Representing Programs as Hypertext

Kurt Nrmark and Kasper sterbye
Aalborg University*

Abstract

This paper discusses the representation of programs as hypertext. Representing programs as hypertext confronts us with the issue of which program structures are to be represented by the external node and link structure, and which program structures are to be represented internal to the nodes. The concept of anchors plays an important role in the interface between the external and the internal structure. We will examine five different models for the interplay between the internal and external structures.

1 Introduction

This paper is obtained by ftp from ftp.iesd.auc.dk:/pup/reports/papers/normark94a. It is an exact copy of the paper published as [7]

The *program representation* (sometimes called the *internal program representation* for clarity) is the data structure which represents a program. The program representation is seen as a contrast to the external *presentation* of the program on the screen, which sometimes is called a view. Text and abstract syntax trees (ASTs) are common program representations today. The problem with a text based system is that there is no structures representing even the most basic syntactic or semantic knowledge of the program. AST-based systems capture all syntactic structures and can be extended to handle many static semantics issues as well. The problem with ASTs is not with representing programs, but when we go beyond programming environments in a narrow sense, and want to find a representation that is suitable for all artifacts produced during software development. We are then confronted with large amounts of information in different formats.

Hypertext is geared towards a uniform representation of information in many different formats. Relationships between two or more information fragments can be represented by links between nodes - but more importantly it is possible through anchors to relate the internals of one node to the internals of another node. Thus we see hypertext as a promising way to integrate programs and their related documents - provided we can find an acceptable way to represent programs using hypertext.

Hypertext is also found attractive as a representation that combines the simplicity of text with some of the expressive power of ASTs. It has been our experience with structure-based editing that the real payoff is at the large scale, where units such as procedures and classes are edited as structures. We find that there is only little payoff at the statement and expression levels. Hypertext seems interesting as a way to

*Department of Mathematics and Computer Science, Fredrik Bajers Vej 7E, 9220 Aalborg , Denmark. (Internet: normark@iesd.auc.dk and kasper@iesd.auc.dk)

represent the overall structures explicitly, while keeping the statement and expression levels as text. One purpose of our research is to examine this hypothesis.

In order to experiment with hypertextual program representation, we have build a prototype system called HyperPro. The system has been functional for more than a year, and we have gained some deeper understanding into the representational issues of storing programs in hypertext. Needless to say, our first design has uncovered some problems. We are interested in being able to show a node at different levels of abstraction, as are typically done in AST-based systems through pretty-printing schemes. This calls for ways to present only part of a node (say, the signature of a procedure). There are fundamentally two different ways to do this, either representing those aspects independent of the actual node, leading to double representations and consistency problems, or we must break the nodes into smaller units which can be accessed independently, but must be combined to get the whole. Breaking the node into smaller units is what we refer to as the internal node structure. We will in this paper discuss alternative ways of organizing internal structures of nodes, and we will discuss how the traditional hypertext model can accommodate these proposals.

We start with a description of basic hypertext concepts, followed by a variation introduced by us, called rich hypertext. Then follows the main section of the paper, which describes five different models of internal node structure in rich hypertext. At the end of the paper we briefly describe some similar work, and we outline our thoughts about programming environments based more solidly on hypertext concepts.

2 Basic Hypertext Concepts

This section briefly introduces a basic hypertext model. This will serve as a definition of vocabulary, and form the basis for our discussions.

A *hypertext* is composed of *nodes*, which are intended as information containers around some contents, and *links* which relates nodes to each others. A node has three major parts, *contents*, *attributes* and *anchors*. A link has two major parts, *specifiers* and *attributes*. An *entity* is a generalization of nodes and links.

The *contents* of a node is where the information in the hypertext is stored. Besides storing this primary information, it is often useful to keep auxiliary information such as author names, node names, modification dates, and so forth in the nodes. This is done using *attributes*, which can be thought of as named fields which hold values of some simple types (formally a partial mapping $entity \times name \rightarrow value$). The notion of *anchors* is a fundamental part of any modern hypertext model, constituting an encapsulation allowing links to relate the internals of one node to the internals of another. The representation of the contents and the attributes are not affected by the anchoring. An anchor has a unique identity (relative to the node), and designates a subpart of, or a position in the node. For nodes with a textual contents *point anchors* or *region anchors* are typical. Point anchors specifies a specific point in the text, whereas region anchors specifies a span of the text.

