Structure - oriented editors

Report**Author(s):**

Burkhardt, Helmar; Nievergelt, Jürg

Publication date:

1980

Permanent link:

<https://doi.org/10.3929/ethz-a-000814166>

Rights / license:

In Copyright - Non-Commercial Use Permitted

Originally published in:

ETH Eidgenössische Technische Hochschule Zürich, Institut für Informatik 38

ETH

Eidgenössische Technische Hochschule
Zürich

Institut für Informatik

H. Burkhart , J. Nievergelt

STRUCTURE-ORIENTED EDITORS

STRUCTURE-ORIENTED EDITORS

H. Burkhardt and J. Nievergelt

Institut fuer Informatik
ETH
CH-8092 Zuerich

May 1980

Paper presented at the GI-Fachtagung
"Textverarbeitung und Informatik", Bayreuth (Germany),
May 1980

© 1980 Institut fuer Informatik der ETH Zuerich

STRUCTURE-ORIENTED EDITORS

H. Burkhart and J. Nievergelt
Informatik,
Swiss Federal Institute of Technology (ETH)
CH-8092 Zurich, Switzerland

ABSTRACT

Editors have evolved over a period of three decades from minor aids for preparing programs to perhaps THE major family of utilities available on interactive systems. In the process, editors for manipulating different types of objects, such as texts, programs, forms, pictures, have multiplied. In contrast to the development of programming languages, no editors have been widely accepted as de facto standards, and few general design principles have become common knowledge.

This paper presents a brief history of the development of editors, assesses the state of the art of editor design, and analyzes the common features of editors. The notion of syntax-directed editor is generalized based on the notion of "site": a common structuring device for all data in a system. As an example of these concepts, a family of structure-oriented editors designed for the interactive system XS-1 is described.

CONTENTS

1. Historical development of editing facilities
 - 1 Tools for program manipulation in a batch environment
 - 2 Time-sharing and the emergence of on-line editors
 - 3 Smart terminals, personal computers, and the resulting proliferation of editors
 - 4 The user's working environment today:
editors in the context of other utilities
2. Common features of editors
3. A family of structure-oriented editors
 - 1 Project history and goals
 - 2 The universal core of the system
 - 3 Document and program preparation
 - 4 Implementation aspects
4. References

1. Historical development of editing facilities

In order to understand why today's editors are as they are, and to free one's mind from the constraints suggested by tradition, it is useful to review briefly the short history of interactive manipulation of information stored in textual form. We sketch a qualitative picture that tries to explain the reasons that lead to overall characteristics, rather than presenting details; [VD 71] is a more technical survey of early editors.

1.1 Tools for program manipulation in a batch environment

From the beginning of commercial data processing through the early sixties, when hardware costs dominated software costs, the programmer did most of his work off-line, working with punched card decks and special equipment for handling cards. The card was his basic unit of information, the deck his basic package; he submitted it as a whole, to be run in a batch mode. In today's terminology, the card punch and verifier, with its tab-setting drum, was his editor, and the programmer himself was the filing system. Using color coding, diagonal lines drawn across the back of the deck, and sequencing numbers, he kept his deck in order. Manual access to a specific card in a well-organized deck of several thousand cards could often be done in less than a minute; many editors and filing system in use today are not as fast in practice, because the search criteria they allow do not make as efficient use of the human eye and its pattern recognition abilities as the old-fashioned card deck did. Insertions, deletions, permutations of card sequences are often done faster by hand than with editors or filing systems. This advantage outweighed the slight inconvenience of having to store and move card decks physically.

Thus the old card deck survives so long because it provides a good medium for manipulating sizable units of information, such as subroutines and data aggregates. The punched card is a poor medium, however, for handling tiny units of information, such as a character. Re-punching an entire card, key stroke by key stroke, because a comma was forgotten at the far left; or changing all occurrences of an identifier X by XX - these tedious activities can cause a dreadful waste of time.

The inconvenience of manipulating information within a card (as opposed to manipulating card sequences) gave rise to specialized "batch editors" - utility programs that accept "correction cards", and perform automatically the specified changes. To avoid excessive waste of cards, the card deck with the correction cards only serves as a back-up; card images of the updated version of the program are permanently stored on disk. Thus the programmer only rarely has to manipulate the physical deck of cards - and with its removal from his touch and sight,

he loses the powerful organizational facilities that it provides. A replacement had to be found: this need, plus the gradual replacement of batch by time-sharing systems that occurred during the sixties, set the stage for the second phase in the development of editors.

