ISSN 0105-8517

Hypertext in an Object-Oriented Programming Environment

Elmer Sandvad

DAIMI PB – 280
May 1989

AARHUS UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
Ny Munkegade 116 – DK 8000 Aarhus C – DENMARK
Telephone: + 45 86 12 71 88 Telex: 64767 aausci dk

Hypertext in an Object-Oriented Programming Environment *

Elmer Sandvad
Computer Science Department,
Aarhus University, Ny Munkegade 116,
DK-8000 Aarhus C, Denmark
e-mail: ess@daimi.dk

May 1989

Abstract

This paper describes how hypertext concepts are supported in an object-oriented programming environment. Program fragments and documentation fragments are modelled as objects in an object-oriented programming language and several kinds of links are provided between these objects. Therefore the term hyperobject system is used. The links support document organizational relationships, abstract presentation, annotations, program semantical relationships, and program-documentation relationships.

Keywords:
Hypertext, Documentation, Grammar Based,
Object-Oriented, Programming Environment

*To be presented at WOODMAN'89: Workshop on Object-Oriented Document Manipulation, Rennes, France, May 29-31 1989.

1 Introduction

Hypertext systems [Conklin 87] have received much attention in recent years. A lot of hypertext systems have demonstrated the usefulness of organizing documents in collections of nodes connected by directed links. Besides the organizational aspects, links can support the variety of relationships between pieces of information. A hyperdocument can be accessed in a non-linear way by following the links, and the reader can make annotations and define new links.

The message of this paper is to demonstrate how the ideas and concepts of hypertext systems can be applied to a programming environment and thereby getting the same advantages as in hypertext systems. Because the starting point is a grammar based programming environment it is possible to go a step further with respect to finer grained relationships and automatical definition of certain link types.

The programming environment is the Mjølner BETA System [1]. The part of the Mjølner BETA System that supports hypertext is called the hyperobject system because nodes in the hypertext network are modelled as objects in an object-oriented programming language. The language is BETA [2]. In this paper the focus will be on the hyperobject system.

The purpose of the hyperobject system is to support documentation and navigation in large programs and their corresponding documentation. The hyperobject system offers the following:

Document types

Two basic document types exist: program documents and documentation documents. The term documentation covers any description in the range between informal prose to formal specifications. In the software life cycle several kinds of documentation documents are produced e.g. specifications, design documentation, program documentation and user

[1]The Mjølner BETA system [Mjølner BETA 89] is one of several results of the Mjølner project [Mjølner 87] which is a Nordic project developing programming environments, that are primarily aimed at supporting the object-oriented style. In this paper only the Mjølner BETA system will be described.

[2]BETA [BETA 87] is a modern object-oriented programming language in the Simula tradition.

documentation but also more unstructured pieces of information are used.

Document structure

Program documents and documentation documents can be plain text objects or structured objects. Most programming languages and some documentation languages have a formal structure that can be described by a context free grammar. An integrated text and structure editor supports the creation and modification of documents in these languages. Text objects are edited by means of the text editor. Structured objects are also called document fragments or just fragments.

$$\begin{tikzcd}
\text{documentation} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{program} \arrow[d, leftrightarrow] \\
\text{documentation} \arrow[r, leftrightarrow] & \text{program}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{structured object} \arrow[r, leftrightarrow] \arrow[d, leftrightarrow] & \text{text object} \arrow[d, leftrightarrow] \\
\text{structured object} \arrow[r, leftrightarrow] & \text{text object}
\end{tikzcd}$$

Fig. 1: Full link symmetry

Links

Links are provided between any combination of document type and document structure, see Fig. 1. Links can go within as well as between documents. The upper diagram illustrates support for links between program objects, between documentation objects, from program objects to documentation objects and vice versa. The lower diagram illustrates one of the most important contributions of the hyperobject system: support for links within as well as between structured objects. The structure of these objects is grammar based. Text objects can be connected to structured objects and vice versa. Annotations are examples of a link from a structured object to a text object.

Hierarchy and Network

The documents can be organized hierarchically supplemented by a network of relationships. In [Conklin 87] the terms organizational and referential links are used. Organizational links implement hierarchical information; they connect a parent node with its children and thus form a strict tree subgraph within the hypertext network graph. Referential links are the kind of links that most clearly distinguishes hypertext. This reference technique is non-hierarchical. The hyperobject system supports organizational as well as referential links. Five different link types are provided by the hyperobject system. These are document organizational links, abstract presentation links, annotation links, program semantical links, and documentation links. The first two are organizational and the last three are referential.

Abstract Presentation

At the fragment level abstract presentation is used as an organizing mechanism. Abstract presentation presentation corresponds very much to outlining known from some word processing systems. Abstract presentation is provided on structured objects, it is based on the formal structure of programs as well as documentation.

Program Semantical Links

Program semantical links are examples of links between structured objects. Structured objects are represented as abstract syntax trees, that are very suitable for providing program structural and program semantical information. An example of a program semantical relationship is a definition-use relationship. Links are automatically generated from uses of an identifier to its definition. This finer grained automatically generated link type is unique in the hyperobject system.

Program View and Documentation View

The hyperobject system can be used from different view points. In the program view the main emphasis is on programming. A collection of program fragments are browsed either hierarchically or by following different referential links. The main focus is on the structure of the program and on the semantic relationships between pieces of code. Sometimes links to documentation objects are followed in order to understand the program.

In the documentation view the emphasis is on producing or reading the documentation of a software system. A collection of documentation fragments are browsed. Sometimes links to program fragments are defined or followed.

The structure of the paper is as follows: The rest of this section discusses hypertext systems for software engineering and syntax-directed document editing. Sections 2-6 present the five link types in detail with focus on functionality. Section 7 presents the basis of the hyperobject system. Section 8 discusses related work.

1.1 Hypertext Systems for Software Engineering

Hypertext systems have been developed for various application areas. NoteCards [Halasz 88] and Neptune [Delisle & Schwartz 86] are examples of general hypertext systems. Neptune however was designed with a specific application area in mind: software engineering. Another example of a hypertext system for software engineering is DIF [Garg & Scacchi 88], but the number of hypertext systems for this application area is very small. Many papers on hypertext systems mention the capability of supporting documentation of software systems either by means of annotations on programs or by means of linking various documentation documents together mutually or together with the program code. But all these examples are at a rather coarse level. Whole text documents are linked together with other text documents. The structure and semantics of the program are either not considered or only supported manually. This is because the starting point of most hypertext systems is a database of text documents, that are linked together.