Links can have attributes in the same manner as nodes. Besides the link must name its source and its destination. For simplicity we work with binary links only, and we will therefore often talk about the source and destination of a link. These endpoints are described using *specifiers*. A link specifier specifies not only a node, but also an anchor identification for each node it relates.

This basic hypertext model is summarized in figure 1.

The Dexter model [4] differs from the above at a few points, most noticeable it allows

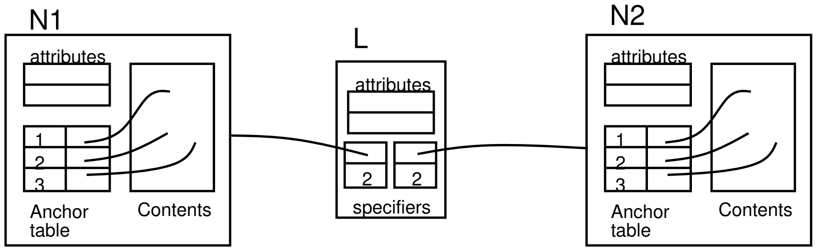

Figure 1: *The link L connects the two nodes N1 and N2. Each node consists of a contents, attributes, and an anchor table, where the value of each anchor designates a subpart of, or a point in the contents. The link consists of two parts, attributes and specifiers, where each specifier of the link designates both a node and an anchor within that node.*

for links to be of any arity, and it allows for links to connect links as well as nodes. The Dexter model also emphasizes a distinction between a storage layer, where the internal structure of a node (and consequently the semantics of anchor values) is not known by the system, and a runtime layer where the internals are known. The runtime layer is typically the editor(s) manipulating the contents of the nodes, and maintaining anchor values. Notice that in an extensible system like HyperPro, the storage layer can be aware of the internals of certains types of nodes, and as such be able to manipulate the anchor values of certain nodes as well.

3 Rich Hypertext

In addition to the basic hypertext concepts, as introduced above, we introduce the notion of *rich hypertext*. A rich hypertext is a hypertext with the following additional characteristics:

1. *Typing*: Nodes and links are typed entities.
2. *Internal structure*: The contents of nodes can be broken into structural subunits.

In addition, it may be attractive to ensure that rich hypertexts obey some topological constraints, which involve the types of node and link instances.

The introduction of typed nodes and links is attractive because it enables us to identify and register the *roles of the nodes*, and the *roles of the links* among these. In a broad sense, the subtools of our hypertext system may take advantage of the typing of entities. In particular, we have drawn on the types of nodes and links in the *interaction engine*, which governs the presentation and interpretation of events on the hypertext.

The overall task of the HyperPro presentation framework (as implemented in the interaction engine) is to compose a hierarchical, textual presentation from a network of nodes and links, given a node as the starting point. The main objective of the framework is to decide how the set of anchored links of a node contributes to the presentation of a node. Based primarily on the types of the involved links (and secondarily on the values of attributes) it is determined which links to present, and how the nodes in the other end of the links should contribute, recursively. There is no single

The text should be broken into pieces which can be understood in relative isolation form their context, and each of which idealy focus on one single aspect of a description. Each piece becomes the contents of a node. For source programs we have found that the named abstractions usually are good candidates for nodes - a procedure, class, type etc.

The size of the node contents should be large enough to convey a coherent and well-balanced meaning of the text; and small enough to fit into a window on a screen. In programming terms, we have found that it can sometimes be a good idea to factor out some fragment of an abstraction into a node of its own, especially when one can write a separate explanation of that node, or if one can give the fragment a name. As an example, initialization parts of a procedure can often be factored out, giving an alternative to procedural abstraction.

