

XS-1: An integrated interactive system and its kernel

G. Beretta, H. Burkhart, P. Fink,
J. Nievergelt, J. Stelovsky,
H. Sugaya, A. Ventura, J. Weydert
Informatik, ETH, CH-8092 Zurich

ABSTRACT

We present a case study of an eXperimental integrated interactive System, XS-1, being implemented on small computers. The primary goal of this project is to provide experimental support for design principles that have emerged from a critique of the behavior of today's interactive systems, and from our earlier implementations. These design principles apply at the user's level and at the system designer's level. The user must have a model of the system that allows him at all times to obtain information about his current data environment and his current command environment; he must be able to obtain such information by means of universal commands that are always active, regardless of the application program he is executing. These requirements at the user level have stringent implications at the system design level, such as: a common representation for data of all types, a file system that handles small and large sets of data in a uniform way, a front-end dialog processor shared by all application programs, and screen layout conventions.

CONTENTS

1. The concept of an operating system and its changing role
2. Deficiencies of conventional operating systems for interactive applications
3. Design principles: an interactive system as seen by the user
4. The kernel of an interactive system and its components
5. Implications for the applications programmer (mode writer)
6. Explore: Visualization of the system state, motion, and dialog history
7. The tree editor: A syntax-directed editor shared by all applications
8. Central dialog control: Getting all modes to talk the same language
9. Tree file system: Efficient management of small and large sets of data
10. State of project, experience, plans

KEYWORDS: *interactive systems, man-machine communication, system design, human factors.*

Current affiliation of authors: Informatik, ETH, CH-8092 Zurich, except:

H. Burkhart: Elektronik, ETH, CH-8092 Zurich

H. Sugaya: BBC Brown Boveri Research Center, CH-5405 Baden

1. The concept of an operating system and its changing role

The type of operating system required for today's dedicated interactive system is radically different from the conventional general-purpose operating system that has been developed over the past two decades. In order to justify this claim, it is useful to start with a brief review of the historical development of operating systems in general, and file systems in particular.

Utility programs such as loaders, dump and copy routines, I/O facilities, assemblers and linkers have been a part of the programming environment on the very first computers. Wilkes, Wheeler and Gill, in their classic 1951 text [WWG 51] on programming the EDSAC, make the subroutine library the basis of programming. On the early computers the task of coordinating and sequencing the actions of utility routines remained the responsibility of the programmer; thus he perceived the collection of these utilities as a "library" rather than as a "system".

As computers evolved during the fifties from experimental machines in the laboratory to production systems in computer centers, the programmer or operator turned out to be too slow a controller of the job flow through the machine. Simple "batch monitors" were written to automatically sequence from job to job and call the various system components, such as the loader, the assembler, or the compiler (just FORTRAN).

The development of I/O channels around 1960 and their ability to interrupt main frame activity led to multiprogramming and to correspondingly more complex coordination tasks. "Supervisors" permanently resident in "low core" evolved to handle all I/O activities and to pair them up with the user programs that had initiated them. During the same period of the early sixties, user demands concerning services to be provided exploded. It was no longer sufficient for the computer center to offer an assembler and perhaps a compiler: COBOL, RPG, Algol had joined FORTRAN. Moreover, the system had to be able to accommodate as yet unspecified compilers and application packages - it had to be open-ended.

Fairly suddenly, the notion of an operating system as a virtual machine (an extension of the basic hardware) and as an interface between the system and the user/programmer ("an operating system is much more a set of conventions than it is a piece of code", [Me 62]) emerged around 1960. The first published use of the term occurred perhaps in a 1961 description of Manchester University's "Atlas operating system" [Ki 61]. Shortly thereafter, IBM's choice of the name OS/360 put the stamp of approval on the abbreviation "OS".

The internal structure of operating systems remained a fruitful subject of research for two decades and evolved considerably. There have been significant advances in our understanding of processor and memory management, scheduling, concurrency, protection. But the interface to the user, namely, the applications programmer, changed little: it was frozen into job control languages that today embody the same concepts that had emerged in 1960, namely, the file as a sizable unit of data to be moved or copied, executed,

and operated upon by programs. Bell Labs' UNIX [BSTJ 78], widely acknowledged as a model of a well-designed operating system, clearly shows its nature as a file handling system in the conventional sense. Today this traditional notion of operating system no longer meets the requirements imposed by two applications that have gained in importance only recently: distributed systems and interactive systems. The first one forces a reconsideration of the core of an operating system due to changes in system architecture; the second forces a redesign of the user interface due to the fact that we have come to value user efficiency more highly than efficient use of hardware. Our paper discusses the implications of the second point.

For two decades the concept of operating system as an interface between user programs and what the manufacturer supplies remained almost unchanged. As we hope to show in the next section, it may be time to revise the prevalent notion of what an operating system's tasks are, at least for the increasingly important class of interactive systems (see also [De 82]).

The rest of the paper consists of two main parts. Sections 2 through 5 present an overview of the motivation, the goals of our project, and the approach we took. Sections 6 through 9 describe in greater detail the structure of the experimental system XS-1. The final section 10 briefly assesses the current state of the project.

2. Deficiencies of conventional operating systems for interactive applications

The recent proliferation of interactive systems challenges the conventional notion of an operating system whose development has been traced above. The user of today's interactive systems is usually not the applications programmer whose needs determined today's JCL's. He is an "end-user" operating a data entry or text processing system, or entering queries to a data base or information system. He is usually not a programmer; even if he is, the demands he imposes on an operating system as he is editing his program at the terminal are very different from those that a non-interactive program imposes when it runs. These differences are primarily of four kinds: timing, data access, data security, and support for user interaction.