In the hyperobject system the starting point is a grammar based programming environment with tools that operate on the formal structure of programs. Program fragments and documentation fragments can be linked together in terms of their structure instead of in terms of positions in a text. In [Delisle & Schwartz 86] automatic support for finer grained relationships like definition-use relationships was advertised for. In the hyperobject system such program semantical links are created automatically and inserted in the structural representation of programs. In section 8 the hyperobject system is compared to NoteCards, Neptune and DIF.

1.2 Syntax-Directed Document Editing

In the software life cycle several kinds of documents are used e.g. specifications, design documentation, program documentation, user documentation in addition to the source code. The languages used in these documents are more or less formal, but even pure prose documents may have a certain structure that can be described formally. There is put a lot of effort currently in developing document standards.

The ODA standard (ISO/DIS 8613) separates the definition of a document into a logical structure and a layout structure. The logical structure of a document (like chapters, sections and paragraphs) can be described by a context free grammar, and thereby be supported by a syntax-directed editor. In order to handle the layout structure additional effort is required, essentially word processing facilities like style and format. In [Hansen & Hestbæk 89] [3] it is described how the ODA standard can be supported in the Mjølner BETA System.

GRIF [André 86] is an example of a structure directed editor for editing and formatting. A special problem with syntax-directed document editing is support for graphics, ranging from graphical notations like mathematical formula and tables to graphical description languages. Edimath [Quint 83] is an example of how graphical notations can be supported. Another example is reported in [Holdam & Nørgaard 86]. In [Sandvad 88] general syntax-directed graphical editing is discussed. In the current prototype of the hyperobject system word processing facilities are very primitive, only the logical structure is supported. Syntax-directed document editing will not be further discussed in this paper.

[3] Is also presented at this workshop.

2 Document Organizational Links

The Mjølner BETA System is aimed at supporting design, implementation and maintenance of large production programs. In order to cope with large systems, it must be possible to break down a program into (or more realistic: build up a program from) smaller components that are tied together in a well-defined way. The fragment system which is based on [Kristensen et al. 83] provides this facility. In the following a simplified version the fragment system will be briefly described with special focus on the hypertext facilities. The focus is on fragmentation of programs but the same principles can be applied to structured documentation documents.

In the BETA programming language there is no special constructs for dividing a program into separate pieces (files). This aspect is handled by the environment. The basic idea is to define a fragment according to the grammar. A fragment is a sentential form i.e. any legal sequence of terminal and nonterminal symbols. A nonterminal symbol is normally used in a syntax-directed editor as a placeholder for empty program templates of the same syntactic category. In the fragment system a nonterminal symbol can act as a slot where an existing fragment with a certain identification and of the same syntactic category can be plugged in. This is illustrated in Fig 2.

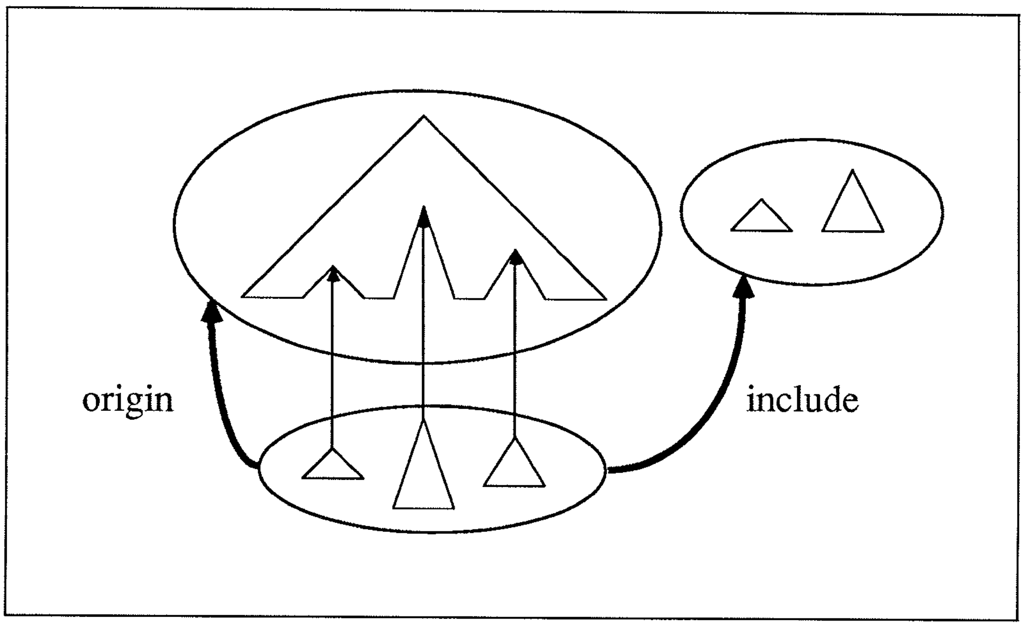


Fig. 2: The fragment system

If the entire program was contained in one program fragment, there would only be one abstract syntax tree (AST). In the figure however subtrees of the AST, have been cut out into isolated program fragments. The program fragments have links to the fragment where they where cut out. A BETA program is a collection of such program fragments.

There are two ways of organizing the variety of fragments. Besides the UNIX file directory there is a group construct that is used to aggregate program fragments. A group [4] is simply a collection of program fragments. Two kinds of links connect groups together. An organizational link (called 'origin') that specifies the group whose slots are going to be filled. And a referential link (called 'include') that is used to refer to a library.