Compositional and organizational relations among the nodes should be represented as links, which may be anchored into the contents of the source node. With regard to modeling programs as hypertext we have found that certain aspects of program structure currently done by means of naming is more naturally done using explicit links. As an example, the superclass of a class is most often specified by name, whereas in hypertext it is more natural to have a direct link to the node, which contains the superclass. We have encountered this phenomenon in all cases where we represent programs as hypertext; the graph representation takes over some aspects currently handled by static semantics.

Structure within the individual node is evidently needed when we have a single node represent an program abstraction, such as an entire procedure. As is seen in this paper, we have not yet come to a conclusion on what the right approach to this is.

Figure 2: *Modeling principles for organizing information in hypertext in general, and in representing source code in particular.*

presentation which serves all needs. Consequently it is our goal to be able to define a whole range of different presentations, which supports the user in different application situations. We believe that it is important to provide typographically clear and aesthetically pleasing views on complex descriptions and documentation. Therefore it is important for us to control the layout (indentation) and decoration (font and color) of the resulting presentations. For further information on the HyperPro presentation framework, the reader is referred to a separate report on the interaction engine [9].

When motivating internal structure of the nodes, it should be noticed that the starting point is a hypertext modeling, which subsumes a determination of which logical units of information belongs to which types of nodes. Figure 2 outline the hypertext modeling principles that we have found useful in our work. When presenting a hypertext on a screen we need to have access to well-defined substructures of nodes. Access to such substructures may be used to form *alternative views* of the hypertext. Such alternative views may be useful as *buttons* of anchored links (typically the name of an abstraction), or more generally as some kinds of *node extracts* (such as a procedure signature, composed of the procedure names and the parameter list). It should be possible to edit these alternative views in such a way that the integrity of the whole node is maintained. *Notice that we do not want to affect the basic hypertext model by these concerns, e.g., in terms of breaking nodes into subnodes of finer granularity,* *as this very well might disturb our overall hypertext modeling principles.*

4 Internal Structures of Nodes

In this section we will discuss a number of different models of internal node structure within the context of rich hypertexts. Recall from the introduction that we look for solutions that fit well with the hypertext starting point of our research. The first of the models is the one currently implemented in HyperPro.

4.1 Model 1: Attribute Aggregation

The original idea of this model is simple, and it can be described as follows:

1. The *contents* of the node holds a textual representation of the node's primary information.
2. A number of *auxiliary attributes* represent secondary information about the node, such as author, creation date, editing mode, and information about the language being used.
3. Links can only be anchored in the text of the contents.

The *actual contents* of a node is defined as some aggregation of the contents, the auxiliary attributes, and a number of text-constants. It is the intention that the contents should be presented inside a "shell" of auxiliary attributes and fixed text, and appropriate decoration should be used to let the user easily distinguish between the different parts of a presentation.

It soon became clear that it would be attractive to represent some parts of the contents in the attributes, in order to provide immediate access to these parts from the presentation framework. Such attributes are called *constituent attributes.* In order to avoid double representation and consistency problems between the constituent attributes and the contents we define the actual contents as an aggregation of the remaining contents, the constituent attributes, text constants, and (if needed) auxiliary attributes. It is worth noticing that such a formation of the actual contents is possible using the already existing presentation framework, and as such it is attractive from a purely pragmatic point of view. During editing, we are able to keep track the positions of the various attributes, and store them properly back into the internal attribute structure of a node instance.

Figure 3 shows a sample node from this model together with a presentation of the actual contents. The node represents a class from the Eiffel programming language [6]. The dots in the node contents illustrate anchor points. These anchors are maintained by the node, but referred to from links (see figure 1). In terms of the example, we conjecture that the first dot is an anchor point to a link with an attribute, and that the two following dots are anchor points to routines of the class. For the sake of this example, the name of the class and the type parameter of the class are represented as attributes. The actual contents is aggregated from the string "class", the attribute Name, "[", the attribute TypeParameters, "]", and the contents.

