means, the obtain the desired presentation.

We decide to use the second alternative in HyperPro.

The second consistency case
Another consistency problem between presentations occurs if an entity E contributes to two or more presentations, possibly on two or more different client machines. To which degree should all affected presentations be updated when an attribute of E is changed?

Again there are several possibilities:

1. All presentations are kept consistent at any time during the interaction with the environment.
2. If a presentation P1 is affected by a change carried out via another presentation P2, then P1 is *invalidated*. It is now up to the user to ask for a regeneration of P1 (using the relevant interaction scheme).
3. We accept inconsistencies among presentations.

The third alternative is not acceptable in a multi-user environment. The first solution is probably too ambitious, at least at the current time in the research and design process. We chose the second solution as a compromise between the two extremes.

A connection between the two consistency cases
There is a connection between the first and the second consistency case, which is worth noticing. Assume that USER1 is editing (writing in) node N and that USER2 is reading node N. On USER2's screen, there are some nested presentations inside the presentation of N. USER2 changes dynamically these subpresentations, while USER1 is editing node N. Now node N is updated, because USER1 is done with his corrections. It is difficult to update USER2's presentation of N, such that the nested presentations are inserted correctly relative to the new contents of N. [24] It may complicate the matters, if the nested presentations can be changed dynamically, as discussed above under the first consistency case. *There are some problems here, which we need to address. We probably need to carry out some experiments with this, to find a good solution.*

4.4.4 The text-oriented Interaction Style.

In this section we will describe the concrete design of the capabilities of the text-oriented interaction style in HyperPro. Compared with section 4.4.2, this section is written with specific mention to textual interaction in a hyperstructure text editor.

There may, however, be some overlap of the contents of this section and the contents of the more general and more overall description, as found in section 4.4.2.

The overall working of textual presentation rules.
We decide that the starting point of a textual presentation always is a node, say N. Thus, the outer entity in a nested textual presentation is always a node. The node N is shown using a given interaction scheme, say IS (see below for details). Each link to and from N, which is anchored in the 'contents' attribute of N, is presented. This may result in a nested block of text in N's presentation, either derived from the link L or from the destination node (in case L is outgoing from N) or the source node (in case L is incoming to N) of L. The presentation of the link (including "the node in the other end of the link") may be empty.

In more details, for each such anchored link (L) to or from N we look for an applicable link presentation rule on L. As part of the node presentation rule, it is formulated which interaction scheme to use on attached links of a given type.

It may be the case that a link (L, say from N to a node M) doesn't contribute with any text. However, the node M contributes with a presentation nested textually into the presentation of N, at the anchor point of L. The presentation of M may be lengthy, or it may be reduced to some simple "button", which can be used as a starting point for a more comprehensive presentation of M. In this situation we augment the presentation of M with the information 'via-link: L'. In that way we are in a position where the link underlying the presentation of M, can be manipulated from a command issued on M. [36]

Alternatively, it may be the case that a link contributes with some text, without presenting its destination (or source) node.[25] You can think of the link presentation as a simple "button" (along the lines described above). Such a link presentation may be "followed", i.e., used as the starting point for further exploration of the program network. More specifically, it is typically possible to initiate new presentations of the source (or destination) node of such a link; It may also be possible to in line expand the link to show its destination node.[26] In any case, however, the exact interaction possibilities are determined of the general as well of the more specific interaction rules. As of now, we have no interaction functionality in mind beside that defined by interaction rules.

As yet another alternative, both the link and the source (destination) node may contribute with text. In that way it may possible in a very direct fashion to work on both the link and its attached node during the interaction with HyperPro. However, this possibility may be confusing, because the user in his or her conceptual understanding need not necessarily distinguish between the link presentation and the node presentation.

A node may contain anchors that map anchor values to links. In turn, links point at nodes, which contain new anchors etc. This defines a circularity between node types and link types via anchors. At the entity instance level we may experience real cycles in the network. In a presentation rule of a node it is specified which interaction scheme to use for the presentation of attached links; And similarly, in a presentation rule of a link it is specified which interaction scheme to use for the presentation of "the node in the other end of the link". During the definition of the presentation rules care must be taken not to define "circularities" that cause infinite presentations. Similarly, we usually want to be careful not to end up with very long presentations, in which every accessible node in a (non-cyclic) program network actually is presented (very long presentations are probably not good for interactive use in the hyperstructure programming environment; such presentations may, however, be input to compilers or similar tools, which expect monolithic input). In practice this implies, that following the presentation of a couple of entity levels relative to the starting point of the presentation, we should apply "thin" presentation schemes (schemes that produce "buttons", or the empty scheme) which cut off further presentation. This could be handled automatically, but we do not believe that this gives us very useful presentations. It is our intention to design the presentation rules "manually" such that the above mentioned qualities are obtained.

We will now turn to a semantic description of first presentations rules, and next event rule in the textual interaction style.

Node presentation rules.
As described in section 3.2, each node N has a required attribute called 'contents'. The attribute called 'contents' as well as other text attributes of N may contribute to a presentation of N. Also, text constants and text literals may contribute. The *sequencing* of such attributes and pieces of text defines the textual composition of the node presentation. Each attribute and text piece may be *indented* relative to the surrounding presentation element. *Justification* to the left, to the right, or centered is planned, but not implemented (and it may be difficult to implement in Epoch). *Font size* and *font face* can be defined for each attribute, but it cannot be varied within an attribute [11]. The color of the text (the color of the normally black pixels of letters) as well as the background color of letters may also be controlled at the attribute level. *Editing permissions* can be given for each of the presented attribute and piece of text. (This includes "read-only" and "write-permission") Finally, the text editing mode (ala Emacs) may be given for each attribute and text piece.