A group browser provides, besides navigational access to groups, the following functionality on a group:

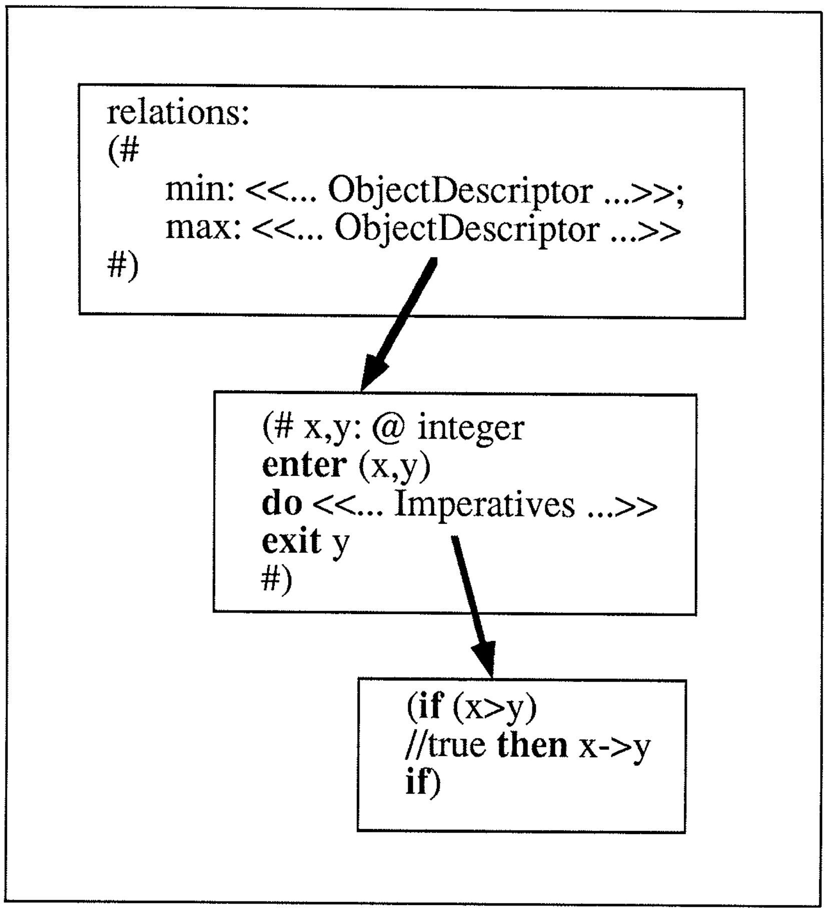
- a survey of the local fragments in the group and links to other groups.
- creation and modification of links to other groups
- addition or deletion of local fragments
- activation of the editor on a local fragment
- activation of the compiler starting with this group. The compiler automatically follows the links to get a complete program. Consistency of versions is checked.
- execution of this group, if possible

The group browser exists in two versions: a graphical one that presents a group as a window with different icons representing links and local fragments and a textual one that presents a group in a menu with entries for each link and local fragment. The interface to the database of groups and fragments is through the group browser. The editor operates on the fragment level. A typical situation in a working session is having a collection of editor windows on the display, one for each fragment.

[4] In this paper the terminology of the fragment system has been adapted slightly. The correct terminology is fragment group, fragment form, fragment link instead of group, fragment and link.

3 Abstract presentation

The fragment system is used to divide large documents into smaller fragments. Most non-trivial fragments, however are normally too big to fit into a window on the screen, even if the window occupies the whole screen space. Inside the editor there is another way of organizing large fragments, or at least the presentation of them. Abstract presentation can be considered as supporting intrafragment organizational links. The user has the possibility manually to substitute any structure in the document by a so-called abstraction, which acts as a link to the suppressed details. In Fig. 3 there are 3 abstractions.



```
relations:
(#
    min: <<... ObjectDescriptor ...>>;
    max: <<... ObjectDescriptor ...>>
#)
```

```
(# x,y: @ integer
enter (x,y)
do <<... Imperatives ...>>
exit y
#)
```

```
(if (x>y)
//true then x->y
if)
```

Fig. 3: Abstract presentation of a program fragment

At the highest abstraction level class `relations` has two operations: `min` and `max`. The details of these two functions are suppressed and instead abstractions are presented. The syntax <<... `ObjectDescriptor` ...>> is not part of the BETA language, but it indicates that a construct of the syntactic category `ObjectDescriptor` has been suppressed. When an abstraction is activated (the link is followed) the underlying structure replaces the abstraction (this is sometimes referred to as replacement links). If the abstraction of the maximum function is activated (e.g. by double clicking with the mouse) the next level of abstraction appears, in this case the body of the maximum function. But this level contains also an abstraction: <<... `Imperatives` ...>> .

Any construct in the program fragment can be abstracted. The operations 'abstract' and 'detail' give the user the possibility to consider the document at any abstraction level. Abstractions can also be inserted automatically when a document is opened. Note that the candidates for abstraction are language specific. For programming languages the candidates might be modules and procedures and for documentation languages it might be chapters and sections. Abstract presentation of a program fragment or a documentation fragment has several advantages:

Overview

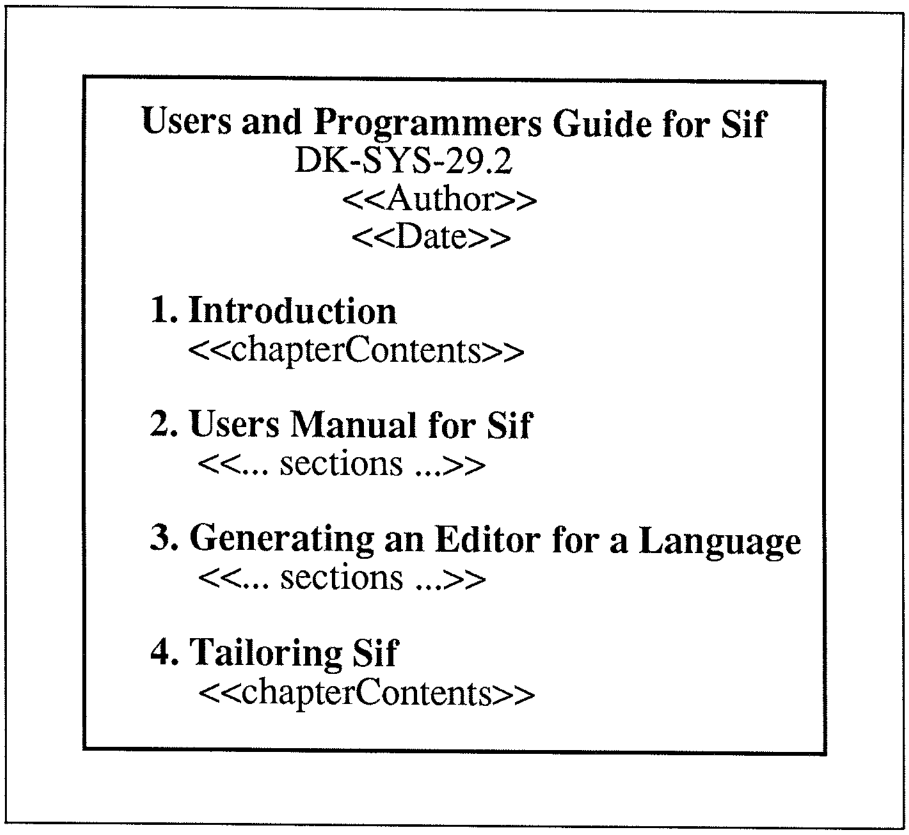
It provides an overview of the document. The whole document can be surveyed at once in one window without scrolling through pages of text. This facility is also known in some word processing systems as outlining.