There are a number of problems with this model. First, on the one hand we want to maximize the coverage of the node contents, because anchoring of attributes is only allowed in that part of the node; on the other hand, we want to allow for several

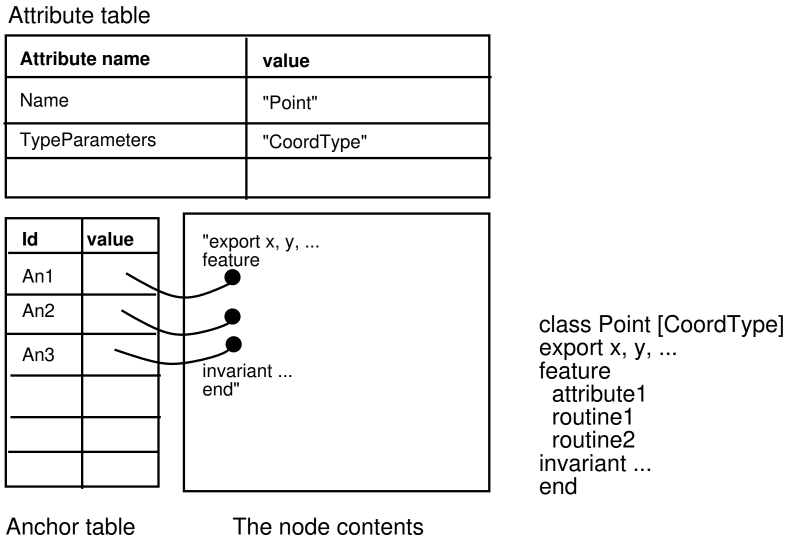

Figure 3: *The representation of a node in model 1 (left) and a node presentation (right).*

fragments of the contents to be factored out into constituent attributes, such as to allow for greater variation in the possible presentations of the node. These desires are contradictory; and in reality the model breaks down if the constituent attributes are equally distributed over the entire span of the actual contents.

Second, the model is only well-suited to define a two-level structure of nodes (consisting of the actual contents, and one level of attributes). Our presentation framework is recursively defined at the macroscopic node and link level, but recursive definition of a single node presentation from its attributes is not easy to obtain in this model. The presentation of a node is a list of presentation elements, in which only the element corresponding to the contents contains nested presentations (presentations of links anchored in the contents, and in turn presentations of the destination nodes of these links, following a recursive pattern).

Third and last, the aggregation of node attributes is simple and primitive, compared with the usual presentation of ASTs, and compared with the presentation framework at the node and link level in HyperPro. The definition of a presentation from the node attribute values, text strings, and the contents is ad hoc, and not nearly as elegant (and powerful) as the presentation framework at the node and link level in HyperPro.

The model just described could of course be repaired and generalized in several directions. Anchoring of links could be allowed in every attribute, at the cost of some added complexity. It is, however, more difficult to realize a generalization to multiple levels of attributes. Not least because of the third problem, we will now discuss a number of different models, some of which may be considered for future versions of HyperPro.

4.2 Model 2: Point Anchored Attributes

The concept of anchored links is one of the primary elements in our basic hypertext model. The idea is here to generalize the anchoring concept to also include anchored

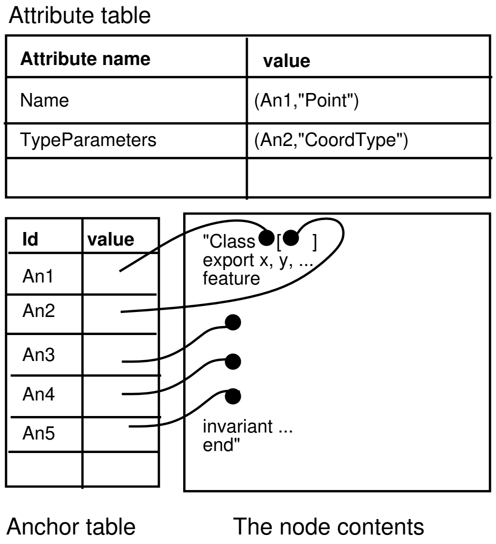

Figure 4: *The representation of a node according to model 2.*

attributes. Thus the model is characterized by the following properties:

1. The textual contents, as in model 1.
2. A number of auxiliary attributes and constituent attributes, similar to the two kinds of attributes from model 1.
3. Links *as well as attributes* may be anchored in the main textual contents.