Link presentation rules.
Link presentation rules are quite similar to node presentation rules. Thus, attributes, text constants, and text literals may be presented; And decorations (fonting and coloring) and permissions of these presentations may be specified. In node presentations the description of how to present anchored links is a dominating concern. The similar concern with regard to link presentations is how to present "the node in the other end of the link" (relative to the node, in which the current link presentation was originated). Thus, in a link presentation rule it can be specified which interaction scheme to use for the presentation of the destination (source) node of the link in question.

Node event rules.
Recall from section 4.4.2 that node event rules are relevant when the *focus* of the interaction is on the contents of a node. We have already in section 4.4.2 described node event rules at the general, interaction-style independent level. In that section, we introduced high level event rules, low level event rules, as well as local interaction rules.

It is the intention that every command on nodes and their presentation must be defined as event rules on node entities. The only exception is the trivial text editing commands (like cursor movements and the like), which are entirely local to the text editor, at not under control of interaction rules (editor specific interaction).

The following kinds of command must be realized via command interaction rules on nodes, and associated with the fundmental text interaction scheme:

1. Selection of a new *focus* (local interaction).
   We use the left mouse button (with no modifiers) for this purpose. Thus, there will be an event rule on 'node' and the 'fundamental text interaction scheme' which simply sets the focus. The focus is a piece of state in the editor, and not the kernel. [27]

2. Making a new (anchored) link from the node (general interaction).
   As a variation, which is also very useful, a command which makes a new node as well as a link, should also be accounted for. [28] This command is a candidate for specialization in nodes, where we have more knowledge about the role of the nodes. Typically, the adding of a node and a link will be reformulated to one or more specific commands, which reflect the adding of constituents (or otherwise related entities) to the current node.

3. Adding/deleting just anchors (general interaction).
   These are command with no underlying manipulations of links. Deleting an anchor implies that we immediately get rid of any contribution from the link, to which the anchor leads. Adding an anchor implies that the link will be subject to presentation via the interaction scheme given in the node presentation rule.

4. Re-presentation in another interaction scheme (general interaction).
   Show a new presentation of the node, in another presentation scheme, possibly in another presentation style.

5. Save all the entities which have been changed in the current window.
   It may be attractive not to issue explicit saving events. Following this approach, saving is implicitly activated, upon certain well-defined events on the windows (such as window closing, and/or focus shift to another window).

The list of general functionality just listed is probably not exhaustive. More will be added during the process where we experiment with the prototype on some real examples.

Link event rules.
A link event rule is activated when a link presentation is in focus. As for nodes, it should be possible to select a presented link.

The following kinds of command must be realized via command interaction rules on 'link', and associated with the fundmental text interaction scheme:

1. Follow the link.
   This causes the target (source) node of the link to be presented. There are at least two variations of this command: The node can be in line presented in the current window; Or the node can be presented in a new window. In both cases it needs to be determined which interaction scheme to use.

2. Deleting the (anchored) link from the node.

3. Redirecting the link to point to another node.

In some situations, it may be difficult or impossible to select a link. Consequently, it may also be difficult or impossible to operate on it. The reasons is that the link is only presented very briefly (with one or a few characters) or not at all. It is a typical link presentation rule just to present the destination node of the link in the same scheme as use for presentation of the link itself. It may be attractive to be able to direct an event issued on a node to the "connecting link", which exists for all by the "outer" node of a textual presentation.

All of mentioned link event rules are general interactions. As for node event rules, other kinds of commands must be anticipated when we get more experience from experiments.

4.4.5 The graph-oriented interaction Style

A graph presentation shows some selected nodes and links in the program network as a graph. Thus, in a graph presentation underlying nodes and links are shown as nodes and links in windows on the screen.

There are a number of issues to discuss (and decide upon) in connection with graph interaction schemes in HyperPro:

1. Given some focus (such as a node) and a graph presentation scheme, which entities of the program network should be shown as part of the presentation of the graph?
2. Given a set of links and nodes (as the result of a solution to the item above), how should the layout of the graph (roughly the positions of the nodes) be determined?
3. Which attributes of entities should be used for node and link labels?
4. How should nodes be drawn in the graph presentation (size, form, placement of label(s), and other graphical details)? Similarly, how should links be drawn, and where should the presentation of a link be connected to the presentation of a node?
5. What are the possible commands on graph presentations?

The rest of this section gives answers to these questions.

The role of composite nodes in graph presentations.
A graph-oriented presentation in HyperPro is a presentation of a composite node, which holds references to exactly those entities (nodes and links) which should be presented in the graph. In the following, these entities will be called a *subgraph*. It is currently outside the scope of the graph presentation mechanisms to discuss how to define a subgraph of a composite node.

The presentation rule of a composite node, which defines a subgraph, determines the interaction scheme to be used for the presentation of the nodes and the links of the graph. To the degree that overall characteristics of the graph presentation should be determined, these characteristics are also determined by the presentation rules of the composite node.