Browsing

Browsing is done by interactively detailing parts of an abstract presentation. If the document is a technical report with chapters and sections and the like, the highest abstraction level can actually be an interactive table of contents. See Fig. 4. In this example the sections of chapter 2 and 3 are suppressed whereas the chapters the chapters 1 and 4, and the author and date are unexpanded nonterminals.

Documentation

Snapshots of a program at different abstraction levels can be very useful for documentation purposes. One example is a so-called functional specification: a survey of modules and procedure headings with comments describing parameters and the purpose of the procedures. The user can select an appropriate abstraction level by detailing or abstracting the relevant constructs of the document and save the actual abstraction level including comments on textual form. An abstraction level can also be generated automatically.



Users and Programmers Guide for Sif
DK-SYS-29.2
<<Author>>
<<Date>>

1. Introduction
<<chapterContents>>

2. Users Manual for Sif
<<... sections ...>>

3. Generating an Editor for a Language
<<... sections ...>>

4. Tailoring Sif
<<chapterContents>>

Fig. 4: Abstract presentation of a documentation fragment

4 Annotations

Another way of compressing information on the screen is the way traditional comments are handled in the editor. This is an example of intrafragment referential links. See Fig. 5.

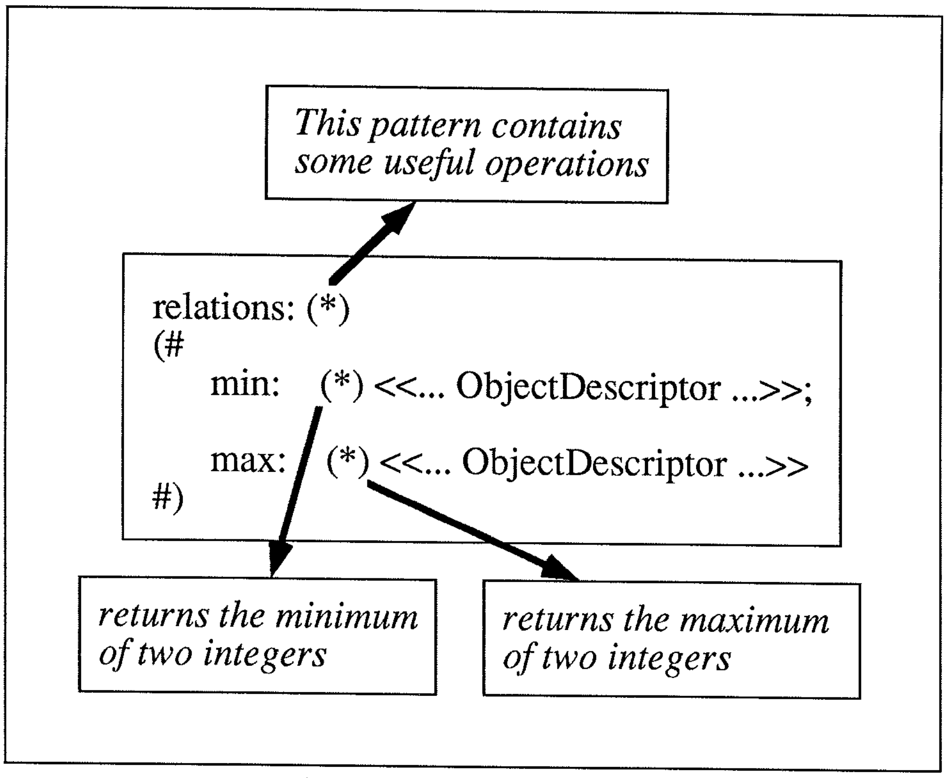

This pattern contains some useful operations
relations: (*)
(#
min: (*) <<... ObjectDescriptor ...>>;
max: (*) <<... ObjectDescriptor ...>>
#)
returns the minimum of two integers
returns the maximum of two integers

Fig. 5: Annotations

Comments in a program are handled by means of links to simple text objects, so-called annotations. Any point in the program can be linked to a text object. If a construct in a program fragment is selected, a text window can be opened and the annotation can be entered. After finishing the annotation a special annotation mark (*) is inserted in the construct to indicate a link from the construct to a text object. Whenever the user selects a program construct with a annotation link, a text window can be activated (e.g. by double clicking with the mouse) and the annotation can be read or modified.