Timing

Timing considerations that determined the design of conventional operating systems were governed by the goal of using hardware as efficiently as possible. In order to achieve this, user programs were interrupted whenever a hardware resource or a system program desired. For certain applications, such as process control, preemption of application programs is impossible. For these, a separate category of real-time operating systems developed; their functions are much more restricted than those of the general-purpose operating systems described above. A modern interactive system behaves like a real-time system: timing constraints are imposed by application programs rather than by a concern for efficient hardware utilization. And these constraints are just as severe as in many process control applications: a highly interactive system should respond

"instantaneously" to "trivial" user request. In the context of human physiology, "instantaneous" means one tenth of a second. In the context of human psychology, "trivial" means a task subjectively perceived as being so minute that the user experiences no sense of "closure": he feels that he has not yet completed any task or achieved any goal, and that he wants to proceed at his full speed. Examples of such trivial tasks are correcting a typing mistake, asking for the next screen full of text in a scrolling mode, or choosing an item from a menu.

Deficiency 1: Conventional operating systems are incapable of guaranteeing instantaneous response to trivial requests.

Data access

Data access patterns are much more random in interactive applications than they are in non-interactive ones. In the latter, the programmer has usually organized his data and layed it out in storage so that data references are clustered, i. e. the program's working set is small. We can load a sizable fraction of a file into central memory and expect that most data references will be confined to this set for a reasonable length of time. In interactive applications, on the other hand, data references are mostly determined by the user's working habits; when he browses through a large data collection, references will be scattered over the entire store, no matter how much care was spent in laying out the data. Although such browsing phases may be short as compared to the time he spends working intensively on a small set of data, such as editing one file, they are psychologically important; browsing is ineffective without immediate response, and the user loses his most powerful tool: the searching and pattern-matching abilities of his eye.

Deficiency 2: Data organization based on the assumption of predictable working set sizes leads to unpredictable response times in highly interactive systems. A storage structure is needed where the user may access chunks of data whose position in storage and whose size are selected at random.

Data security

Today's operating systems emphasize the protection of the user's data against misuse by the system or by other users. In an interactive system, however, the user is his own worst enemy: careless deletion of a file, overwriting information because of the wrong copy direction, forgetting text in a buffer that will not be saved - such mishaps are experienced by everybody. Techniques developed to protect the user from his own mistakes are few and primitive, such as the sporadic question *Are you sure?* The fact that the user is sure doesn't mean he is right. Or he may get into the habit of typing *YES* even before the system asks for this reassurance. What the user needs is a universal *undo* that reverses the dialog at least part of the way.

Deficiency 3: Conventional security measures are based on prevention, but an interactive system needs tools for recovery, such as a universal *undo*, and for restoring damaged data, such as a scavenger for patching files.

Support for user interaction

Conventional operating systems contain an I/O subsystem optimized with respect to efficient hardware utilization for large amounts of I/O, as it

typically occurs in batch processing. They tend to support interactive I/O at the low level of what all terminals have in common: lines of alphanumeric characters, with a varied assortment of control functions invoked via escape sequences. Several aspects essential to good man-machine communication are absent or insufficiently supported, such as: graphics; features that support screen layout, such as windows; features that support dialog control, such as a hierarchic structure of the user input into characters, parameters, commands, command sequences, or the provision of inputs that are active at all times, even while another input is being processed (see also [No 81]).

Deficiency 4: Conventional operating systems support I/O for batch processing and for line-oriented interactive terminals, but not for the sophisticated interface a modern interactive system has to present to the user.

Thus we conclude that an operating system for a highly interactive system must be substantially different from a traditional one. Our project addresses all 4 deficiencies mentioned above, but focusses on the last one: What support for user interaction the kernel of an interactive system should supply. In order to answer this question, we analyze the external behavior that a user expects from an interactive system, and from this derive the characteristics of such a kernel.

3. Design principles: an interactive system as seen by the user

A study of existing user interfaces to interactive systems reveals a great variety of approaches. At one end of the spectrum we find the drive to perfect the I/O aspect of interaction technology, ranging from the physical design of the interface according to human factors points of view (see e.g. [GV 80]), to the development of powerful software techniques such as the window concept that gives the user several independent work areas on the screen. As an example, the "programmer's assistant" [Tei 79] is an application based on this concept. At the other end of the spectrum the literature on the design of user interfaces reflects an increasing influence of psychology and cognitive science, as can be seen in recent surveys on the psychology of human-computer interaction such as [Mo 81].

Our approach lies between the two positions outlined above. We don't emphasize sophisticated hardware interfaces that may be needed in some demanding applications such as CAD, but not in others, and we don't subject our analysis to the rigor of controlled experiments. We have observed hundreds of users of interactive systems, in particular casual users [Nie 80], and we have simply drawn common sense conclusions from a vast pool of experience. We aim at establishing concepts and principles whose systematic application avoids the majority of design errors observed in today's man-machine dialogs [Nie 82], [HBR 81]. In order to serve as helpful guidelines for the design process such rules must be carefully formulated. As an example to avoid, consider the two admonishments *give as much information to the user as possible* and *do not clutter the screen with superfluous information*.

Each sounds reasonable in isolation, but they clearly

contradict each other unless we specify *WHAT* kind of information is meant in each case.

In spite of the current emphasis on man-machine communication, the insights gained so far tend to be applied only locally, within an individual application program, rather than globally for an entire system on which many applications run. Today's interactive system presents itself to the user as a collection of unrelated application programs. The user is confronted with several distinct interfaces, each one based on its own set of concepts and principles of interaction. As the community of users of command languages grows rapidly and includes an increasing fraction of novices or occasional users, we must develop systems in which all application programs talk the same language. We attempt to realize this goal in two steps:

- 1) By introducing a conceptual framework for the characterization of interaction sufficiently general to apply to any task performed interactively, independent of any application. We define a *user's model of the system* that is valid in all interactive application programs that run on the system.
- 2) By designing a kernel of an interactive system that supports this model and avoids the deficiencies of traditional operating system discussed in section 2. In particular, it contains a front-end dialog processor that handles all user interaction, thus ensuring a uniform dialog style.