Node and link presentation.
Each node and link of a subgraph is presented via use of a graph presentation rule. Given an entity and the graph interaction scheme, an applicable rule is located.

For nodes, the rule determines the shape of the node border, the text to be the node label (in the simple case equal to a single attribute, but maybe produced by application of a textual presentation scheme on the node), font family, font size, and coloring of the label.

A link is drawn as a straight line from its source node to its destination node. The graph presentation rule of a link determines the graphical characteristics of the line (thickness, dashing), and how to present the orientation of link (drawing of the arrow). Furthermore, the rule determines the label to be attached to the link, and the font family, size, and color to be used when printing the label.

In a future, more detailed design stage, we should discuss whether we want some kind of anchoring of links to and from composite nodes in participating nodes or links. There is also a possibility of having graphs of graphs, and similar presentations of these.

Commands on graph presentations
Graph interaction consists of individual interactions on the nodes and the links of the presented subgraph.

The following kinds of commands must be supported on presented nodes and links of a subgraph:

1. *Textual editing of node labels and link labels.*
   The label of the entity in focus is edited. Preferably, this editing should be done directly on the respective labels.

2. *Selection of a new focus.*
   Both nodes and links in a graph presentation can be made focus of the interaction.

3. *Commands on the underlying program network.*
   When adding a node or a link, it should be an option whether the new entity should be part of the composite node, which represent the presented subgraph. Similarly, when deleting a node or a link, it should be an option whether to delete the entity from the program network, or only to delete the entity from the composite node [17].

4. *Commands to initiate new presentations.*
   It should be possible to ask for presentations of meaningful entities using another presentation scheme (possibly in another presentation style).

5. *Commands to change the appearance of the graph.*
   It should be possible to change the positions of the presented nodes [18].

Each command (in the kinds of commands mentioned above) are defined and controlled by event rules, which belong to graph-oriented interaction schemes.

In the current prototype, the graph-oriented interaction schemes are not supported.

4.5 Notes

There are some notes, which are obsolete in the sense that they are not referred to any more from the main text. The remaining of these are retained in order to remember earlier, but rejected design ideas and issues.

Please notice that a few of the notes contains references to working notes, which are not available in general to readers outside the project group. We have found it valuable to retain such reference for internal use.

1. The term "presentation scheme" originates indirectly from the Gandalf work on syntax-directed editing and programming environments. In [11] the term "unparsing scheme" is used. In our opinion, the activity of "unparsing" is not very natural, because it makes us concentrate on the opposite of some process, which is not directly relevant for our discussion. Instead, we use the term "presentation". The use of the term "scheme" signals a systematic and organized framework (cf. Webster's Ninth New Collegiate Dictionary, or a similar dictionary), in our case for the mapping of the internal structures to external presentation based on rules of declarative nature.

2. In WEB, pieces of documentation and (often incomplete) program fragments are presented in an enumerated sequence of items. These items are cross referenced, and indexed. The idea of literate programming is to make program descriptions that can be read, understood and appreciated by people, for instance students. WEB supports a "batch" working style. Descriptions of the literate program is via use of one preprocessor prepared for pretty printing, and via use of another preprocessor prepared for compilation and execution. Literate programming in HyperPro will be much more interactive; The programmer is at any time during the program development process looking at and working on the current appearance of the literate program.

   Decorative as well as notational program elements are important in WEB. Systematic use of fonting, pretty printing, and macros are used. At present, it is not clear how far we can go in that respect in our interactive setting.

3. A two dimensional table is the fundamental structure underlying interaction following the *table style*. The table style may be a useful alternative to the graph style for overview purposes, especially in the cases where the presented information can be regularly classified with respect to (at least) two criteria. The table interaction style may furthermore be used as the basis for a spreadsheet-like interaction.

4. The *picture style* interaction may contain structured drawings, such as a composition of geometrical figures. Such figures may be mixed with pieces of text, graphs, tables or, more generally, nested pictures. Structured drawings are well-known from a variety of interactive drawing tools (including hypertext systems such a HyperCard) on workstations and PCs. It is our hypothesis that structured drawings may serve well in a hyperstructure programming environment in situations where various sketches need to be connected with information presented in one of the more common styles described above.

5. The part of the interaction schemes which interprets the input actions from the keyboard and the mouse corresponds to some degree to the the keyboard mappings, which are part of modes in Emacs.

6. Currently, we use a shortcut, which allows us to map a low level event directly to a description of an action (a procedure), which changes the underlying network. Such a shortcut may, in general, be useful, because it may free from specifying one level of rules (the high level rules). The high level event rules came into the design because of our support of menus, governed by menu rules. Pressing the mouse on a menu entry will cause a high level event to be interpreted. In case a high level event already exists, it is handy to be able to map from low level events to high level events.

7. The purpose of some commands may be to change some properties of a presentation or its window, independent of the underlying hyperstructure. Such commands are only of secondary interest to us.

8. Conceptually, it is nice to have one, single root (*interaction-scheme*) in the interaction scheme hierarchy. This scheme may be considered as an abstract scheme with no real contents.

9. In some presentation styles, it may be natural to emphasize some attributes in comparison with others. If the attributes are pieces of text, one attribute may be the primary one, in which links can be anchored. It may be adequate to let the presentation scheme determine the name of the primary attribute. Secondary attributes contribute in a more simple way to the presentation. Some attributes may serve the role as formatting information for other attributes. Other attributes may serve as anchor definition tables for some given attributes.