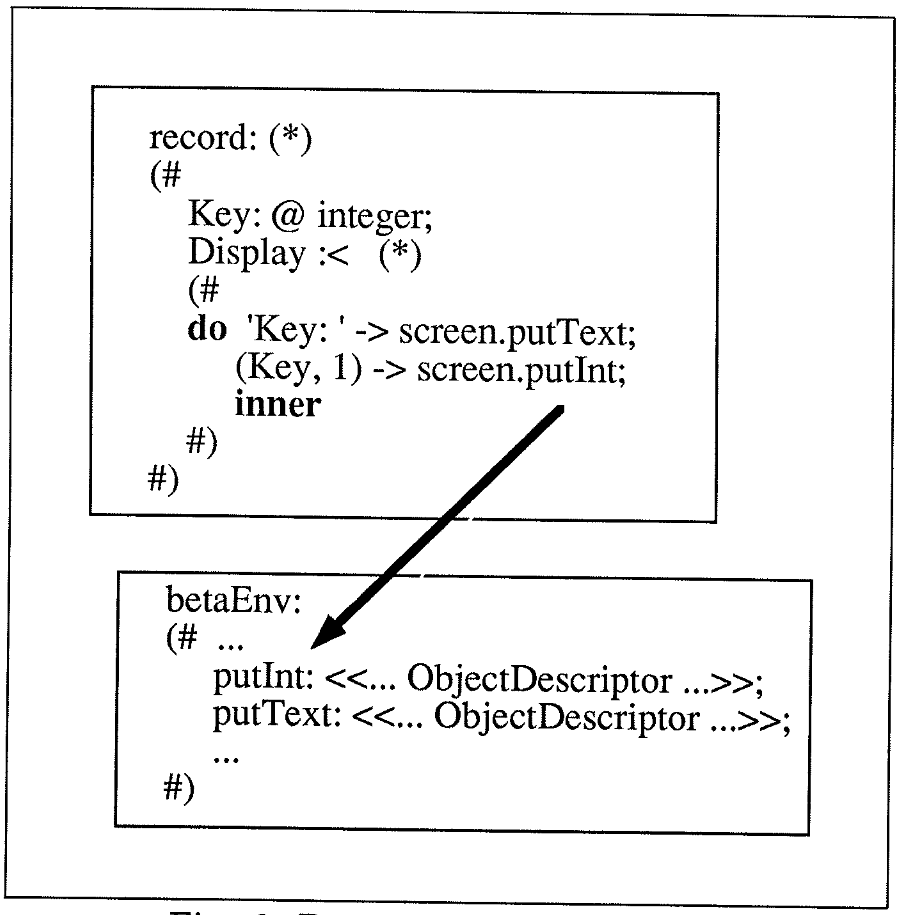
When the normal textual form of the program is requested, the annotation is inserted instead of the annotation mark as a conventional comment. In addition this link type is automatically set up if the program is parsed from textual form.

Annotations can be copied or moved around in the program fragment by means of usual cut, copy, paste operations.

A simple text object can also be on a separate file, but in this case it is not considered as a conventional comment, but rather as an isolated piece of documentation.

5 Program Semantical Links

The program semantical links are used to reflect the static semantic information of a program. For example definition-use relationships and superclass relationships. Such relationships are automatically deducible from the program and should be supported by an automated tool. In the Mjølner BETA System program semantical links are set up by the checker. These links are used in the checking and coding processes, but are also provided to the user in the editor. Fig. 6 shows an example of a link from a use of the `putInt` procedure to its definition.



```
record: (*)
(#
   Key: @ integer;
   Display :<  (*)
   (#
   do 'Key: ' -> screen.putText;
       (Key, 1) -> screen.putInt;
       inner
   #)
#)
```

```
betaEnv:
(# ...
    putInt: <<... ObjectDescriptor ...>>;
    putText: <<... ObjectDescriptor ...>>;
    ...
#)
```

Fig. 6: Program semantical links

When a construct is selected in a program fragment (in the editor) a menu presents the available links from that construct (if any). If the program fragment has been parsed from textual form and the checker has not been activated, no semantical links are inserted.

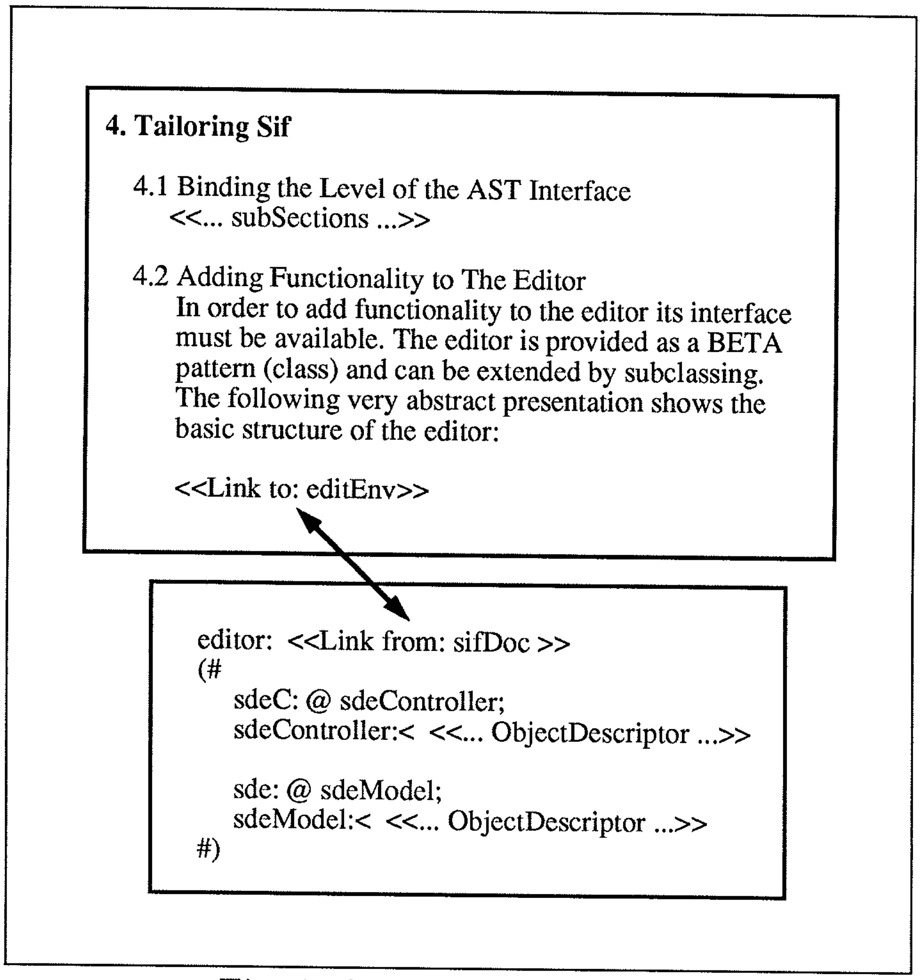
Note that program semantical relationships go across the fragment structure.