In the remainder of this section we briefly recapitulate the concepts referred to in point 1) above, first presented in [NW 80]. The rest of the paper is devoted to 2), the design of the kernel.

Analysis of man machine dialogs shows that most systems handle different levels of data and of interaction in different ways. This usually leads to dialog design errors, as the following examples show:

Ex 1) Inconsistent treatment of data environment: Many editors let the user inspect the text of one file, but fail to tell him the name of the file which he is editing, or the names of other text files from which he may want to use parts.

Ex 2) Inconsistent treatment of command environment: Every command interpreter provides a *quit* at the command level, but many fail to provide a quit or abort within a command, e. g. at the parameter level, so that an action erroneously started must be completed.

In the first case the user lacks information about a part of the current system state. He cannot obtain it without leaving the editor, thus changing his dialog state. In the second case the system lacks a universal *quit* (other than an emergency abort like *Control-C* or *reset*).

An analysis of the difficulties experienced by many users has led to three questions to which a user may want to obtain an answer at any instant of the dialog:

- Where am I ? (Which data are accessible and may be affected by my commands ?)
- What can I do ? (What commands are active ?)
- How did I get here and where can I go ?

As later sections show, the interactive system XS-1 incorporates concepts that provide a well-defined answer to each question above:

- a *SITE* as a representation of the collection of

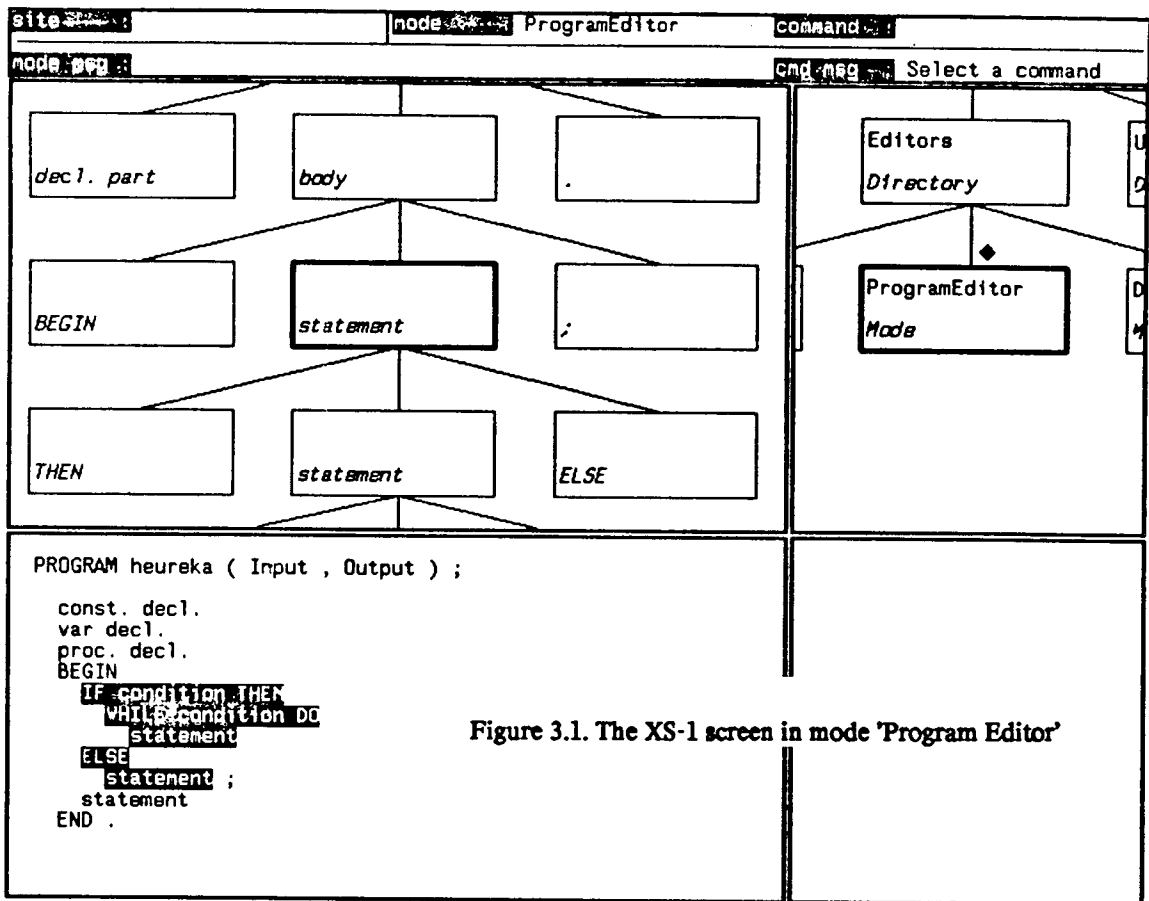


Figure 3.1. The XS-1 screen in mode 'Program Editor'