Figure 4 illustrates the representation of a node, in terms of the same example as used above for the first of our models. As can be seen from the figure, the actual contents may be assembled by determining how anchored values (attributes and links) are *inserted* into the contents of the node for the sake of generating a node presentation. This is different from model 1, in which the actual contents is formed as some *concatenation* of text constants, attribute values, and the node contents.

Insertion of an anchored attribute value may involve a filtering condition (whether or not to include it). Insertion of an anchored link involves a type-specific filtering condition, and a recursive activation of the presentation framework on the link. (Link presentation is similar to model 1). Both cases also involve a determination of the decoration of the inserted elements.

On the positive side, the coverage of the node contents is slightly broadened, and hereby it is possible to anchor links at more places than in model 1. In model 2 it is not a problem to present an attribute in the middle of the node contents. Model 2 makes better use of the anchoring concept (because both attributes and links may be anchored) hereby avoiding the ad hoc creation of the actual contents as present in model 1. However, in situations where the node contents is not presented, we are still forced to aggregate the presentation from text constants and attribute values. On the negative side, we are still left with a two-level structures of the nodes (the actual contents with nested attributes).

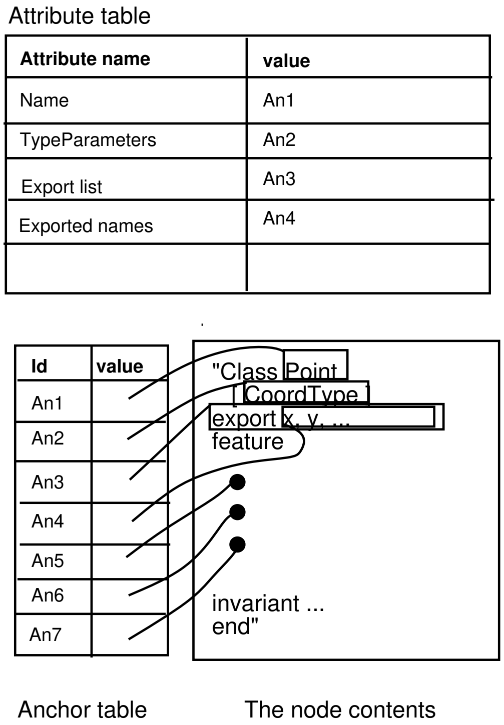

Figure 5: *The representation of a node according to model 3.*

4.3 Model 3: Region anchored attributes

In the models handled until now, anchors have been associated with points in the node contents. We will now, in addition, introduce anchors that are associated with regions of text.

The concept of a region-valued anchor can be realized as an aggregation of two point-valued anchors. The idea embodied in this model is to represent the entire actual contents of the node in the contents-part of the node, and to superimpose a number of region-valued anchors on the main contents. Some of the attributes may now refer to a region-valued anchor. The "real value" of such attributes may, or course, be regarded as the subtext spanned by the region of the anchor.

Figure 5 shows our example, in which we use region-valued anchors as attributes. The value of the attribute Name is an anchor; the value of that anchor is a region of the main contents. The example is extended somehow in comparison with figure 3 and 4; the export list is a region, in which the list of the names is a region itself.

Compared with the previous models, no concatenation nor insertions of attributes are necessary in order to form the actual contents of the node. However, some specifications of the decoration of the presentation of the node may still be necessary. In model 3 it is straightforward to nest regions into each other, and hereby a multi-level internal structure can be imposed onto the main contents. It is also possible to have point anchored links in the middle of regions (leaving the impression that links can be anchored in the text of the attributes). Alternative presentations ("buttons", etc.), in terms of concatenated attribute values (subtexts of the node contents) and textual constants, must still be formed by the exact same means as in model 1 and model 2.