Interactive program analysis is normally not considered being part of program documentation, but language specific inspection of a program is often useful when trying to understand it. This kind of program traversal can be considered as non-hierarchical browsing.

6 Documentation Links

Documentation links are used to support all other kinds of relationships between documentation fragments mutually and between documentation fragments and program fragments. This link type is manually created by the user in the editor. The documentation link type is the basic mechanism for supporting integration of program and documentation.

Any point in a program or fragment can be linked to another point in the same or another fragment. When a construct in a fragment is selected in the editor, the construct can be marked as a link source. The link destination is chosen by selecting another construct in the same or another fragment (possibly after activating an editor instance on the destination fragment using the group browser) and issuing the 'make link destination' command. A link mark is inserted in the source construct as well as the destination construct. A descriptive text can be associated with each end of the link. See Fig. 7.



4. Tailoring Sif

4.1 Binding the Level of the AST Interface
<<... subSections ...>>

4.2 Adding Functionality to The Editor
In order to add functionality to the editor its interface must be available. The editor is provided as a BETA pattern (class) and can be extended by subclassing. The following very abstract presentation shows the basic structure of the editor:

<<Link to: editEnv>>

```
editor:  <<Link from: sifDoc >>
(#
   sdeC: @ sdeController;
   sdeController:< <<... ObjectDescriptor ...>>

   sde: @ sdeModel;
   sdeModel:< <<... ObjectDescriptor ...>>
#)
```

Fig. 7: Documentation links

One example of the usefulness of this kind of link, is the relationships between the technical program documentation describing an implementation and the corresponding source code. In the documentation there will often be references to the source code or pieces of source code at different abstraction levels might have been inserted. Conversely there might be references to the documentation in the source code or documentation pieces are inserted (often as comments). A major documentation problem in existing development environments is the lack of support for such relationships. In the hyperobject system links are created instead of inserting (copying) the related piece of program or documentation. The abstraction level of the link points can also be specified.

A special problem is maintenance of consistency between various documentation fragments mutually and consistency between documentation fragments and program fragments. Documentation links are visual in the editor both at the link source and at the link destination, and the user is notified if modified fragments contain links to other fragments.

7 The Basis

7.1 Nodes

The node types of the hyperobject system are groups, fragments and text objects. Because the starting point of the hyperobject system is a grammar based programming environment, the focus is on the structural representation of documents. The basic node type is the fragment.

Each fragment contains the internal representation of the document. The internal representation of program fragments, and the integration medium of the tools in the environment (editor, checker, coder, debugger etc.), is abstract syntax trees - ASTs. The editor [Borup & Sandvad 88] creates and manipulates AST's through the metaprogramming system that provides a programming interface to ASTs. The metaprogramming system models ASTs as objects i.e. the grammar hierarchy is modelled by means of an inheritance and aggregation hierarchy of objects [Madsen & Nørgaard 88].

7.2 Links

Links are characterized by type, direction, source and destination. Depending on the concrete hypertext system the link source and link destination can either be the entire node or a point or a region in the node. A link point normally refers to a position in a text, whereas a link region refers to a set of contiguous characters. If graphical node types are supported link points and link regions can refer to points or areas in a picture, respectively.

The link types in the hyperobject system are program organizational links, abstract presentation links, annotation links, program semantical links and documentation links. Documentation links can be further typed by labelling them in a systematic way.

Most links in the hyperobject system are bi-directional.

In the hyperobject system, link sources and link destinations of a fragment are expressed by mainly two references: a reference to a fragment and a reference to an AST-node in the fragment. Because the internal representation of a fragment is an AST there is no distinction between link points and link regions. Any point in an AST is the root of a subtree that in turn corresponds to a region. A third reference is used in the cases where the link point is at a finer level than an AST-node. This is often the case in documentation fragments. A documentation fragment has a certain logical structure that is reflected in the underlying AST, but a great part of a documentation fragment consists of free text, for example a paragraph. In these cases a point or an area is specified inside the text object, as known from other hypertext systems.

Link sources and link destinations of a text object are expressed by two references: a reference to a text object (either internal or a file), and a point or an area inside the text object.

7.3 Persistency and Presentation

Fragments are persistent objects. Persistence and presentation are important qualities in a hypertext system. Conklin describes hypertext as follows: "Windows on the screen are associated with objects in a database, and links are provided between these objects, both graphically (as labelled tokens) and in the database (as pointers)". Hypertext nodes must have a visual presentation and they must be able to survive between working sessions. In the Mjølner BETA system program fragments and documentation fragments have these qualities. The presentation is provided by the integrated text and structure editor.

Currently only abstract syntax trees are persistent objects, but it is planned that some kind of persistence will be supported in BETA in the future. This opens for further experiments with the hyperobject system. In the current system, links are not modelled as objects, but merely as references between objects. Links are "inserted" in the objects, they are not separate objects. If links were modelled as separate objects with attributes like identification and references to source and destination nodes, and link types were organized in inheritance hierarchies, it would be possible to express any kind of links between any kind of objects, thus providing a more general hyperobject system. Modelling links as separate objects would furthermore give the possibility of having multiple views (links) on the same network of objects.

7.4 Browsing

The nodes of a hypertext system can be browsed in three ways: (1) by navigational access: the links are followed and windows are opened successively, (2) by querying access: a hypertext network is searched for a string, keyword or an attribute value (of a node or a link), (3) by direct access: a hypertext network is presented in a two-dimensional graph and nodes are accessed by selecting nodes in the graph. The term browser is often used for this interactive graph.

In the hyperobject system browsing is done by navigational and direct access.