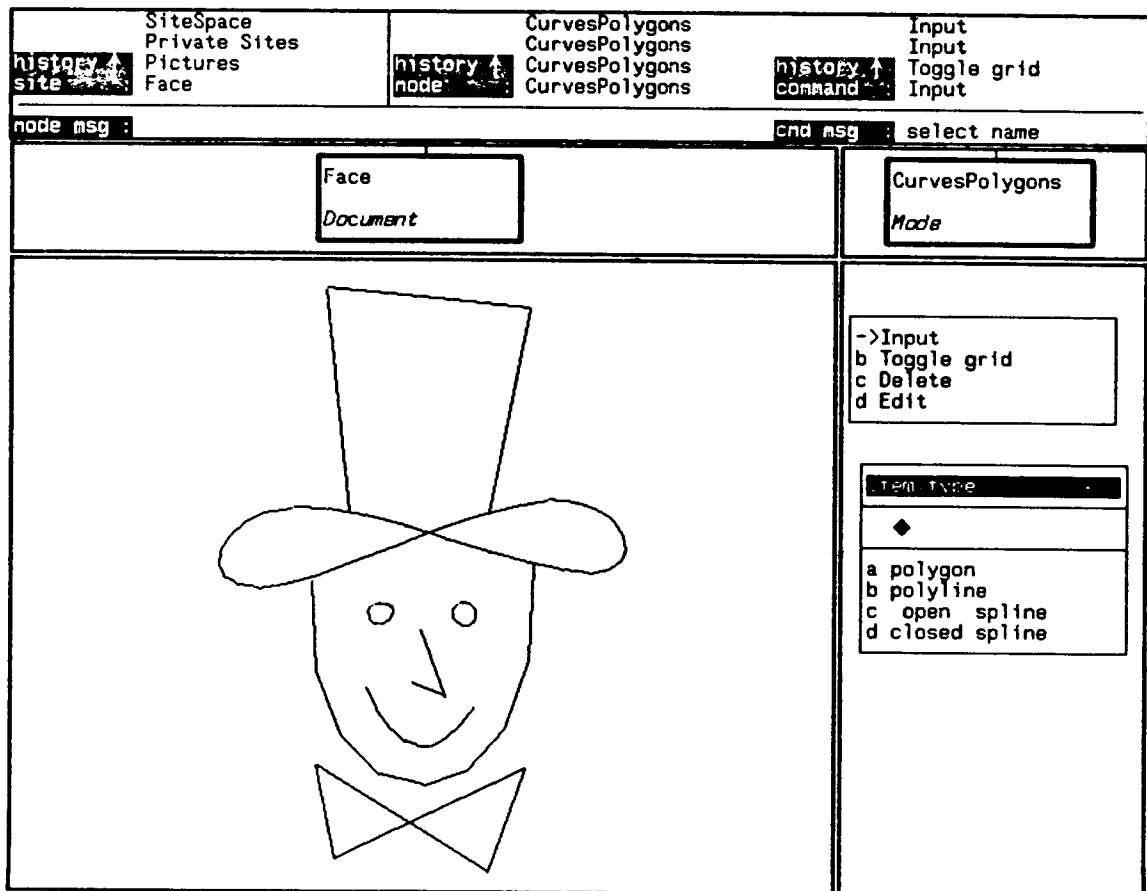


Figure 3.2. Editing curves with the mode 'CurvesPolygons'

data currently accessible

- a **MODE** as a representation of the collection of actions currently active
- a **TRAIL** as a sequence of pairs <site, mode> which represents a dialog session.

The user's data and command environment are presented to him in terms of these three concepts **IN ANY PART OF THE SYSTEM and AT ALL TIMES**, not just sporadically. The realization of this strict requirement leads naturally to the concept of **UNIVERSAL COMMANDS** for state inquiries and general dialog operations. They are universal in several meanings of this term, in particular: they are always active, are always invoked by the same physical actions, and always have the same interpretation regardless of which mode is active. As evidence for the usefulness of these concepts, notice that the examples of dialog errors above would have been avoided by their consistent application. The problems arose from the fact that different levels of data structures (text within a file on one hand, entire text files on the other) were conceptually disconnected and that different levels of command selection (main dialog and subdialog) were separated and handled by different programs in the system.

To illustrate the concepts of sites, modes and trails, figure 3.1 shows a snapshot of the user's interaction with a program editor. The state of the system is displayed in the two large center windows:

- the current site, shown as a heavy box in the left window, is of type *statement*;
- the current mode, shown as a heavy box in the right window, is named *Program Editor*.

The left bottom window displays the data attached to the current site (inverted) embedded in surrounding data. The tree structure imposed on this data is shown twice: in the site window and (by means of indentation) in the data window.

Figure 3.2 is a snapshot of the screen in a graphic mode *CurvesPolygons* where we can draw curves by entering control points. The user can dynamically adjust the size and shape of the five windows that make up the logical screen. In figure 3.2 he has enlarged the data window, compressed the site and mode windows, and scrolled in a part of the trail window (upper screen) that displays the recent dialog history. On our trail to the current site *Face* we passed through *Site Space*, *Private Sites* and *Pictures*, while the current mode *CurvesPolygons* did not change. The command window at the lower right shows the currently active command *Input* and its currently active parameter, the *item type*.

4. The kernel of an interactive system and its components

The concepts described in the preceding sections have been realized in a prototype system XS-1, which we now present as a case study of an integrated interactive system for small computers. Our present interest lies in the system's kernel, rather than in the application programs that run on it. We consider it as an experiment in designing an operating system that avoids the deficiencies described in section 2 and supports the concepts of section 3.

The functions and structure of the kernel

In conventional interactive systems the application program is responsible for defining, controlling and displaying its data and its dialog. The operating system does not communicate with the user except for starting or aborting the application program. By contrast, the kernel of XS-1 provides a highly structured space in which all the data and all commands are embedded, and provides operations to explore this space and manipulate its elements. The function of the kernel is to allow application programs to structure their data in the common space and to leave the dialog treatment to the system. As a consequence, the user interface is mostly determined by the kernel, rather than by the application program, and the user gets the impression that all programs in the entire system *talk the same language*.