1.2 Time-sharing and the emergence of on-line editors

Time-sharing systems, that allow the use of a computer by many users at different work stations simultaneously, put the programmer back where he had been in the earliest days: on-line, but without the worry of wasting processor time while thinking at the terminal. Thus creation of new programs at the terminal became as feasible as editing existing programs. The batch editing utilities mentioned earlier were expanded into on-line editors. These tended to support the same line-oriented operations as the batch editors: replace, modify, insert, delete a line or a sequence of contiguous lines. A few structurally similar commands suffice for an interactive dialog that is constrained anyway by a line-oriented terminal such as a teletype: print line, move a line cursor up or down.

Such line-oriented editors tailored to teletype terminals matched the technology of the day but not the user's needs. Slow-speed devices such as a teletype often have a line-buffer; this amount of data is perceived by the user as "directly accessible", that is, changes on this data appear to be "immediate". This feature makes it difficult or exceedingly slow for the programmer to see his recent changes in the context of an entire page - he typically sees the one line being modified, and keeps a mental map of its neighborhood. In view of this inconvenience it is surprising that line-oriented editors have survived to this day.

As long as texts to be edited have a natural line structure, such as source programs in assembly language or Fortran, one learns to live with line-oriented editors. With block-structured programming languages, or with free-form texts such as documentation, the restriction of the user's "working space" to a single line becomes exceedingly inefficient. The spreading of alpha-numeric display terminals with higher transmission rates and/or larger buffers paved the way for the third stage in the development of editors - the phase of proliferation .

1.3 Smart terminals, personal computers, and the resulting proliferation of editors

Our account now lumps together several developments that should perhaps be discussed separately. They all happened at about the same time, however, during the decade of the seventies. These trends continue unabated today; we must strive to channel them

into a more disciplined evolution than has been the case in the past.

Several reasons concurred to change the status of editors from that of one utility program among many, to a software component of equal importance to operating systems and compilers. Economics provided the major reason. Low cost computers (primarily memory), drastically widened the range of activities performed on a computer. Text processing, computer graphics, computer-aided design, computerized message systems - all these activities require editors, and not just one per application: a picture editor, for example, must be tailored to the specific class of pictures to be manipulated with it.

A second reason for the increasing importance and variety of editors is the rapid improvement in terminal hardware. Large buffer memories and local processing in the terminal give the user access to as much of his data as he can work on at one time, as fast as his eyes can follow. Graphics, user-definable fonts, color - these increasingly popular features make computers applicable to all kinds of documents, not just text.

Terminals attached to a central computer are giving way to stand-alone, single user computers, where the screen is refreshed directly out of central memory. The large bandwidth of this communication channel makes fast animation possible, even in several independent windows that the user may define to see different aspects of his data. Smalltalk [GO 76] is a well-known example of this trend.

The increasing convenience of a terminal or personal computer as the user's "work desk" - thinking and creating rather than just executing pre-planned actions - has the consequence that the user is most often in an editing mode. [MI 77] measured that up to 75% of all activities in an interactive system are editing operations. Thus editors are among the most used software components of an interactive system, on a par with operating and filing systems, but ahead of compilers.

Since the entire structure of an editor is visible to the user (which is not the case with compilers or operating systems), it is not sufficient that it "just works" - it must work in a clean, understandable way. This is rarely the case today, when most editors in use have grown in an ad hoc fashion; this usually leads to a sufficient set of commands, but a lack of structure. Since the need for editors is great and the shortcomings of current ones are fairly clear, editors may become one of the major areas of software research in this decade.

Some recent papers give an indication of the direction this research is likely to take. [FR 80] proposes a generalized text editor that can be specialized to different classes of data. The editor is separated into a data-independent, front-end display

manager that calls on a set of editing primitives that actually edit data. The common structure imposed on all objects is a textual one.

1.4 The user's working environment today: editors in the context of other utilities

Consider writing a long report on your computer. You will be using a text editor, a formatter, the filing system; perhaps specialized editors for programs or pictures. The interface between all of these utilities is the data that is manipulated by each of them, and must be passed from one to the other. How well is this interface engineered for the user's needs?