The added power in model 3 comes from the slightly more complicated concept of regions. The HyperPro kernel, which manages the representation of nodes, needs to be concerned with the bookkeeping of the regions. [1] Furthermore, we may need to check for overlapping regions, that are not nested properly into each other.

Region-valued anchors are, however, useful in other areas than 'internal node structure'. It will be very useful to have region-valued anchors in the destination end of a link.[2] It is not yet clear whether we will make use of region-valued anchors for source anchoring of links.

4.4 Model 4: The Freja system

In a Masters project, Jesper Care and Jens Peter Hedegaard have designed a reflexive hypertext model named Freja [2]. However, from the point of this discussion the important aspect is that they allow for anchoring in attributes as well as in the node contents, and that they have a different view on presentation.

In Freja, the authors logically distinguish between *plain nodes*, each of which has one main contents, and *structural nodes*, which do not have a single main contents, but are composed of several attributes, possible with anchoring into several attributes. Plain nodes are useful for representing documentation and simple paragraphs of text, whereas structural nodes are better suited for representing programs.

Following the Freja model, the representation of the Eiffel example will be as shown in figure 6. Notice that we have expanded the example to include a class comment, and we specify that we can anchor links in the invariant and in the comment. (The model only allows for point-anchored links). The presentation of structural nodes is specified as a fixed text of the presentation scheme, with openings for attributes, and in particular attributes with embedded anchors. The presentation specification also allows for the presentation of *un-anchored* links. While the presentations in Freja are specified using hypertext itself, the functionality is similar to specifying a presentation as:

> "class @Name [@TypeParameters] - - @comment export @Export
> feature &featureTypeLinks invariant @Invariant end"

with @Attributename meaning that the actual attribute value is to be inserted, and &Linktype meaning that links of the particular type will be embedded at that point.

The strong points of this model is that it treats the different levels of abstraction (full contents, buttons, and other node extracts) in a uniform manner, it solves the problem we have in HyperPro where we try to extend the coverage as much as possible to enable anchoring, and it does this in a consistent manner. The weak sides are that the model has given up the idea of storing the logical contents as a whole, and that it does not handle hierarchical structuring. In addition, there are currently no means to handle the mutual presentation sequencing among links of a given type.

[1] In the current HyperPro system, where we only support point anchors, the necessary bookkeeping can be delegated entirely to the textual editor front end of the system.

[2] In the current version of HyperPro, the source end of a link can only associate point-valued anchors. This is not considered a limitation, because the HyperPro presentation framework may make the user believe, that the link presentation is a region of text. We have not yet implemented anchoring in the destination end of the link.

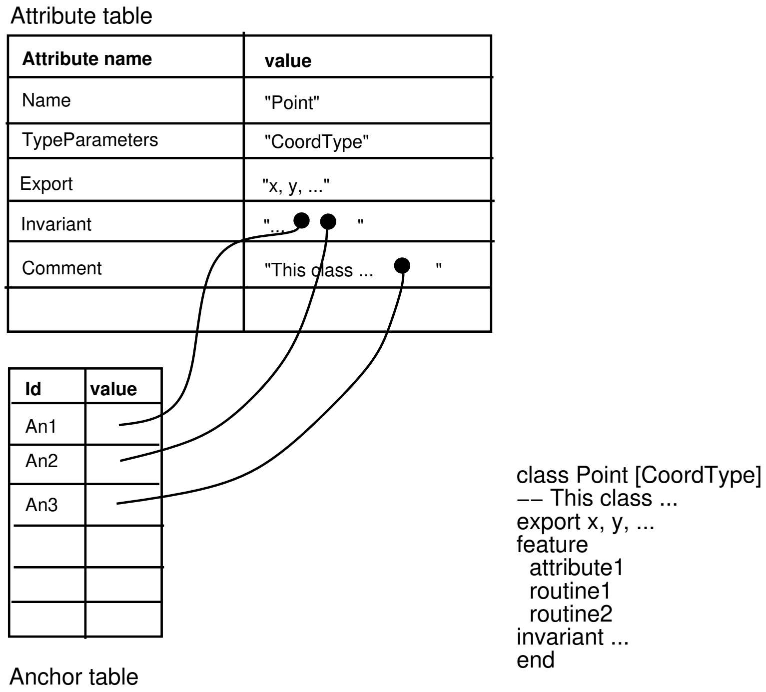

Figure 6: *Model 4: The Freja structured node; notice the absence of a node contents, and the possibility to anchor into attributes.*