10. It will be the case that the rules for determining the focus is defined by event rules.

11. We may want to supporting multiple fonting in the 'contents' attribute. This can, for instance be realized through a decoration attribute. In order to support this, we have to keep track of regions of text during editing. However, we have to do this anyway, in order to administrate nested textual presentations. There will only have to be two kinds of "text blocks": those which represent attributes from entities, and those which represent subtexts displayed in a particular font. This discussion is also related to support of an *inner structure* of entities.

12. It would be a minor extension to allow links to contribute to a presentation *around* the presentation of the destination node of the link.

13. It is natural to use cover names for both anchored incoming and anchored outgoing links.

14. It is most natural to use nested node presentations for nodes connected to the original node via an *outgoing* link.

15. In some situations it may be convenient to produce cover names by using other textual presentation schemes on nodes. This is perfectly possible if we somehow can separate (small) nested, textual node presentations from (long) cover names.

16. HyperPro must make use of a consistent mouse button policy. We can probably use a convention of from a programming environment (such as Smalltalk) or one from a window system.

17. The variation of the adding and deletion commands on entities in a graph presentation can, e.g., be accomplished by using a modifier key (shift, control) on a single, menu-driven add/delete command.

18. The positions (x-y coordinate) of each node of a composite node, which represent a subgraph, must be remembered. The natural place to store this information is in an attribute of the composite node. In that way, a node may be part of several graphs, and have different positions in each of these graphs.

19. An interaction scheme hierarchy does not contain much functionality in terms of operations. The rules are defined within entities, and with proper refence to an interaction scheme. Actually, an interaction scheme hierarchy may simply be thought of a directed, acyclic graph. Because an interaction scheme can inherit from more than one interaction scheme, we cannot in a language like plain Smalltalk80 define interaction schemes as classes. However, we can define a special data structure, which make up this hierarchy together with operations which supports exploration of the subtype and super type relations among schemes. This has actually been done.

20. In an earlier design, we also worked with 'anchor presentation rules'. This is not an issue any more.

21. Assume that we are in the process of presenting node N using interaction scheme IS1. In the simples case, a node presentation scheme states that all links to or from the presentation of N should be presented using some interaction scheme IS2. (In an even simpler case, IS2 = IS1, but this may cause infinite presentations). It is probably necessary to provide for individual interaction schemes on the attached links, based on for instance the types of the links. In/outgoing links may also be treated differently.

22. Given two links L1 and L2 of the same type. Are L1 and L2 always presented in the same way? No, not necessarily. It may be possible in the enabling condition of a link presentation rules to test on the source (destination) node of the link, and hereby provide for differences in the presentations of L1 and L2.

23. There exists a Danish working note on local interaction rules, which contains some more details. The name of the note is local-interaktion.text.

24. For more details on this, please see the Danish working note called re-presentation.tex.

25. Earlier in the design process of HyperPro, we operated with anchor presentations. Some anchor presentations were just "cover names" of links. A cover name was generated without use of presentation schemes on links or nodes (it was generated by an anchor presentation rule).

    In order to remember our earlier design considerations, this is the text on anchor presentation:

    (beginning of obsolete text:)
    A presentation of an anchor can give one of three possible results:

    (a) Nothing (filtering out the link). Such a presentation can never be focussed on.

    (b) A (presumably) short *cover name*, which is generated without application of any presentation scheme on any node or link. Such a presentation can be in focus, and it is being considered as an anchor presentation [13] [15].

(c) A nested, textual node presentation (obtained via a trivial presentation of the intermediate link). Such a presentation is obtained via use of a designated presentation scheme on another node [14]. A nested node presentation can be in focus.

For node and link presentation with interaction scheme S, it is an error (all though a non-fatal one) if no applicable presentation rule exist. As mentioned earlier, it is not an error if no applicable anchor presentation rule can be located. Rather, it implies filtering of some links to/from the node.
(end obsolete text).

26. It is a matter of detailed design, whether in line expanding of link presentation should be supported. Such dynamic expanding may cause some problems with later updating of views on shared portions of the program network. See also the note number 24.

27. The focus is a piece of state in the editor, and not in the kernel. When a general command is activated (a command which needs to be carried out in the kernel), some standard informations are transferred from the editor to the kernel. These are: the entity number, the attribute name (if it makes sense), the interaction scheme, and the keyboard/mouse transition. All but the last of these informations are stored as properties on text blocks (on zone properties in Epoch), and they are defined during the creation of the presentation. Thus, it is enough to have knowledge of the **focus** in the editor, because other commands implicitly will transfer the focus information to the kernel, when a general command is issued. Consequently, the event which change the editor focus may be local to the editor.

28. See the working note 'kernel-command-rules.tex' for an example of the steps involved in the realization of the command "make-new-link".

29. In the current prototype, all interaction scheme names need to be unique. Thus, within the text-oriented and the graph-oriented style, no pair of interaction schemes are allowed to have the same name.

30. A note for readers that have knowledge of CLOS (the Common Lisp Object System): The double specialization of rules is quite similar to multi-methods in CLOS, and the process of looking up a rule may be seen as an analogue action to the determination of the sorted sequence of applicable methods in CLOS, given some concrete arguments. In CLOS, the first parameter position is more significant than the following parameter positions. In HyperPro, it is not clear which of the "entity type" and the "interaction scheme" should have precedence over the other. See note 31.