The traditional editor works on a restricted set of data - a file (or perhaps two files), which is a unit of the filing system. This unit becomes the user's data environment while he is in the editor - that is, he has to be aware that his actions affect this particular set of data, not a smaller or a larger one. In our example of report writing, the logical data unit from the user's point of view is the entire report. The report will be structured into subunits such as chapters, paragraphs, sentences, words. How can the logical structure that the user imposes on his text be mapped onto the structure and units provided by the utilities, such as files, segments of files, lines, characters?

Today's user can typically choose among two unsatisfactory alternatives: either he puts the entire report in one file, thus relying entirely on the editor for all data manipulation, or he splits the report among many files, preferably along chapter and section boundaries.

In the first case, the question arises whether the editor sufficiently supports meaningful structures that the user would like to impose on his data. Text editors begin to be aware of words and sentences, thus making it unnecessary for the user to point in each instance to the beginning and end of the corresponding character string. They rarely support higher structures such as paragraphs, sections and subsections, or collections of documents. "Section 1.4 should become an Appendix", with all the organizational consequences this implies, will require the user to view on the screen and point to all the places where a change occurs. Perhaps a formatting program can be more helpful for this task, but this implies that the user has to learn to cope with yet another utility. In the second case, the first question is whether the filing system supports structures. One-level name spaces, which mix all files in a linear directory, are unsuitable; hierarchically structured directories are better. In either case, edit operations and filing system operations are not tied together, and force the user to juggle back and forth between these two utilities: introducing a new chapter in the edit mode does not

automatically update the directory and the partition of the text into files.

We can summarize this situation as follows: the first choice (entire report in one file) leads to a simple user interface (only the editor is involved) but an insufficient structure in the data environment; the second choice (many files) may provide sufficient data structuring facilities, but leads to a complex user interface. The complexity increases when we consider the interplay not only of text editor and filing system, but other utilities as well.

In conclusion: the adequacy of the today's systems is limited not so much by the shortcomings of any one editor, but by a lack of standards that all editors and utilities in the system must conform to. Such standards are particularly important with respect to data description and manipulation; but they are also necessary at the command level - for example, in a unified way of aborting erroneous commands, or referencing data of various types.

2. Common features of editors

In tracing the development of editors we have seen on the one hand the necessity of having many editors for different types of objects, on the other hand the undesirability of exposing the user to many different dialog contexts. Is there a way to reconcile these apparently contradictory design requirements?

We think there is, and in section 3 present a design for a family of editors (including an editor that plays the role of a traditional filing system) all of which converse with the user in a uniform way, using the same concepts and notations, regardless of the type of objects they deal with. Thus the user perceives the complexity of the entire family of editors as being only moderately greater than that of a single one. This section presents an intuitive justification of why this is possible.

The commands of most editors can usefully be grouped into half a dozen categories. The most important of these are described below.

Data reference:

Identification of an existing object (primitive or structured) in the data collection accessible to the user; for the purpose of performing an operation on it. Names, addresses, line numbers, pointing with an input device - these are some of the common referencing mechanisms.

Motion:

Change of data environment, i.e. of objects that are affected

by subsequent commands. Cursor motion, window setting, get or attach a file, incrementing an index, following a pointer, - are frequent examples of motion commands.

View of data:

Controlling the way an object is displayed. Scrolling, zooming, change of font, color or brightness, elimination of hidden lines are examples of the commands that usually do not change the objects themselves, but only their temporary appearance on the screen.

Management of data objects or storage:

Operations on attributes that are attached to objects, but do not depend on the type of object, e.g.: set protection status, kill an object, create a name or allocate space for a new object, make a copy or move an object to another work space. (Notice: the implementation of a copy routine may depend on the type of object, but the user's perception need not).

Operations on command sequences:

Extend the command language, or redefine the meaning of command sequences. Command macros, executing a profile at log-in time, setting status variables that alter the system's behavior are familiar examples.

Last but not least, in fact the very purpose of editors, is to provide useful

Operations on data objects!

This last category is totally dependent on the type of object being manipulated. Spelling check or hyphenation of text, syntax check of a procedure, rotation or dilation of geometric objects - this is where editors do and must differ.

The description of an editor correctly emphasizes the operations it provides for manipulating objects, since that is its reason for existence. This leads to the superficial conclusion that a text editor and a picture editor have little in common, since operations on text are unrelated to operations on pictures. In fact, the two editors may well share the majority of their commands, in two distinct meanings: a static count of commands listed in the manual, and a dynamic count of the frequency with which they are used.