In order to achieve this goal, the kernel of XS-1 has four major components:

EXPLORE makes sites, modes and trails visible to the user. It defines and controls all motion on these spaces and handles user requests for screen layout changes by means of universal commands. This meta-dialog allows the user to inspect the current system state without changing it.

TREE EDITOR provides universal commands for structural manipulations that are common to most editors regardless of the type of data objects they manage. It is syntax-directed in order to reflect the relationships between different data types.

CENTRAL DIALOG CONTROL is a front-end dialog processor that handles user input at all levels, from key press to command, checks it for syntactic correctness and records it. It directs information to Explore, the tree editor, and to modes.

TREE FILE SYSTEM realizes the data access and manipulation requirements imposed by Explore and the tree editor. It handles data of any size, from a single character to a large file, in a uniform way.

The kernel also contains a screen management package that handles an arbitrary number of windows. It can be considered to be a virtual terminal.

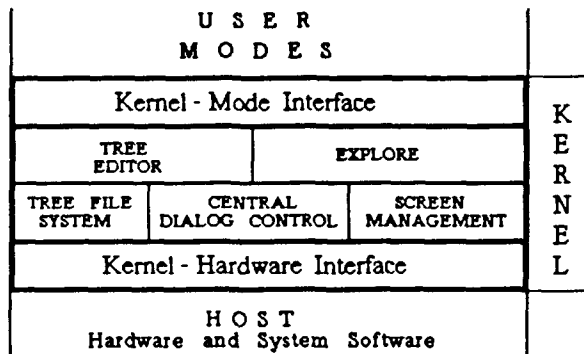


Figure 4.1 Components of the XS-1 kernel

Interface between the kernel and modes

Conventional interactive application programs devote much code and time to screen layout and to dialog handling. In XS-1 these activities are centralized in the kernel. A mode presents its data on a virtual

screen and lets the screen handling package map the virtual screen onto the physical screen in a window of arbitrary size and position. The kernel handles the entire dialog and provides universal commands for visualizing the state of the system and for performing structural changes to the site and mode spaces. A sizable portion of the code of typical application programs resides in the kernel; only the mode-specific operations remain to be written by the applications programmer.

The interface between the kernel and modes is syntax-directed with respect to data and to commands: A mode defines the syntax of the site space it works on, and the syntax of its commands. The former drives the tree editor, the latter drives central dialog control. Any collection of data, independent of their size or type, may be attached to a node in the site tree, and any collection of commands may be attached to a node in the mode tree. Data attached to the current site is affected by commands; commands attached to the current mode are active.

Interface between the kernel and host

This interface maps the requirements of the tree file system, of central dialog control, and the screen management package onto corresponding utilities of a conventional operating system, such as schedulers and drivers.

5. Implications for the applications programmer (mode writer)

The design strategy explained so far, of a kernel that manages the user-interface for all application programs, in particular the concept "universal commands + modes", has profound consequences for the behavior and the structure of application programs. In summary: many useful facilities are provided, many constraints must be obeyed. Compared to the conventional applications programmer, the mode writer's workload is reduced, but he has to be aware of a more complex environment which guarantees uniform system behavior for the end-user.

Facilities: A great part of the user interface, above all any application-independent dialog (exit from an application, display of command history, etc.), is provided by the kernel in the form of universal commands, and remains active during the application. Even the application-specific part of the interface need not be programmed explicitly; the syntax of the mode commands is described by means of tables, and the kernel places prompts on the screen and handles all user entries (such as feedback to invalid user entries).

Constraints: All data is embedded in the space of sites, all commands are embedded in the space of modes. Data becomes accessible, and commands are activated, by means of universal commands, but the mode writer has no control over the user's activation of the latter. In order to write a "well-behaved mode", he has to understand the XS-1 philosophy and structure his data and commands accordingly.

Let us illustrate both points by means of an example: adding a graphics mode for editing curves and polygons. The mode writer must define and

implement new object types, say *pictures*; define commands for handling these objects, and implement corresponding operations. This work involves two parts: definitions within the framework of the kernel interface, and programs in the underlying programming language, Modula-2.

Embedding new data types and modes into the XS-1 spaces

We define new object types by defining their syntax in tree form. Figure 5.1 shows that a picture consists of an arbitrary number of picture elements; a picture element can be a closed curve, an open curve, a polygon, a polyline, or a polymarker. The syntax is entered with the tree editor as a subtree of the node *Grammars* in the site space.

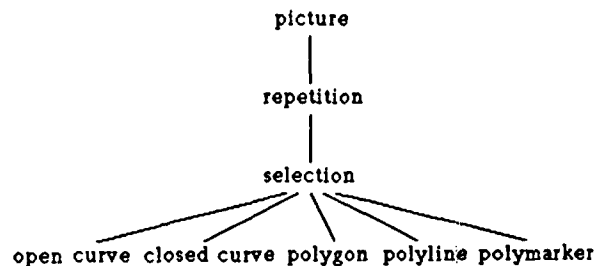


Figure 5.1 Syntax of data type 'picture'

Analogously, in the mode tree we find a node *Editors*, with sons such as *Program editor* and *Graphic editors*.

The latter is of type *directory*: it has no commands, it merely serves as an aid to the user for locating related modes. Since the XS-1 spaces are always visible and easily explored, a large mode tends to be structured into submodes with few commands each. We add our new mode *Curves & Polygons*, of type *mode*, next to the existing mode *Graph plotting*.

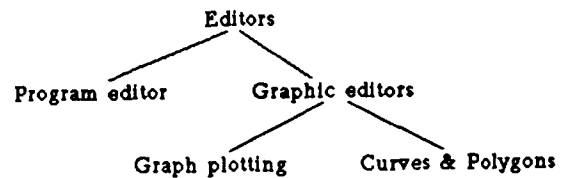


Figure 5.2 'Curves & Polygons' embedded in the mode space

Implementation of data, commands and operations

The mode writer chooses the level of detail at which his data syntax is defined in tree form as shown above. In our example, we choose *polygon* as a terminal symbol of the grammar, and represent a polygon internally as a sequence of points stored in an array. This concrete implementation of the data types is outside the kernel interface.