4.5 Model 5: AST as internal structure

As an alternative to the previous models we might choose to represent the programs internally as abstract syntax trees (ASTs). This will give us a model where we can present the programs at many levels of detail, and we can draw upon years of experience in manipulating and accessing such structures. While we can see many attractive properties of such an approach, we have refrained ourselves from this model. The reason is that part of our research is to investigate how far we can go (and which problems we encounter) with text-based representations.

If we decide at some point that we are not able to come through with a text based-model, we must reconsider an integrated hypertext and AST-based model. The anchoring mechanism is still interesting in an integrated model, because we can have the anchor values pointing to subtrees of the AST. If we provide the appropriate editors to handle this new type of nodes, the anchor encapsulation will allow us to handle AST-based nodes on an equal footing with text-based nodes.

For the moment we view the introduction of AST-based nodes more as an introduction of a new media type with its own set of problems, than as a direct solution to our text related representation research.

Model 5 is not directly comparable to the Mjlner system [10], where the fundamental model of information is an AST, with some possibilities to link from one AST to another. The Mjlner approach and model 5 might however be able to be merged with some efforts. One problem in taking ASTs as a starting point is that ASTs are not geared towards the representation of relatively unstructured text, such as documentation.

5 Related Work

Dynamic Design, described by Bigelow and Riley in [1], is one of the earliest works on representing source code in hypertext. One eminent difference between HyperPro and Dynamic Design is our emphasis on language independence, whereas Dynamic Design was designed specifically to support the C programming language. Because we do not commit our system to a specific language, there are issues pertaining to the semantics of programming languages which we cannot easily support, which is handled by Dynamic Design. Most noticeable: automatic conversion of existing source code into hypertext, and links reflecting the static semantic aspects of the language, for example links from name applications to name declarations. On the other hand, Dynamic Design does not support the ideas of abstract presentation and flexible interaction patterns, which has guided much of our research in HyperPro.
At the issue of modeling principles, Bigelow and Riley states the following two principles: 1) If a piece of data is referenced from more than one place, it should be in a node by itself. 2) It is the applications using the hypertext that are the final arbitrator of what goes into a node (in their example they discuss the granularity of the (incremental) compiler; can it compile a single procedure, or must it work with a whole module). The second point is interesting from our perspective, as we have paid much attention to facilities for aggregating text. Thus, we are free to chose a finer granularity than what is demanded by the applications, as we can combine the text of many different nodes and attributes to form the text to be handed to a specific application.

Where we have taken the approach that the node granularity is to be kept at a certain level (guided by a set of principles, such as those in figure 2), the IPSEN project described in [3] is based on directed attributed graphs. Attributes are attached to nodes only, and are related through directed edges. Both nodes and edges have types. When going to such fine grained structure, there is no longer a need for anchors, at the cost that all structure must be represented at the same graph-oriented form. We believe that the hypertext concept of anchors is more appropriate, as it provides a convenient information hiding principle that allows us to integrate information of different formats into our system. One very promising aspect of IPSEN though, is their use of graph grammars. Graph grammars provides a formally well founded technique for restructuring a graph, which addresses a need we have felt at several occasions: how do we define high-level commands that will manipulate the hypertext in such a way that overall structural constraints are not violated. We have so far avoided using graph grammars, but it is becoming evident that we need to do something to solve the problem of specifying and enforcing well-formed hypertext.

6 Perspectives and Conclusions

In this paper we have discussed how to represent source programs in terms of the basic hypertext concepts of nodes, links, attributes, and anchors. The main emphasis of our discussion has been how to make an internal node structure fit into an external hypertext structure, and how a traditional hypertext model can accommodate an internal node structure. In this section we will from a somewhat broader perspective argue that hypertext may be interesting as an underlying foundation for programming environments.