The key to making these "overhead operations" similar for all editors, regardless of the domain of objects, is a universal structure defined on all data within a system. The next section shows how this can be achieved in practice.

3. A family of structure-oriented editors

We describe the structure and current state of implementation of XS-1 - an interactive system whose design is based on the concepts mentioned above. The structural dependencies of data of ALL types (currently text, programs, pictures) are reflected by a common structure: A tree whose nodes represent subsets of data ("sites"). The user decides how deeply he wishes to superimpose this universal structure upon his data. To the extent that he does, the common structural operations provided by a tree editor apply, and take care of many of the overhead operations required while editing. Specialized editors for each type of data can be restricted to data-specific operations that actually modify data. Since all of them share the same structural core, they form an integrated family of editors.

3.1 Project history and goals

The conceptual framework introduced above has been strongly influenced by the experience in the design and use of the interactive system XS-0 [NI 77,78]. The primary goal of that project was the development of an integrated system for teaching programming. CAI lessons provide instruction, and a programming system intended to be self-explanatory allows the student to write and run programs, and interact with running programs at the source language level. A simple command language was needed to allow novices to learn to use the system on their own. The design concepts of XS-0 have proven to be valid. The system has been used in introductory courses at ETH and at a secondary school.

XS-0 also served as a test bed for research in interactive systems. A variety of special purpose editors were written to enhance the system's utility for text processing. At this point we decided to redesign the entire system from scratch, and to address the question: How should a personal computer ideally behave?

3.2 The universal core of the system

3.2.1 A data space based on a tree structure

Whatever type of data a user wishes to process (e.g. text, programs, pictures), as soon as the amount of data exceeds a certain threshold, the user tends to structure his data - into chapters and paragraphs, procedures and modules, subpictures and patterns, as the semantics of the data may suggest. At any moment the user works only on a restricted data collection (e.g.

a paragraph) and needs only a subset of all operations present in the system (e.g. the text editor commands). Convenience as well as safety are enhanced if the system explicitly makes the user aware of his current active data and command environment. We chose a hierarchical data model - a tree superimposed upon the data - as a common structuring device throughout the system. A node in the tree represents a "site", that is, a collection of data which is the active data environment when the user is at this node. As the user gets deeper into the tree his active environment gets smaller.

Since data access is controlled by position in the tree, the user must be able to traverse the tree easily and quickly. A universal motion facility called EXPLORE provides commands that are active at all times:

- Relative motion commands for moving up, down, right or left in the tree allow the user to extend, restrict or change the scope of data that can be accessed.
- Absolute motion to any node that can be identified (for example by means of a name) allows a complete change of the data environment.

The user's question "where am I?", meaning "what data are affected by my actions?", is answered at all times by displaying an identification of the current site on the screen, and by making a map of the vicinity available at the press of a key.

3.2.2. The tree editor - a universal tool

If the tree is to serve as a useful skeleton for organizing all data, it must be highly dynamic; if data-reference and data-movement operations in all types of editors are to look alike, it is best to concentrate them in one shared facility, a flexible tree editor. The tree editor is used for editing the data space at the site level. Since a site can contain any amount of data, from a single character to all the data in the entire system, the tree editor incorporates services that are spread over several utilities in conventional systems.

The set of operations of the tree editor is shown in Table 1. It was not designed to be minimal (although it is implemented by means of a minimal set of four basic tree operations), but rather to be convenient for frequently occurring structural changes such as moving procedures in a program text. The following principles were observed:

PRESERVE CURRENT SITE: All operations with the exception of explicit motion commands leave the user's current site unchanged. This can be achieved naturally by defining all operations (e.g. copy, delete) so that their operands are necessarily successors of the current site.

PRESERVE SYNTACTIC CORRECTNESS OF TREE: XS-1 subtrees can be forced to obey syntactic constraints given by explicit tables (for example, a subtree that structures a source program must obey the syntax of the programming language). After completion of any operation the syntactic correctness of the tree must be guaranteed.