    There are other analogies (and differences) between the HyperPro rule framework and CLOS methods. Rules in HyperPro are name-less, and they always have two implicit parameters: the entity type and the interaction scheme. There are several kinds of rules in the HyperPro framework: presentation rules, two kinds of event rules, and menu rules. This may be compared to the roles of methods in CLOS (primary, before, after, and around), although this analogy should not be taken too far. In CLOS, declarative method combination is an important issues. We have just started to explore the similar possibilities in HyperPro.

    The similarities noticed above are not a result of coincidences. In the design process, which have resulted in the current HyperPro interaction engine, one of the important inspirations were indeed the method lookup techniques in CLOS.

31. There are basically two different ways to lookup a rule given an entity E, and an interaction scheme I. Please refer to figure 4.2 on page 25 during this discussion.

    We will here assume that the result of the lookup process is an ordered list of rules, with the most specific one first in the sequence.

    The interaction scheme dominated approach: The interaction rules in the entity type ET of E are consulted. All the rules, which belong directly to interaction scheme I is tested for applicability in the order, in which they are defined in ET. The applicable rules are collected. In the same way, the ET rules which belong to each of the super schemes of I is consulted and collected. The immediate super schemes of I are examined in an order from left to right, as found in the definition of I. (Readers that have knowledge of the Common Lisp Object System (CLOS) [1] will recognize that we are inspired by the CLOS technique for resolution of ambiguities implied by multiple inheritance.) Finally, the whole collection process is repeated on each entity type in the chain of super entities of ET.

    Relative to figure 4.2 the sequence of rules 5, 4, 3, 2, and 1 will be the result when 'node X' is presented using 'Interaction Scheme 2' (this is based on the assumption that all rules are applicable).

    This is the currently implemented approach.

    The entity type dominated approach: Again, the interaction rules belonging to the scheme I of the entity type ET are consulted. But instead of considering rules of ET belonging to one of the super schemes of I, we consult rules of the super entity type of ET *in the scheme I*. Hereby we first collect all the rules which directly belong to the interaction scheme I. Finally, the process is repeated for the superschemes of I, in precedence order from I. This gives rule 5, 3, 4, 2, and 1 following the same example as above.

32. When a class category that starts with "HyperPro" and ends with "Types" is selected, an entity type template will appear in the system browser instead of the usual class template. In the entity type template it is possible to specify the immediate super entity type as well as transient and required attributes.

33. When presenting a link L, we can be sure that we are in context, where we have just presented a node N to which L is attached. (Every presentation is started on a node). We will furthermore assume, that L is anchored in N. L may be an incoming or and outgoing link from N. In a presentation situation of L, we can present *the opposite node of L* compared to N. We cannot in general know whether the opposite node is the source node or the destination node of L.

34. In the current version of HyperPro local interaction is not yet supported. We will probably go for a solution in which we enumerate the mapping from input events to procedure names in association with the interaction scheme. (This is actually a slightly simpler alternative to the proposal described in the main text of this report). However, the solution is along the lines with the current treatment of general interaction.

35. In the current language for interaction rules, it is not possible reference attributes of neighbor nodes. However, it is possible to write and activate a Smalltalk routine from a rule, which accesses such attributes.

36. About hidden presentations of links: In the current implementation of the HyperPro prototype, it is not possible to hide a link presentation entirely in

the text editor. In the internal data structure, on which the editor works, there is a level which stems from the link, even if the link isn't visible in the presentation. Even on the screen, there is at least a single space, which is the presentation of the link. The latter is is, however, an anomaly of the current implementation of the text HyperPro text editor. The reason for this choice is due to Epoch, in which is difficult (impossible) to nest a zone into an empty zone. In any case, it is not easy to select a link if it isn't presented (or presented very "light").

37. If, during the collection of menu entries, a high level event is reached more than once, it is safe and recommended to through away all but the most specific menu item.

Chapter 5

The storage layer

The purpose of the storage layer is to provide shared persistent storage and other services for the kernel layer. It has not been the intention of the HyperPro project to do research in the area of storage models for hypertext, and the present chapter serves merely to document what has been done.