We find that that hypertext has something to offer at three important levels in a programming environment:

- *At the storage level*, because it can handle persistency in an alternative way to file systems, and because some amount of multi-user coordination may be done in the server, which manages the hypertext store.
- *At the level of the internal representation*, because hypertext modeling in general seems to be interesting also for structuring of programs, because links make up a systematic representations of relations among pieces of program and documentation, and finally because the hypertext concepts are strong enough to provide for some internal structure on the nodes (as argued in this paper).
- *At the user interface level*, because the hypertext user interface metaphor (creating and following links) may be interesting alternatives to the use of special-purpose browsers and structure-oriented editors.

In other parts of our work [9] we have demonstrated how the user interface of a hypertext-based programming environment can be managed by use of a component which we call an interaction engine. Many of the motivations behind the research reported in the present paper comes from our work on the interaction engine. We have also been concerned with how to handle versioning of information in hypertext [8]. Clearly, this is also of prime interest in case we want to use hypertext as a main foundation of a programming environment. There are, however, important areas of programming environments where we until now only have found weak arguments in favor of hypertext support. One such area is program execution (testing, debugging, and analysis of the dynamic properties of a program).

There are also areas where we have identified problems with our current hypertext support. One such problem is related to effective interaction at an appropriate abstraction level on the hypertext. It may sometimes be tedious to manipulate only single links during interaction with the system. In some cases it would be attractive to manipulate subnetworks instead, and to handle the problem of how to link or unlink such a subnetwork relative to an existing network. As already mentioned, it might be interesting to try out the graph editing techniques, as developed in graph grammar-based systems, on hypertext based systems. Another problem is related to the situation where many cross reference links are needed. As an example one can imagine a link from each applied name to its definition. Again, it is not realistic that the programmer creates each of these links manually. The notion of transient links created as the result of some program analysis (as used in [5]) may be interesting. Within these areas, more research and experiments are needed.

References

[1] James Bigelow and Victor Riley. Manipulating sorce code in dynamic design. In *Proceedings of the HyperText'87 conference*, pages 397–408, 1987.

[2] Jesper Carøe and Jens Peter Hedegaard. *Meta systems in Hypertext*. Aalborg University, Department of Mathematics and Computer Science, Institute for Electronic Systems, Fredrik Bajers Vej 7, DK-9220 Aalborg, 1993. In Danish.

[3] G. Engels, C. Lewerenz, M. Nagl, W. Schäfer, and A. Schürr. Building integrated software development environments part I: Tool specification. *ACM Transaction on Software Engineering and Methodology*, 1(2):135–167, April 1992.

[4] Frank Halasz and Mayer Schwartz. The Dexter hypertext reference model. *Communication of the ACM*, 37(2):30–39, February 1994.

[5] Jussis Koskinen, Jukka Paakki, and Airi Salminen. Program text as hypertext: Using program dependencies for transient linking. Unpublished paper from Department of Computer Science and Information Systems, Univeristy of Juväskylä, PO Box 35, SF-40351 Juväskylä, Finland, 1994.

[6] Bertrand Meyer. *Object-oriented software construction.* Prentice Hall, 1988.

[7] Kurt Nørmark and Kasper Østerbye. Representing programs as hypertext. In Boris Magnusson, Görel Hedin, and Sten Minör, editors, *Proceedings of Nordic Workshop on Programming Environment Research*, pages 11–24. Lund Institute of Technology, Lund University, Sweden, 1994. LU-CS-TR:94-127.

[8] 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.

[9] Kasper Østerbye and Kurt Nørmark. Tailorable interaction using the HyperPro interaction engine. Technical Report R94-2017, Aalborg University, Department of Mathematics and Computer Science. Fredrik Bajers Vej 7E, 9220 Aalborg Ø, Denmark, April 1994.

[10] Elmer Sandvad. *Object Oriented Environments*, chapter 23, pages 345–358. Prentice Hall, 1993.