Analogously, we define the syntax of the commands according to the kernel interface rules, in the form of command tables: the keywords, the prompts, the number and types of parameters, their interdependencies, and a link to the programs that

perform the operations to be invoked by these commands. These programs are outside the kernel interface.

Figure 3.2 shows how the kernel displays these commands on the screen. Only the mode commands that have to do with the editing of pictures, are displayed all the time. The universal commands appear at the press of a mouse button.

An assessment of the kernel's contribution to mode writing

The data-specific operations of a small mode such as *Curves & Polygons*, which calls upon a sophisticated graphics and screen management package, are easily programmed. The kernel provides the rest, namely the complete man-machine interface, so the entire applications program is implemented with little effort. XS-1 shows itself at its best for the realization of a collection of small modes. For a large monolithic applications program the relative benefits for the implementor would be smaller; the only savings are due to the fact that he doesn't need to design his own man-machine interface.

The usefulness of the kernel components to the mode writer is also demonstrated by the skeleton of the *ProgramEditor* that at the moment has no mode-specific commands at all: all editing is handled using the syntax-driven tree editor. This is possible because the natural form of source programs is tree-structured.

The kernel not only saves the applications programmer coding time, but also encourages modular programming. The separation of command structure and actions leads to many programs with identical interface to the kernel, that are relatively independent of each other, and hence can be combined in different subsets.

This completes our bird's eye view of the system; the remaining sections describe the four major components of the kernel in greater detail.

6. Explore: Visualization of the system state, motion and dialog history

Explore is the kernel component that makes the entire state of the system visible to the user at all times. A state in the user's model of the system (section 3) is a pair <SITE,MODE> that defines the DATA that may be affected by the user's operations and the COMMANDS that are active at this moment. A TRAIL is a sequence of states that describes a dialog session. Explore presents the state and trail as shown in figures 3.1, 3.2, and 6.1.

In order to explore the state spaces, Explore provides view and motion commands. Since the site and mode spaces have the same structure, a tree, the same set of commands applies to both. View commands include *grow* or *shrink* any screen area at the expense of the others, *change tree display*, *scroll*. Relative motion commands are defined with respect to the current node (*up*, *down*, *left*, *right*); absolute motion with respect to a node identified either by pointing on the screen, or by naming (*goto*, *mark* and *return*).

The command and the trail areas have a linear structure which is considered to be a degenerate tree. Hence the tree motion commands defined on the site

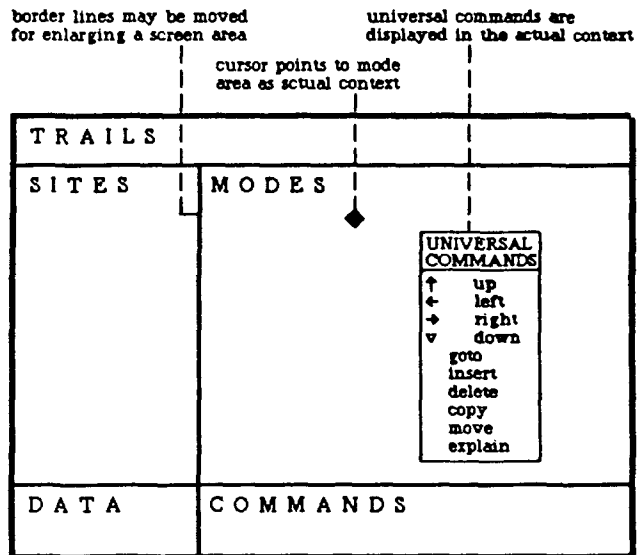


Figure 6.1 Screen layout showing menu of universal commands invoked in the mode space

and mode spaces can also be interpreted in these areas. The command *up*, for instance, has the following interpretations:

- *move up* in the site space
- *leave the actual mode* in the mode space
- *end the parameter sequence of the actual command* in the command space
- *stop trail execution* in the trail space.

The possibility of this common interpretation, together with the fact that they are always active and invoked by the same physical action, makes these commands universal. They are invoked in each space by moving the cursor into the corresponding screen area, where a menu appears upon request.

7. The tree editor: A syntax-directed editor shared by all applications

The space of sites and the space of modes represent the structural dependencies among data and commands, respectively. Since we impose the same tree structure on both, the tree editor is used for structural changes in both state spaces. The universal presence of its commands makes it superfluous for a mode to offer any structural operations; only the data-specific operations have to be provided by each mode.

The tree editor offers a small set of basic operations: Change node name and type, insert a new node (with empty data), delete a subtree, copy or move a subtree, shrink or expand the tree vertically, and split or merge nodes and the corresponding data. In all but the last two operations data attached to a node remains unaffected, whereas split and merge alter the allocation of data to nodes. It is up to each mode to define the effect of a split or merge of data involved. As an example, a text editor may interpret a merge of consecutive paragraphs as concatenation.

The tree editor is directed by syntax tables activated

upon entry into a mode. Each node has a type field that serves as a classification scheme for the data attached to it, such as *text*, *program*, *picture*, *document*. Context-free grammars whose symbols are node types drive the tree editor; the legality of all its operations is checked before execution. The top of the site space has the type *directory*, which is the start symbol of the default grammar that allows arbitrary modifications. As an example of how kernel components are used by modes, a the skeleton of a syntax-driven program editor is reduced to defining a syntax table of the programming language.