USE PROVEN TECHNIQUES: Experience with the use of many editors shows that a few concepts and techniques are universally useful. Some of those that we have used are:

- It must be possible to identify ANY location which is an argument of an operation by pointing to the screen. A cursor pointing to a node identifies the entire subtree (and its data) of which it is the root; for inserting a subtree the cursor points to a "gap".
- The user must be able to choose, among several views of the same data, the most convenient one; a tree structure, for example can be shown by drawing various types of pictures, or by means of indentation. Since many realistic structures do not fit onto the screen, view selection and zooming are important operations.
- Data transfer is an operation that requires a source and a target. This appears to conflict with the demand that the user is always at one site. In order to avoid error-prone "remote effects" (i.e. a command affects data which is not at the user's current site) an internal buffer is used: it can be loaded at one site and unloaded at another.

Operation	Effect on tree
INSERT	Insert a new site (with empty data collection) into a specified gap
DELETE	Delete a subtree (nodes and data)
COPY	Copy a subtree to a specified gap
MOVE	Move a subtree to a specified gap
SPLIT	Split a node and its data into two
MERGE	Merge two nodes and their data
EXPAND	Insert an intermediate level in the tree
SHRINK	Delete an intermediate level of the tree
ORDER	Permute the nodes on a tree level

Table 1: Tree editor commands

3.3 Document and program preparation

3.3.1 Design principles

DOCUMENT SYSTEMS MUST PROVIDE HIGHER STRUCTURES
THAN TODAY'S TEXT SYSTEMS DO

The best text processing systems on market support low-level textual structures such as words, sentences, paragraphs, footnotes and headings, figures and forms, lines and pages: automatic word wraparound, page headers, formatting options, are examples of standard services. Today's systems are of little help, however, for maintaining an overall view of a document in the process of being created, and for managing the dynamic interrelations of various text parts.

This lack of emphasis on higher structures is surprising in view of the fact that the author's need for explicit representation of semantic relationships was recognized over a dozen years ago, and implemented in prototype systems (for example at Stanford Research Institute, [EN 68]). One explanation for the lack of influence of these pioneering systems is that arbitrary structures such as Hypertext ([NE 67], [CA 69]) are costly to implement; another, that business and scientific literature typically requires a more disciplined structure than an arbitrary semantic net.

For these reasons, XS-1 imposes a tree structure on its documents. This provides access, manipulation and understanding of data on a higher level of abstraction. The structural operations of the tree editor get text-specific interpretations like "Insert a new paragraph", "Split a chapter into two", "Merge two documents".

We agree with "Make it similar to something the user already knows" [SN 78]: e.g. getting a first impression of a document is normally done by looking at the table of contents; in our terminology this means viewing the text at the level of sites.

INTEGRATE DATA ORGANIZATION, EDITOR AND FORMATTER

Data organization, editor and formatter - the major components of a text processing system - must appear to the user as one integrated facility, the document-edit mode. In today's systems these components have been designed in isolation which entails several disadvantages: the separation of data organization (particularly at the file level) and editor has the consequence that the data cannot be viewed conveniently in its semantic context; the separation of editor and formatter has the consequence that the final form of the output cannot be seen at edit time, but only after a time-consuming cycle of changing the format specifications, feeding the output processor, and looking at paper output.

DOCUMENTS CONTAIN BOTH TEXT AND GRAPHICS

Today's document processing systems do not support pictures integrated into the text. The user specifies empty spaces to be filled in by hand. Thus the document data can be seen in its full context only in the final version. This inability to enter pictures at the time he is writing the matching text often leads the user to omit them completely. XS-1 encourages the use of graphics by providing a submode "graphics editing" within the document system.

3.3.2 Mode structure of the document system

The utilities mentioned so far as well as some others are presented to the user in the form of a tree of submodes as shown in Fig.3.1.

Fig. 3.1 The mode tree of the document system

- Archive: Store document on a backup storage device adding a descriptor (author name, date, keywords etc.)
- Find: Search for relevant documents by descriptor or full-text search.
- Print: Create hard copy output; this usually invokes a depth-first traversal of a subtree, printing the data attached to nodes.

3.3.3 The text processing part of the system

TEXT EDITOR: The text editor operates on a logical window, a

rectangular area that fits entirely onto the screen). All operations are interpreted relative to a text cursor that can be moved within this window. Text editor operations are restricted to the level of characters, words and sentences (the mode-specific data level), since higher structures are handled by the tree editor. The command language was designed to be "orthogonal" that is, any operations (e.g. DELETE) is defined on all the structures recognized by this mode. This makes the operation set powerful and the command language consistent.