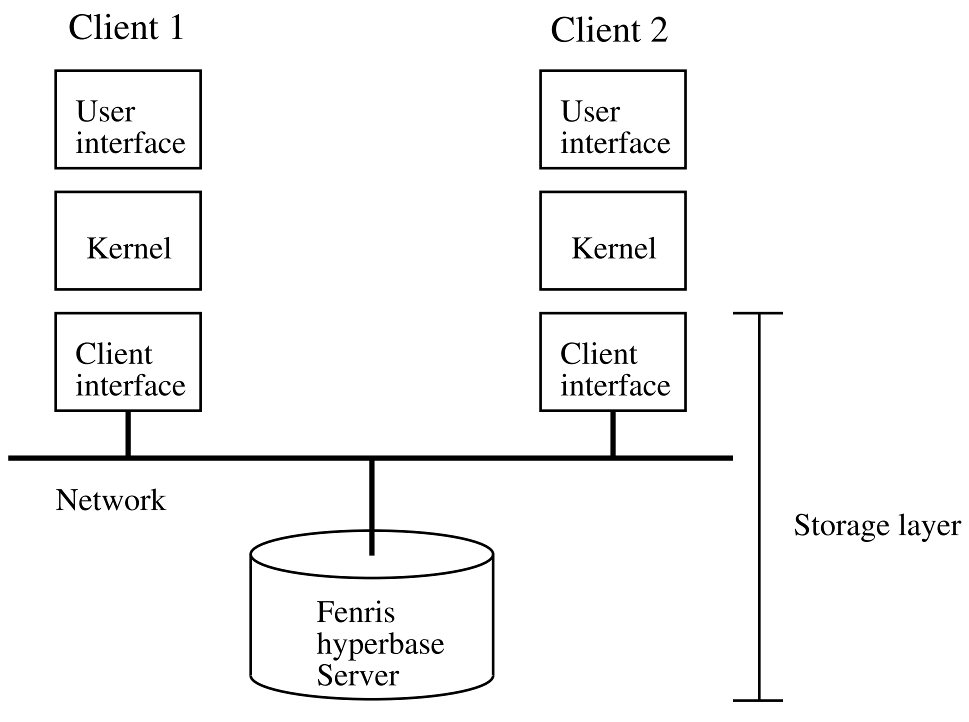


Figure 5.1: *The overall architecture of HyperPro. Each client consists of a User Interface, a kernel, and a client interface. The client interface is one part of the storage layer, the other important part of the storage layer is the Fenris hyperbase*

As illustrated in figure 5.1 the storage layer consists of two parts, a hyperbase server that provides the basic services of the storage layer, and a client interface to mediate communication between the server and the kernel. In this project we have been able to use the Fenris server [3] from an earlier project.

The rest of this chapter will focus on the interplay between Fenris, the client interface, and the kernel. Before we examine this model, we will shortly recapture what services the storage layer must support:

- *Persistence.* Most obviously the storage layer must support shared persistent storage of the hypertext entities, and make them available for retrieval.
- *Access contention.* When several clients want to access the same set of nodes, conflicts can arise when several clients try to access the same entities at the same time. The storage layer must provide means (locking and events) to address these problems.
- *Distribution.* It should be possible to divide the hypertext in to separate documents that can be stored at geographically separate places. This is not a topic we have addressed in our present project. However, we have some ideas which is discussed in the note section, note 1.

5.1 The client interface

The client interface must provide an interface for the kernel that supports the development of the data model described in chapter 3, and an interface to the Fenris hyperbase to deal with the peculiarities of the server. We will start with a discussion of how the client interface interacts with the kernel. The complete design of the storage layer is illustrated in figure 5.2. The figure will be explained in the subsequent sections.

5.1.1 Kernel interaction

The main problem to be addressed in designing the interface between the client interface and the kernel is how to provide a storage mechanism which can stay ignorant of the exact design of the kernel entity hierarchy.

As seen from section 3.2, all entities have attributes, and some attributes contains entity ids (to be explained shortly). As a first requirement we must ensure that all attributes can be stored, and second we must provide a means for handling entities in the attributes.

The Fenris hyperbase is able to store and retrieve individual attributes. The entity manager in figure 5.2 interacts with the root of the entity hierarchy, EntityType, to retrieve and store attributes. The client interface maintains a simple write-through invariant: *The attribute values maintained in the kernel layer and storage layer are consistent at all times.*

When an attribute is stored the entity manager examines the contents of the attributes. References to other entities should not be followed and stored recursively, as the referenced entities are up to date following the above invariant. On the other hand, when we retrieve an attribute from Fenris, we must return something to the

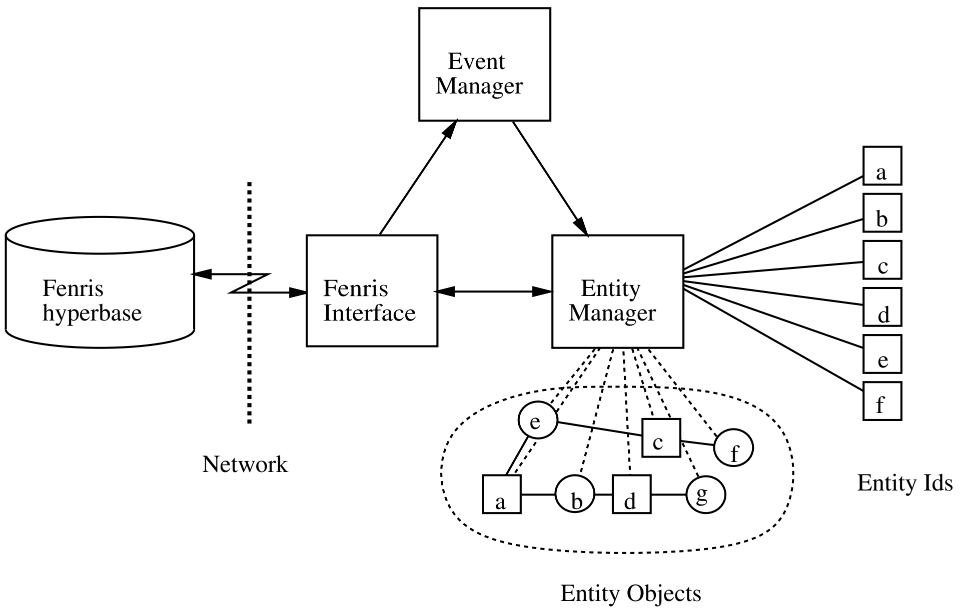


Figure 5.2: *The internals of the client interface. The Fenris interface makes available to services of the Fenris hyperbase. The Entity manager caches and maintains entities at the client side, and the event manager handles events from the Fenris hyperbase.*

kernel that reflects the entity that was stored. Again, we can not retrieve the actual entity because that way we would most likely end up retrieving the entire hyper document.