The tree editor is an interactive tool: Parameters for tree operations are entered by pointing at the corresponding tree locations, as for example the source and target of a copy operation (the root of a subtree and a gap). Input of commands with their parameters and feedback is managed by the central dialog control facility to be described next.

8. Central dialog control: Getting all modes to talk the same language

Central dialog control handles I/O for Explore, for the tree editor, and for all modes, at various levels of interaction, from key stroke to command, in a uniform way. From the user's point of view, a key stroke is as much a unit of interaction as a command: he may wish to cancel or repeat either, and expects a feedback such as accepted or rejected in either case. In conventional systems control and feedback tend to be different for the key stroke and the command: whereas a key stroke is cancelled by a *backspace*, commands usually can't be cancelled (undone).

The lack of common input control facilities has unpleasant consequences as in the following examples. A user may have to reenter a whole command because of a single wrong parameter, or reenter an entire name because of a single mistyped character. Reentering a value for a parameter and reentering a command are conceptually equivalent; both actions can be considered to be editing operations on the dialog. The corresponding mechanisms should be identical and always available.

Central dialog control provides a uniform user interface at all levels of interaction. A hierarchy of dialog components is defined as the objects to be manipulated by the user during interaction. At the lowest level dialog atoms are defined (e.g. a key stroke or a cursor movement). Symbols are built out of atoms according to several rules (e.g. a set or a sequence of atoms). Parameters and commands are constructed in a similar way. The sequence of commands entered by the user is another representation of his trail (the sequence of states traversed during the dialog). Central dialog control records it, displays it in the trail screen area, and makes it available to the user as an object to be edited and replayed.

Central dialog control dispatches information extracted from the user input to the other kernel components: Universal commands to Explore or the tree editor, all other commands to the active mode. The universal commands are predefined. At any moment (even in the midst of a parameter specification) the user may activate them. A collection

of commands that are related to each other are grouped into a mode. The structure of a mode, i.e. its set of commands with their interdependencies, is described by means of command tables activated upon entry into the mode. Interdependencies are specified by interaction rules that can activate different subsets of commands or parameters.

9. Tree file system: Efficient management of small and large sets of data

We have analyzed the requirements imposed on file systems by highly interactive applications such as text editing and interactive information retrieval in section 2, and stressed the importance of response time for user efficiency. From the burgeoning literature on the psychology of man-computer interaction we can derive real-time constraints that a file system must obey. In sections 6 (Explore) and 7 (Tree Editor) we have presented a user interface that allows basic operations on data by means of universal commands. The task of the file system consists in an efficient implementation of access and manipulation primitives for data of any type and size.

The first step is a choice of data structure, both at the logical and at the physical level. The logical organization must allow fast physical access to any set of data that can be specified by Explore. A tree structure was chosen as a feasible compromise between the quest for generality in representing relationships among data and the necessity of efficient realisation. The choice of data structure at the physical level was between address computation techniques and lists. The former promise fast access to static data sets, but are difficult to maintain when frequent insertions and deletions are necessary. We were willing to accept slower random access and chose lists that are easier to modify. Careful allocation of nodes to pages yields an efficient way to implement a large tree on a disk. We first describe the tree file organization and its access functions. We then provide some statistics about the file system's behavior in a document retrieval application.

Tree file

The logical description of a tree node is sketched below:

- position in the tree (links to neighbor nodes)
- node attributes:
 - name (character sequence)
 - type
 - data (sequence of bytes)

A name (not necessarily unique) can be given to a node. The tree file system provides an information retrieval mechanism on node names. The node-type field contains an indication of type of data the node represents (Section 6 describes its use for syntax-oriented purposes). The node data field contains a necessary description for accessing node data. The sets of data attached to the nodes are organized into groups and stored on a file.

Tree file functions

The tree file system maintains a window into the tree - the "current node". This window can be moved to any node in the tree and is used to inspect and modify

all information related to a node.

- Change of attributes:

Setting and retrieving the name and type of the current node. Reading, writing, inserting, and deleting an arbitrary number of bytes at a specified node data position. Some node attributes (name, data) may be absent.

- Motion:

a) Relative motion to the neighbors of the current node, e.g. motion to the brothers (left, right, leftmost, rightmost brother), to the father, or to one of its sons.

b) Absolute motion to a node recognized by its name or system-generated identification. In the case of an ambiguous node name, all nodes with the same name are retrieved. The path through the tree can be controlled for reasons of protection.

c) Traversal of the current subtree. An arbitrary action can be executed at each node.

- Tree manipulations:

All structural manipulations are performed in the son generation of the current node: Insertion of a new node and deletion of a subtree. Expanding a tree by one level (i.e. inserting a new node between the current node and an arbitrary sequence of its sons) and shrinking the tree. Splitting a son into twin nodes (the son's node data and its subtrees will be split as well) and merging a sequence of sons and their data. Moving and copying a subtree.

- Recovery:

The tree file can be reset to a backup state; during this state, all modification to the tree are backed up on shadow pages. A backup state can be terminated either by discarding it, thus making the current state valid (commit), or by returning to the previous state (undo). Several backup states can be stored.

We measured the file system's response time behavior in terms of the number of disk accesses, since the cost of accessing data in the main memory is negligible in comparison. The node allocation scheme is such that no disk access is required for motion to the brothers, whereas motion to the father or to one of the sons requires at most one. The absolute motion can be characterized by the distance of the current and target node to the top of the tree. Reading and writing of node data requires disk accesses proportional to its size. Insertion and deletion of a node requires regularly one disk access. To give an illustrative figure, the traversal of a subtree with about 8000 nodes that was generated for the information retrieval system [FB 82] takes about 10 seconds on the Lillith computer [Wi 81].