Orthogonality reduces the effort needed to learn a language: if the user knows the terms "DELETE", "WORD", and "<-", he can deduce that the combination of these means "delete the word to the left of the cursor".

Frequent operations are invoked by a single key press on a separate function key-block.

The "line" does not have a dominant role. In the word-wraparound mode the user can ignore line ends: if a word doesn't fit onto a line it is moved to the beginning of the next. If this line also overflows, the process continues until either the overflow text fits onto a line or a paragraph end is reached. Word deletion causes the reverse ripple effect.

The editor is parametrized in the sense that options like wraparound, insert-or-replace character, read-only or modify modes can be switched on or off.

FORMATTER: The formatter performs output transformations of the data; it affects only the view, not the content. There is no post-processor that interprets in-line code to produce nice output on the printer. Formatting is done interactively - the effect can be seen immediately on the screen. Formatting information can be stored permanently in the form of a descriptor attached to the site, and interpreted on demand when output has to be reproduced. Separating text and formatting information has the advantage that the same text can easily be output in different formats.

The formatter observes the structure of the text: for example, if a site is tagged with an "adjust right border" command, this holds for the whole subtree represented by the site.

3.3.4 A syntax-oriented program editor

Conventional programming systems include one or more general purpose text editors and compilers. Despite the fact that highly structured objects, source programs, are passed between them, their interface is usually a low-level structure, namely a long sequence of characters. The compiler has to reconstruct a syntactic structure that was present in the user's mind when he entered his program. The historical development (see 1.1)

explains how this situation came about: when compilers became widely used, punched card equipment was mainly used for entering programs, and this allows only minimal consistency checks. Researchers recognized early [HA 71] that a program editor should be aware of the syntax of the language, and help the user at the earliest possible moment to recognize trivial errors such as mismatching BEGIN-END pairs.

The XS-1 program editor is syntax-directed: The tree of sites is used to represent the syntactical structure of a program, where the type of each site corresponds to a nonterminal symbol of the underlying grammar. The production rules of the grammar are represented in the form of a syntax tree.

Operations on the structural level may affect the syntactic correctness of a program: e.g. moving a procedure to another part of the program, or deleting variable declarations. The latter example shows that for most of today's programming languages a complete consistency check requires a global analysis (type check) of the program.

Just as the integration of editors and formatters have drastic consequences on the design of a system, so does the integration of editors and compilers. Ideally, an incremental compiler that supports interactive programming is the logical conclusion.

3.4 Implementation aspects

3.4.1 Hardware configuration

XS-1 is being implemented on widely used minicomputer hardware components. The current configuration consists of a network of 3 DEC LSI-11/03 micro computers, each with 28K 16-bit words of central memory. There are 3 disk drives in the network: a 20 MByte PERTEC disk with 6 fixed and 2 removable logical disk drives, and two DEC RL-01 drives with 5 MByte cartridges. Each processor has fast access to one disk, and slower access to the others (for backup and sharing of files among users). The graphics terminal currently used is a HP 2648A with a 24 x 80 character screen and a separately stored graphics screen of 720 x 360 points resolution. A higher performance experimental graphics terminal is under development.

An LA-36 DEC-Writer and various teletypes are used for text output. We are currently looking for low-cost graphics printers.

3.4.2 Implementation language

The software is implemented in the high level programming language MODULA-2 [WI 78], a descendant of Pascal. A 5-pass compiler runs on and produces machine code for the PDP-11. MODULA-2 supports the use of modules, which permit controlled

access to objects and routines, and can be compiled separately. Modules are particularly useful in a sizable software project such as XS-1, where several programmers cooperate, some of them only transiently.

3.4.3 System structure

The software system can be usefully divided into 4 layers:

Fig. 4.1 System structure

Layer 0 represents the low level operating system characterized by disk block and console byte transfer.