In order to address the above problems we have introduced a new object type, an EntityId, which is a reference to a hypertext entity. The entity manager, EntityId, and EntityType works closely together.

The entity manager maintains a mapping from entity ids to entity objects, and this mapping is the only references to entity objects. This enables us to let the entity manager cache and manipulate entity objects at is own will. If the entity manager is asked for an entity object that is not currently available, it can retrieve it from the hyperbase, and return the just retrieved entity object just as if it had been readily available, the only difference being response time.

In much the same manner as the entity manager provides a mapping from entity ids to entity objects, the entity objects themselves provides a mapping from attribute names to attribute values. If the value of an attribute is needed, and the attribute has not yet been read from persistent storage, the entity-object will ask the entity manager to retrieve the value, the entity object then returns this value (and caches it for later use).

Entity object creation. Now returning to the really interesting problem. The entity type hierarchy is defined on the kernel side, and new entity objects are created on the kernel side, not the storage layer side; how can we organize the storage layer so that we can support arbitrary new specialized entity types, and the operations they define. This is done using the idea of proxies. When a new entity is created at the kernel side, the entity manager is consulted (see note 2); the entity manager instructs Fenris to create a new persistent entity, and the entity manager retrieves the identity of this new persistent entity. Then an entity object (a kernel representation of the persistent entity) is created by the entity manager, but an *entity-id* is returned instead of the actual new entity object. All future references to the new entity object will go through the entity id.

Entity object operations. The operations for the entity objects are defined as part of the entity type hierarchy, and not the entity ids. However, we would like to avoid the code in the operations to be cluttered with explicit de-referencing through the manager. Instead we just send the messages to the entity id. The operations are not defined for the entity id, which causes Smalltalk to send the "message not understood" message to the entity id. The "doesNotUnderstand:" message has been redefined in entity id to retrieve the entity object from the entity manager (that might have to retrieve the entity object from the server), and the entity id then forwards the message to the entity object.

This makes it transparent to the kernel level that it is not dealing with the real entity objects, but entity ids. The only exception is that it is not possible in the specification of the entity type operations to pass *self* as a parameter to other operations, as that would mean that the real entity object was made known to other entity objects. Instead of self, one should use *thisEntity*, which is the entity id for the entity object.

The scheme of using entity ids as proxies also prevents us from having to retrieve the entire hypertext network recursively when we read an attribute that contains other entities. When we store an attribute which contains references to other entities, it really contains only entity ids, which we can safely store in the persistent store. When we later retrieve that attribute, we only need to recreate the entity id. The entity manager does not actually retrieve the entity object until someone calls one of its operations.

To illustrate the use of proxies, lets us consider a simple call from the kernel side. Assume we have a reference L to an entity id for a link, and we want to get the contents of the destination node of the link. This can be done as:

```
contents := (L destinationNode) atAttribute: #contents.
```

Let us examine in relative detail what takes place to execute this statement. This is captured in figure 5.3.

At 1, L (an EntityId) receives the message "destinationNode". This method is not defined for L, and the runtime system instead sends the "doesNotUnderstand:" message to L (2). The doesNotUnderstand method has been defined, and calls (3) getEntity in the entity manager in order to get the real link object. The entity manager now interacts with Fenris to create a client side link object (4) which is returned to the doesNotUnderstand method of L. L now has a reference to the real object and it can forward the destinationNode message to the real link object (the message that failed is passed as a parameter to "doesNotUnderstand:").

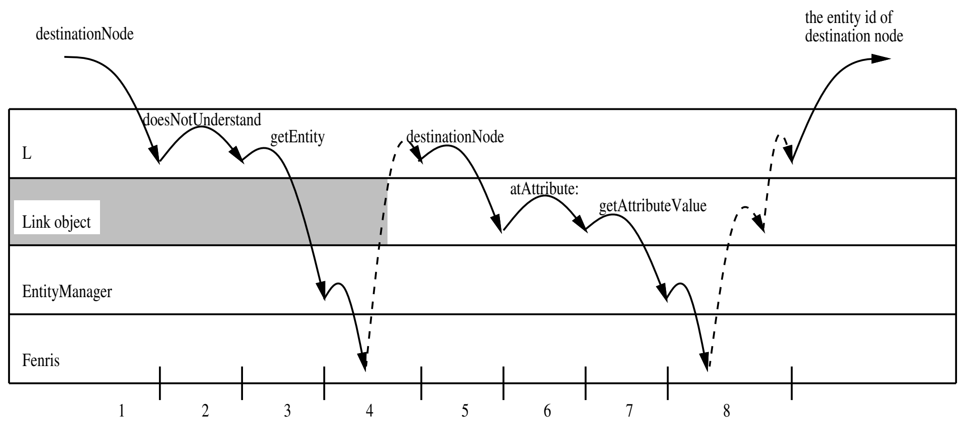


Figure 5.3: *Retrieving the destination node of a link using proxies. The four lines illustrate different objects: L the EntityId for the link we are trying to follow, Link object the client side representation of the link (which is not created until the end of the gray area), Entity manager and Fenris. Solid arrows indicate a method call, an broken lines a return, possibly through several methods.*

The destinationNode method now calls "atAttribute:" at self to get the entity id for the destination node (5). It does this by sending to self, which is safe. The this again calls upon the entity manager to get the attribute value (7). The entity manager asks Fenris to retrieve the attribute from the hyperbase (8), and returns the value (an entity id for the destination node) through the layers.