8 Related Work

8.1 NoteCards

NoteCards [Halasz 88] is a general purpose hypermedia system, but it was originally designed to be a tool for idea processing and authoring in research environment. The system provides the user with a network of electronic note cards interconnected by typed links. The basic node type is a note card. A note card can be a piece of text, a structured drawing or a bitmap image. New card types can be added. Links are typed by means of a user-chosen label. Link sources are points but link destinations are whole cards. Collections of note cards can be organized or categorized into specialized cards called fileboxes, that corresponds very much to UNIX file directories. Another specialized card is the browser that contains a structural diagram of a network of note cards. The browser provides direct access to nodes in a hypertext network. NoteCards has a high degree of extensibility and tailorability [Trigg et al. 87]. It is integrated with the Xerox Lisp programming environment, it provides a programming interface that allows the user to create new card types, and the user is provided with a set of parameters that can be used to adapt the system. In addition cards and links can be processed under program control.

The basic difference between NoteCards and the hyperobject system is the emphasis on text and graphics versus program or documentation structure.

The point to point links of the hyperobject system are more general, because the destination points can be inside documents. The abstract syntax trees define a hierarchical organization of hypertext nodes. In fact an AST can be considered as a tree of note cards, where the text leaves of the AST corresponds to a note card.

Concerning tailorability in the hyperobject system, new node types are created by specifying a grammar for the particular language used in the new document type.

The hyperobject system, or more generally the Mjølner BETA System provides a programming interface and a set of notifications (events) that are called when important events occur in the system. Examples are link creation, link deletion and link following. These events can be caught in extensions of the system [5]. In fact, part of the hyperobject system has been developed by extending the original programming environment using this kind of tailorability. Tailorability in the Mjølner BETA System is discussed further in [Nørgaard & Sandvad 89]. Such notifications can also be used to support executable links.

8.2 DIF

DIF (Documents Integration Facility) [Garg & Scacchi 88] is a hypertext system which helps integrate and manage the documents produced throughout the life cycle of software projects. It was designed for use in the System Factory, an experimental laboratory created to study the development, use and maintenance of software systems. A hypertext of software information is built by the software engineers over eight life cycle activities. Each activity culminates in producing a document: requirements specification, functional specification, architecturial specification, detail design specification, source code document, user manual and system maintenance guide.

Organizational structure is provided by describing the structure of each document. All projects in the System Factory have the same structure of documents. Each document is described by a form, which is a tree

[5]The programming interface to the editor for example, is a BETA class. The editor can be extended and tailored by constructing a subclass that catches the events. Events are modelled by means of virtual procedures.

structured organization of basic templates (BTs) to be filled in with information. A form corresponds very well to a table of contents of sections and subsections. The task of the software engineer when documenting is to fill out these predefined forms. A BT is a file that can contain different kinds of information: informal text, formal specifications, graphical descriptions, C code or executable code. At the BT level, DIF has an interface to tools in the UNIX environment e.g. C compiler, Emacs-like language-directed editors and nroff/troff. Revision control is supported at the form level by means of the RCS system.

BTs constitute the nodes of the hypertext system. The user can define links between BTs across projects. Links define semantic relationships between existing nodes, but annotation links create new nodes. Links can be operational which means that if a link to an executable BT is followed the BT is executed. A set of keywords can be defined for each BT and attributes can be associated to links. The structural information (forms) and the semantical information (links) are stored in a relational database. This gives the user comprehensive querying facilities for retrieval of documents.

In the hyperobject system there is no predefined document types, but any of the forms in the System Factory could be supported by describing them in a context free grammar and generating a syntax-directed editor for the particular language used. The forms mainly describe the logical structure of documents using sections and subsections. All document types with this structure can be expressed in the same grammar.

Abstract presentation provides the same overview as a form. The individual sections could be in the same document at lower abstraction levels or they could be in separate documents connected by links.

DIF has advanced querying capabilities on nodes and links, in contrast to the hyperobject system . But the links in DIF are at a rather coarse level. In NoteCards terminology the links between BTs are global links.

In [Garg & Scacchi 88] it is not described how a form can organize a C program. It seems difficult to do that in a predefined way.

The main difference between DIF and the hyperobject system can be summarized in: In DIF the emphasis is on production of standardized documents that can be linked together and in the hyperobject system the emphasis is on finer grained organization and linking of program fragments and documentation fragments. The preferences of DIF is the multiproject management, revision management and querying facilities.

8.3 Neptune

Neptune [Delisle & Schwartz 86] is also a general hypertext system but it was designed with special attention to computer aided software engineering (CASE). Neptune is a layered architecture. At the bottom level is a transaction-based server named the Hypertext Abstract Machine (HAM). The HAM provides storage and access mechanisms for nodes and links. The HAM provides distributed access over a computer network, synchronization for multi-user access and a transaction-based crash recovery. Neptune is a layer of functionality on top of the HAM. The HAM provides operations for creating, modifying and accessing nodes and links. It maintains a complete version history of the so-called hypergraph. Any version of the hypergraph can be accessed.

There are no restrictions on the node type. The contents of a node is just binary data. A link source as well as a link destination is expressed by an offset within the contents of a node. A link can be attached to a specific version of a node or the "current" version of the node. Attribute/value pairs can be attached to nodes and links. Navigational and querying access are based on these attribute/value pairs. There are three kinds of browsers: a graph browser provides a pictorial view of a sub-graph of nodes, a document browser provides a hierarchical view of nodes (much like the Smalltalk browser) and a node browser that is a text editor, where links can be created and followed. One specialized browser is a node difference browser. Neptune also generates events, i.e. calls user defined procedures written in Smalltalk, C or Modula-2.

When using Neptune for CASE applications two aspects have gained special attention: organizational relationships and program semantical relationships.

The hierarchical structure of documents (sections and sub-sections) is expressed using a node to represent each section or sub-section with links connecting each node to its immediate descendant sections or sub-sections. The hierarchical structure of program source code is expressed in a similar way. For example a Pascal program can be represented as a tree with a node for each procedure or function. In a language like Modula-2 the program requires a directed graph of modules.

The attribute/value pairs are used to organize/categorize nodes and links. An example of a node attribute could be 'projectComponent' with the values possible values: requirement, specification, design, comment, source code, object code, symbol table and documentation. An example of a link attribute could be: leadsTo, comments, refersTo, callsProcedure, implements and isDefinedBy.