Layer 1 falls into 2 parts:

- The "Command/Response Manager" implements the centralized management of commands (input) and responses (output). It guarantees that all modes converse with the user in a consistent way; for example, exiting from a mode or aborting a command are always started by the same key press and cause the same system reaction.
- The second part is concerned with the implementation of the structured data space. The XS-1 tree is a directory with thousands of nodes, some of which have large data collections, others only tiny ones. This range imposes strong demands on the efficiency of implementation, both with respect to time and space requirements. In particular it is critical that the "small sites", the nodes with few data, are implemented efficiently. Since many sequential operations on data cause tree traversals, small sites must be allocated contiguously on the disk whenever possible. When sites are in central memory, it is important to code the structural information so that the space overhead is small. The tree is encoded as a binary tree (each node points to its first son and to its right brother), requiring one 16-bit machine word per node. Such a "bare" node further contains a type and pointers to a descriptor (including its name) and data.
Module "Tree Page" Manager partitions the entire tree into appropriate pages that are stored on disk blocks. The "Accessible Tree" Manager implements the access and modification of the nodes currently accessible. It supports several windows into the tree; these are used, for example, when copying a subtree to another place, or when returning to a previously marked environment. Finally, module "Visible Tree" Manager further restricts the environment to that of the tree display and offers different views.

Layer 2 represents the dialog control level. The dialog state of a user is defined by a pair (site/mode). "Trails" [NI 79] - visible and editable time sequences of dialog states - are introduced to define operations on the dialog.

Layer 3 is that of data-specific modes. A mode requires two specifications to be given: The first concerns the description of DATA to be defined and manipulated at the site level. This is done by the specification of data types and corresponding production rules. The other concerns the description of the INTERACTION with the user. This is done by

describing the commands, the mode structure imposed, and responses provided - all according to rules specified by the interface between EXPLORE and the Command-Response Manager.

The software described is under development. A prototype consisting of the tree core, the free-form text editor, and a syntax-directed editor is running.

4. References

- [CA 69] S.Carmody,W.Gross,Th.Nelson,D.Rice,A.van Dam:
"A Hypertext Editing System for the /360"
in "Pertinent Concepts in Computer Graphics",
ed. M.Faiman and J.Nievergelt, Illinois Press,
(March 1969), pp.291-330
- [EN 68] Douglas C. Engelbart and William K. English:
"A research center for augmenting human intellect"
Proc. FJCC, Vol.33, Pt.1, AFIPS Press, Montvale, N.J.
(1968), pp.395-410
- [FR 80] Christopher W. Fraser: "A Generalized Text Editor"
Communications of the ACM, Vol.23, No.3, (March 1980)
pp. 154-158
- [GO 76] Adele Goldberg and Alan Kay: "SMALLTALK-72 Instruction
Manual "
SSL76-6, XEROX Palo Alto Research Center, (March 1976)
- [HA 71] W.J. Hansen: "User engineering principles for interactive
systems"
Proc. FJCC, Vol. 39, AFIPS Press, Montvale, N.J.,
(1971), pp. 523-532
- [MI 77] Lance A. Miller and John C. Thomas Jr.: "Behavioural
Issues in the use of interactive systems"
Int. J. Man-Machine Studies, 9 (1977), pp. 509-536
- [NE 68] Theodor H. Nelson: "Getting it out of our system"
in "Information retrieval: critical view"
ed. G.Schecter, Thompson Books, Washington, D.C.
(1967), pp. 191-210
- [NI 77] J.Nievergelt,H.P.Frei,H.Burkhart,Ch.Jacobi,B.Plattner,
H.Sugaya,B.Weibel,J.Weydert: "XS-0: A self-explanatory
School Computer"
Berichte des Instituts fuer Informatik/Nr.21, (Aug.1977)
Eidgenoessische Technische Hochschule Zuerich
- [NI 78] J.Nievergelt,H.P.Frei,H.Burkhart,Ch.Jacobi,B.Plattner,

H.Sugaya,B.Weibel,J.Weydert: "Interactive System for PASCAL programming" in PASCAL, ed. H.W. Wippermann, Carl Hanser Verlag Muenchen Wien, (1978), pp. 13-24

- [NI 79] Juerg Nievergelt and Jean Weydert: "Sites, modes and trails: Telling the user of an interactive system where he is, what he can do, and how to get to places" Berichte des Instituts fuer Informatik/Nr.28, (Jan.1979) Eidgenoessische Technische Hochschule Zuerich
- [SN 78] James Sneeringer: "User-interface Design for Text Editing: A Case Study" Software - Practice and Experience, Vol. 8, (1978) pp. 543-557
- [VD 71] Andries van Dam and David E. Rice: "On-line Text Editing: A survey" Computing Surveys, Vol.9, No.3 (Sept. 1971), pp. 93-114
- [WI 78] Niklaus Wirth: "MODULA-2" Berichte des Instituts fuer Informatik/Nr.27, (Dez. 1978) Eidgenoessische Technische Hochschule Zuerich

Acknowledgement:

B.Plattner, J.Stelovsky, H.Sugaya and J.Weydert are codesigners and implementors of the XS-1 system.
B.Boser, B.Frei, H.P.Giger, H.Thimbleby and F. Vonaesch have contributed to the software.