10. State of project, experience, plans

XS-1 is written in Modula-2 [Wi 80] and has been running in several preliminary versions. A prototype ran on a DEC LSI-11 [Bu 81], but the different components of the kernel could not be integrated into one system on a machine that provides only 28K words of memory. The tree file system [Su 82] has been running successfully for two years. It is the core of another project besides XS-1, an interactive

information retrieval system [FB 82]. The complete kernel of XS-1 as described in this paper now runs on a Lillith machine [Wi 81] with 128 K 16-bit words of memory. Half a dozen modes have been written for testing the design of the interface between the kernel and modes, and for demonstration purposes: the skeleton of a syntax-directed program editor, small graphics editors, an Othello game program, and utilities for file conversion and node-data patching. On earlier versions of XS-1 we also had an editor for command syntax tables, a rudimentary mail server and envelope editor, and a fast text search facility; we hope to adapt these to the current kernel interface. The major applications project under way is a collection of modes to facilitate the development of small interactive programs.

Interactive application programs written for XS-1 present an entirely different structure to the user than conventional ones: they consist of several small modes, rather than of one monolithic package. We consider this to be an advantage particularly for the casual user. At a certain moment he must be aware of only a few relevant actions; the universal motion commands on the space of modes gives him convenient access to a large set of operations.

In an analogous way, data that conventionally would belong to a file is structured into objects of small size. A document in the tree of sites typically has leaf nodes which contain one paragraph each. We consider the effective management of highly structured data and command environments, and a simple, consistent user's model of the system, to be the major achievements of the XS-1 project.

ACKNOWLEDGMENT

We are indebted to many people who contributed to the development of XS-1 in the form of projects, in particular to W. Abramowicz, F. Caratti, H. P. Giger, P. Hofmann, K. Inoue, A. Kierulf, R. Schaub, R. Straus, F. Vonaesch. The hardware and systems software provided by N. Wirth and his group have been essential. Sharing software with the information retrieval group of H.P. Frei has been a great help.

REFERENCES

[BSTJ 78]

UNIX Time-Sharing System, The Bell System Technical Journal, Vol 57, No 6, Part 2, July-August 1978.

[Bu 81] Burkhart, H.,

Begriffe zur Systematisierung der Mensch-Maschine Schnittstelle und ihre Anwendung auf den Entwurf von Editoren, Dissertation 6838 ETH and Report 43, Informatik, ETH, 1981.

[De 82] Denning, P. J.,

Are Operating Systems obsolete?, Comm. ACM, Vol 25, No 4, 225-227, April 1982.

[FB 82] Frei, H.P. and Baertschi, M.,

A Data Organization for Information Retrieval on a Personal Computer, Personal Computing (to appear), Teubner, Stuttgart, 1982.

[GV 80] E. Grandjean, and E. Vigliani (editors),

Ergonomic aspects of visual display terminals, Proc. Intern. Workshop Milano, March 1980, Taylor and Francis Ltd., London, 1980.

- [HBR 81] P. Hayes, E. Ball, and R. Reddy,
Breaking the man-machine communication barrier,
IEEE Computer, Vol 14, No 3, 19-30, March 1981.
- [KHPS 61] Kilburn, T., Howarth, D. J., Payne, R. B.
and Sumner, F. H.,
The Manchester University Atlas operating system.
Part I: Internal organization, Part II: User's
description, Computer Journal, Vol 4, No 3,
222-229, Oct 1961.
- [LWSS 80] H. F. Ledgard, J. A. Whiteside, W.
Seymour, and A. Singer,
An experiment on human engineering of interactive
software, IEEE Trans. Software Engr., Vol 6, No 6,
602-604, Nov 1980.
- [Me 62] Mealey, G. H.,
Operating systems, Rand Corporation Report
P-2584, May 1962.
- [Mi 56] G. A. Miller,
The magical number seven, plus or minus two: Some
limits on our capacity for processing information,
Psych. Review, Vol 63, No 2, 81-96, March 1956.
- [Mo 81] T. P. Moran (guest editor),
The psychology of human-computer interaction,
Special Issue, ACM Computing Surveys, Vol 13,
No 1, March 1981.
- [Ni 80] J. Nievergelt,
A pragmatic introduction to courseware design,
IEEE Computer, Vol 13, No 9, 7-21, Sep 1980.
- [Ni 82] J. Nievergelt,
Errors in dialog design and how to avoid them, Proc.
Intern. Zurich Seminar on Digital
Communications, IEEE, March 1982.
- [NW 80] J. Nievergelt and J. Weydert,
Sites, modes, and trails: Telling the user of an
interactive system where he is, what he can do, and
how to get places, in R. A. Guedj (ed.),
Methodology of Interaction, 327-338, North
Holland Publ. Co., Amsterdam 1980.
- [No 81] D. A. Norman,
The trouble with UNIX, 139-150, Datamation, Nov
1981.
- [Su 82] H. Sugaya,
Tree file: A data organization for interactive
programs, Dissertation 6944 ETH, Zurich, 1982.
- [Te 79] W. Teitelman,
A display oriented programmer's assistant, Int. J.
Man-Machine Studies, Vol 11, 157-187, 1979.
- [We 82] J. Weydert,
Eine Dialogmaschine, Dissertation ETH Zurich,
1982.
- [WWG 51] Wilkes, M. V., Wheeler, D. J. and Gill, S.,
The preparation of programs for an electronic digital
computer, 1951 (reprinted Addison-Wesley 1957).
- [Wi 80] Wirth, N.,
Modula-2, Report 36, Informatik, ETH, March
1980.
- [Wi 81] Wirth, N.,
Lilith: A personal computer for the software
engineer, Proc. 5-th Int. Conf. on Software Eng.,
2-15, IEEE Computer Society Press, March 1981.