This node entity id is now send the message "atAttribute: #contents", and the whole story is repeated.

As mentioned, the entity manager caches entities. Had the link object been present in the cache, we would not have had step 4 above, and had the destination node been cached in the link object, we could have avoided step 7 and 8. Accessing Fenris is the time expensive part, as that involves network communication.

5.1.2 The Fenris interface

The client interface of the storage layer must also take care of interfacing between the Fenris hyperbase written in ELK[1], and the client interface written in Smalltalk.

Fenris provides a number of primitive operations on a simple hypertext model, which is available using a low level socket interface (Higher level interfaces were available for clients written in ELK too, see note 3).

The Fenris interface part illustrated on figure 5.2, really consists of two parts. A low level network interface, that takes care of encoding/decoding information, and a higher level which provides higher level interface that allows easy access to the functionality of the server. These high level operations is used by both the entity

[1]Extensible Lisp Kernel, a scheme extension with special facilities for interfacing to C and dynamic loading. See [9] ??? reference

manager and event manager (see below). The operations include operations to read/store specific attributes in a specific entities, to create new entities etc.

5.1.3 The event manager

The event manager is responsible for handling events that come from Fenris. An important part of the Fenris hyperbase server is that it possible to subscribe to specific events. As an example we can ask the server to notify us when a new entity is created, or when attributes are assigned new values etc.

Such events are sent from the Fenris hyperbase, and received by the Fenris interface, which forwards the event to the event manager. This division of labour mainly serves to make the Fenris interface cleaner, and allows us to have a central place to examine events. The event manager works together with the entity manager for those events that have to do with changing attribute values. The event manager informs the entity manager that an attribute has changed, and the entity manager can the either retrieve the new value, or simply remove the value from the cache.

The events part of HyperPro has not been developed, and there it is currently not possible to subscribe to events. However, Fenris is able to send events, and the event manager is able to receive them, but we have not yet tied the ends together, as we have spend our time on other aspects of the system.

5.2 Notes

1. We have tried to foresee the situation where we distribute the hypertext. The idea is that we can have several Fenris hyperbases active at the same time. Each entity id not only contains an identification of an entity within a singe Fenris hyperbase, but also a identification of which entity manager it belongs to. That is, it is possible to have several storage layers active at the same time. This is completely transparent to the kernel layer. Fenris however, only supports identification through a number unique within the server. It should be possible to have entity identifies be both a identification of the hyperbase and a unique number within the hyperbase.

2. When we create a new node it is necessary to ask the manager to create a new persistent entity. We have prepared the client interface for handling multiple Fenris servers, as discussed in note 1. Therefore there might be several managers around, and we must specify in which hyperbase the entity should be created. There are (at least) three basic ways to do this.

   - the new operation is given a name of the server in which the entity must be created, or
   - the new operation is passed some other entity and the new entity is created the same place as that entity, or
   - the system must maintain a "default" document.

   We provide operations supporting the first two.

3. The communication between the Fenris hyperbase and the client interface is implemented directly through TCP/IPC sockets. There are several reasons

for this. First, sockets are nice and symmetric, and easily allows for two way communication and both ends can initiate the communication once a connections has been made. This is opposed to a remote procedure call interaction, where it was not obvious how the server should call back to the clients written in different languages. Secondly, we have have for the above reason had much experience in using socket based communication, and as such it is not much of a problem. And Smalltalk has an excellent library for socket communication.

To facilitate information exchange between Fenris and the client interface, we have developed a simple common data format, that is easy to implement, and avoids the problems of different size integers and problems relating to different conventions for string encodings.

Bibliography

[1] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonay E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification X3J13 document 88-002R. *Sigplan Notices*, 23(Special Issue), September 1988.

[2] 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.

[3] Martin Boel, Asbjørn Gregersen, Poul Larsen, and Frenk Hejselbæk Møller. Manual og dokumentation til FENRIS version 2.0. Technical Report IR 91-02, Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, 1991.

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

[6] Per Abrahamsen et al. Introduction of graph-grammars in programming environments. Project report from Department of Mathematics and Computer Science, Institute of Electronic Systems, Aalborg University, June 1990.

[7] Engels G. and Schäfer W. Graph grammer engineering: A method used for the development of an integrated programming support environment. In *Joint conference on theory and practice of software development, Berlin 85*, 1985.

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

[9] 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.

[10] Claus Lewerentz. Extended programming in the large in a software development environment. In Peter Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 173–182, November 1988. *Software Engineering Notes*, Volume 13(5), November 1988 and *Sigplan Notices*, Volume 24(2), February 1989.

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

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

[13] 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.

[14] 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.

[15] Kurt Nørmark. The design of a hypersturcture programming environment for eiffel. Technical Report Forthcoming, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, 1993.

[16] 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.

[17] 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.

[18] Kasper sterbye and Kurt Nrmark. The vision and the work in the hyperpro project. Technical Report R-93-2012, Aalborg University, Department for Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, april 1993.

[19] 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.

[20] 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.

[21] 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.