Berichte des Instituts für Informatik

- *Nr. 1 N.Wirth: The Programming Language PASCAL
- *Nr. 2 N.Wirth: Program development by step-wise refinement
- Nr. 3 P.Läuchli: Reduktion elektrischer Netzwerke und Gauss'sche Elimination
- Nr. 4 W.Gander, A.Mazzario: Numerische Prozeduren I
- *Nr. 5 N.Wirth: The Programming Language PASCAL (Revised Report)
- *Nr. 6 C.A.R.Hoare, N.Wirth: An Axiomatic Definition of the Language PASCAL
- Nr. 7 W.Gander, A.Mazzario: Numerische Prozeduren II
- Nr. 8 E.Engeler, E.Wiedmer, E.Zachos: Ein Einblick in die Theorie der Berechnungen
- *Nr. 9 H.P.Frei: Computer Aided Instruction: The Author Language and the System THALES
- *Nr.10 K.V.Nori, U.Ammann, K.Jensen, H.H.Nägeli, Ch.Jacobi: The PASCAL 'P' Compiler: Implementation Notes (Revised Edition)
- Nr.11 G.I.Ugron, F.R.Lüthi: Das Informations-System ELSBETH
- *Nr.12 N.Wirth: PASCAL-S: A subset and its Implementation
- *Nr.13 U.Ammann: Code Generation in a PASCAL Compiler
- Nr.14 K.Lieberherr: Toward Feasible Solutions of NP-Complete Problems
- *Nr.15 E.Engeler: Structural Relations between Programs and Problem
- Nr.16 W.Bucher: A contribution to solving large linear problems
- *Nr.17 N.Wirth: Programming languages: what to demand and how to access them and Professor Cleverbyte's visit to heaven
- *Nr.18 N.Wirth: MODULA: A language for modular multiprogramming
- *Nr.19 N.Wirth: The use of MODULA and Design and Implementation of MODULA
- Nr.20 E.Wiedmer: Exaktes Rechnen mit reellen Zahlen
- *Nr.21 J.Nievergelt, XS-0, a Self-explanatory School Computer
H.P.Frei,
et al.:

- Nr.22 P.Läuchli: Ein Problem der ganzzahligen Approximation
- Nr.23 K.Bucher: Automatisches Zeichnen von Diagrammen
- Nr.24 E.Engeler: Generalized Galois Theory and its Application to Complexity
- Nr.25 U.Ammann: Error Recovery in Recursive Descent Parsers and Run-time Storage Organization
- Nr.26 E.Zachos: Kombinatorische Logik und S-Terme
- *Nr.27 N.Wirth: MODULA-2
- Nr.28 J.Nievergelt, Sites, Modes and Trails: Telling the User of an
J.Weydert: Interactive System where he is, what he can do,
and how to get to Places.
- Nr.29 A.C.Shaw: On the Specification of Graphic Command Languages and their Processors
- *Nr.30 B.Thurnherr, Global Data Base Aspects, Consequences for the
C.A.Zehnder: Relational Model and a Conceptual Schema Language
- Nr.31 A.C.Shaw: Software Specification Languages based on regular Expressions
- Nr.32 E.Engeler Algebras and Combinators
- Nr.33 N.Wirth A Collection of PASCAL Programs
- Nr.34 R.Marti, Meta Data Base Design - Consistent Description of
J.Rebsamen, a Data Base Management System
B.Thurnherr:
- Nr.35 H.H.Nägeli, Preventing Storage Overflows in High-level
R.Schoenberger: Languages
J.Hoppe: A Simple Nucleus Written in MODULA-2
- Nr.36 N. Wirth: MODULA-2
- Nr.37 Hp.Bürkler, EDV-Projektentwicklung - Ein Arbeitsheft für
C.A.Zehnder: Informatik-Studenten
- Nr.38 H.Burkhart, Structure-oriented editors
J.Nievergelt:

* out of print