The preferences of Neptune are distributed multi-user access, version management of the hypergraph and querying facilities based on attribute/ value pairs of nodes and links. Neptune has better support for finer grained relationships than DIF, because link points are defined as a node plus an offset within it. However when it comes to support for program semantical relationships the basis of text documents and links in a relational database does not suffice. The variety of program semantical relationships that are expressed by means of link attributes like callsProcedure and isDefined must be stored in the database. Work is currently going on for automating this process, but the database approach seems not to be appropriate for this kind of finer grained relationships. The structural representation of program fragments in the hyperobject system is better suited for that purpose, because links are inserted in the abstract syntax trees.

Neptune organizes documentation as well as programs in trees or graphs of nodes. In the hyperobject system the fragment system and abstract presentation are used to organize the structure of documents. The grammar based approach has several advantages over the Neptune solution. Abstract presentation can be generated automatically. Violations to a documentation standard or to the grammar of a programming language are prevented because documents are edited structurally.

9 Conclusion

It has been demonstrated how ideas and concepts from hypertext systems can be applied to a programming environment. The qualities of hypertext systems have shown to be useful for programming environments too, but if the programming environment is grammar based it is possible to go a step further. The emphasis on structure, based on the grammar of documentation languages as well as programming languages is one of the most important contributions of the hyperobject system.

Finer grained relationships between program fragments are very useful in programming. Due to the grammar basis it is possible to automate the definition of such links and to represent them in a convenient way.

The uniform treatment of documentation and programs means that the same facilities are available for both kinds of documents: syntax-directed editing, abstract presentation and (hyper)linking facilities. The hyperobject system provides good support for navigation in large programs and their corresponding documentation.

The object-oriented approach, of modelling the abstract syntax trees of fragments as object hierarchies and modelling links as object references, has been very suitable for expressing relationships between documents.

Status and Future

The Mjølner BETA System is currently an industrial prototype [6] , i.e. it can be used for practical purposes by motivated users. It needs some finishing to become a robust product.

Some of the qualities of the hyperobject system described in this paper are provided by the current Mjølner BETA System and others only exist in an experimental version of the system. The support for documentation links is very experimental and only the logical structure of documentation fragments is fully supported. The support for linking text objects together has had low priority. Instead the focus has been on the structured objects. Finally only navigational access is currently supported.

Future work will focus on improving the support for documentation links, querying facilities, and graphical presentation of the hyperobject network.

[6]The Mjølner BETA System is implemented on SUN, APOLLO and will be ported to the Macintosh this year.

To get a complete documentation tool the facilities for syntax-directed document editing must be improved, essentially word processing facilities like format and style.

Acknowledgement

I wish to thank Kaj Grønbæk, Ole Lehrmann Madsen, Claus Nørgaard, Lars Bak Petersen and the anonymous referees for many helpful comments on this paper. This work has been supported by *The Danish Natural Science Research Council*, FTU grant no. 5.17.5.1.25 and the Mjølner project, that has been partly funded by a grant from *The Nordic fund for Technology and Industrial Development.*

References

[André 86]	J. André: GRIF plus MINT or how to abide by a layout-sheet. In: J. Miller (ed.): PROTEXT III Conference Proceedings, Boole Press, 1986.
[BETA 87]	B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language. In: B.D. Shriver, P. Wegner (ed.): Research Directions in Object Oriented Programming, MIT Press, 1987.
[Conklin 87]	J. Conklin: Hypertext: An Introduction and Survey. IEEE Computer, December 1987.
[Borup & Sandvad 88]	K. Borup, E. Sandvad: Users and Programmers Guide for Sif - A Syntax-Directed Editor. Project Mjølner Working Note DK-SYS-29.2, December 1988.
[Delisle & Schwartz 86]	N. Delisle, M. Schwartz: Neptune: A Hypertext System for CAD Applications. SIGMOD Record, vol. 15, no. 2, 1986.

[Halasz 88] F.G. Halasz: Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. Communications of the ACM, vol. 31, no. 7, July 1988.

[Hansen & Hestbæk 89] P.B. Hansen, B. Hestbæk: Trud - an Interactive Syntax Directed Document Editor. In: WOODMAN'89 Proceedings, BIGRE, May 1989 (These proceedings).

[Holdam & Nørgaard 86] J. Holdam, C. Nørgaard: Gipsy - a Grammar Based Interactive Document Processing System. In: J. Miller (ed.): PROTEXT III Conference Proceedings, Boole Press, 1986.

[Kristensen et al. 83] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Syntax Directed Program Modularization. In: P. Degano, E. Sandewall (eds.): Interactive Computing Systems, North-Holland, 1983.

[Garg & Scacchi 88] P.K. Garg, W. Scacchi: A Hypertext System to Manage Software Life Cycle Documents. In: B.D. Shriver (ed.): Hawaii International Conference on System Sciences - 21, IEEE, January 1988.

[Madsen & Nørgaard 88] O.L. Madsen, C. Nørgaard: An Object-Oriented Metaprogramming System. In: B.D. Shriver (ed.): Hawaii International Conference on System Sciences - 21, IEEE, January 1988.

[Mjølner 87] H.P. Dahle, M. Løfgren, O.L. Madsen, B. Magnusson (eds): The Mjølner Project, In: Proceedings of EUROSOFT '87, London, June 1987.

[Mjølner BETA 89] O.L. Madsen, C. Nørgaard, L.B. Petersen, E. Sandvad: An Overview of the Mjølner BETA System. Mjølner Informatics, Science Park Aarhus and Computer Science Department, Aarhus University, March 1989.

[Nørgaard & Sandvad 89] C. Nørgaard, E. Sandvad: Reusability and Tailorability in the Mjølner BETA System. Mjølner Informatics, Science Park Aarhus and Computer Science Department, Aarhus University, March 1989.

[Quint 83] V. Quint: An interactive system for mathematical text processing. Technology and Science of Informatics (TSI), vol. 2, no. 3, 1983.

[Sandvad 88] E. Sandvad: Syntax-Directed Graphical Editing. Computer Science Department, Aarhus University, June 1988.

[Trigg et al. 87] R.H. Trigg, T.P. Moran, F.G. Halasz: Adaptability and Tailorability in NoteCards. In: Proceedings of INTERACT '87, Stuttgart, Septemper 1987.