A SYSTEM OF COMMUNICATING RESIDENTIAL ENVIRONMENTS

by

Erik Sandewall, Henrik Sörensen, Claes Strömberg

Informatics Laboratory
Linkoeping University
Linkoeping, Sweden

Abstract: The SCREEN system is an experimental tool for development and maintenance of application software. It is organized as a System of Communicating REsidential ENvironments, where each environment may be e.g. a programming environment or an end-user environment. Environments are able to send and receive modules which contain programs and data, and the maintenance of an application (throughout the software life-cycle) is performed by communicating such modules. Each programming environment may send several modules to each of several end-user environments, to account for specialized user needs, as well as updates of those modules. End-user environments consist of a fixed framework, into which contributed modules can plug in. The framework is designed specifically for each class of applications, and can be viewed as a special-purpose operating system for that class of applications. The organizational principles used in the SCREEN system are an extension of the residential programming systems that have been developed for Lisp, but they have an increased emphasis on software engineering issues. They represent an alternative to the classical principles for the production and use of software, which are based on the use of compilers, linkage editors, and general-purpose operating systems.

1. Scope.

The *SCREEN* system is an experimental tool for development and maintenance of application software. It is organized as a System of Communicating REsidential ENvironments, where each environment may be either of:

- a conventional programming environment, i.e. a tool for the development of programs used by a programmer
- an end-user environment, i.e. a system used by an end user for a purpose dictated by his or her job (roughly equivalent to the 'end user facility' of the Codasyl EUF group)

but potentially also for example:

- an end-user programming environment, i.e. an environment which enables an end user to specify the system's behavior patterns in application-oriented terms
- a control environment, i.e. an entity which autonomously controls some aspects of the total system's behavior, e.g. causing certain programs to be run at certain hours.

Environments are able to send and receive modules which contain programs and/or data; to extract such modules from the web of information stored in the environment; and to integrate incoming modules into their structure. In particular, this means that when the system software has been augmented or updates have been performed and checked out in a programming environment, fresh copies of appropriate modules may be sent in suitable form (e.g. compiled code) to other environments where they are implanted or substituted into the internal structure.

Environments are roughly equivalent to the 'systems', 'workspaces', or 'sysouts' of residential Lisp or APL programming systems, but compared to e.g. Interlisp they contain additional structure and additional tools which support the sending and receiving of modules.

This architecture has been developed in the context of office information systems, although we believe that it is appropriate for other application areas as well. The examples in this report will be taken from office applications.

The *SCREEN* system has been used as a substrate for experimental applications in hospital data processing (ref. 1, 2) and in office services for a university department (ref. 3, 4, 5). The software presently supported by the system has the size of several hundred line printer pages of prettyprinted (Lisp) code.

The present report has the following contents:

2. a brief description of the services for the end user and for the programmer, that have dictated the architecture of the system

3. a discussion of how the software production technology supported by the *SCREEN* system relates to compiler based production technologies, and to conventional programming environments

4. a description of the application-oriented structures in the *SCREEN* system, i.e. structures which reflect the various services that are offered to the user.

5. a description of local system-oriented structures, i.e. the structures of the contents in the environment's data base: procedures, data, and their description (which is also stored in the environment, recursively)

6-7. descriptions of global system-oriented structures for use during the operation

2. End user environments and programmer environments

An end user environment (EUE) is an integrated set of software for one user. It may for example be the software for one personal computer. We will speak about services 'in' the EUE with the understanding that the containment is perceived by the end user, whereas the actual implementation allows several EUE to share certain parts, invisibly to the user.

The EUE contains some general purpose services, for example a text editor and a mail system. But it also contains support for activities which are specific to one user or a limited set of users. In a research group, for example, persons who are in charge of receiving visitors, or of editing a newsletter, or of administrating computer and terminal equipment for members of the group, may have special services in their EUE's for administrating these jobs. The visitor handler may e.g. have a special tool whereby he can set up a proposed schedule for a visitor, who is to see different people at different hours, have mail messages sent automatically to the proposed visitees, have the answers collected or reminders sent out, still automatically, and finally obtain a reasonable printout of the day's schedule for

some of the user-specific programs may be programmed by the user himself. Some may be programmed for him by another member of the group. Finally, for cases of information flow in the organization (for example the routine path of a purchase order or an application for leave of absence), one person (programmer) will design the computer routine for that information flow, and have appropriate, coordinated modules sent out to the participating EUE's. In fact, one of the major developments in our applications has been the development of a special-purpose language whereby such information flows may be specified concisely, debugged in the programming environment, and then transformed to higher-performance modules which are sent out to EUE's (ref. 1, 2, 4).

In order to help the end user find his way around, the EUE contains a non-trivial dialogue mechanism at the top. (The command language provided by a conventional time-sharing system on the other hand, is an example of a simpler dialogue mechanism). The current *SCREEN* system uses a hierarchical menue system (for command menues, data-set menues, etc.) as the major dialogue mechanism.

Since amendments to an EUE (additions or updates) may come in continously, the EUE is organized as a framework within which modules are located in a well-defined way. External amendments attach more contents to the framework, and the dialogue mechanism is directed by those current contents. The simplest type of update is to exchange or add an item in the 'menue of available operations', but there are also other kinds. For example, an Agenda facility in the end user system allows the user to ask the EUE to later remind him of phone calls, deadlines for submission of papers, or other duties. An external contributor may now add more classes of tasks to the agenda, so that an agenda printout may be

4 PHONE CALLS TO MAKE OF WHICH 1 ASAP
11 NEW MESSAGES IN COMPUTER MAIL
LISP CONFERENCE PAPERS DUE NEXT WEEK
6 LEAVE OF ABSENCE REQUESTS PENDING
14% OF WORKING TIME THIS WEEK SCHEDULED FOR MEETINGS

where the last three items are the results of three contributed modules. Each line has been obtained from a small program which inspected the current data base to compute the figures. Of course the user can proceed directly from this agenda

menue to doing the tasks, or for finding out more about them, which means that he invokes other programs or data associated with each module.

Thus a module that is sent to the EUE in order to provide one service, often consists of several parts. For another example, an additional command for a command loop consists of:

- the name of the command
- brief description of the effect of the command, for use in menues
- a longer description, to be printed out whenever asked for
- a description of the parameter collection (for example the syntax for parameters, and/or a prescription for the dialogue that asks for paramters from the user or from the context where the command is called)
- a procedure that is to be called in order to execute the command

This information is plugged into the EUE in such a fashion that it is available in all the various relevant ways. Since the transferred unit consists of several parts which plug into different positions in the receiving environment, the term *multidule* is proposed for the composite.

The other major type of environment corresponds to what is usually called programming environments. However we shall instead use the term development environment, (DE) in order to emphasize a broader intended scope than merely programming. The requirements on the DE overlap with those on the EUE. In particular, a good dialogue facility in the DE helps the designer keep a large reportoire of design tools, and a plethora of details in the supported systems, under his fingertips. Other important facilities are:

- special purpose languages, for describing specific classes of applications or specific technical functions, and storing the descriptions in the data base of the DE. Our current system supports special purpose languages for information flows and for command loops, as described above, but also for interactive handling of forms on the screen, and interfaces to facilities developed by Tore Risch at Uppsala University, for compiling data base queries and report specifications into programs, as well as special-purpose languages for internal use by SCREEN.

- self-description: the purpose of the DE is not only to test-execute programs, compile them, and send them to the EUE:s, but also to maintain miscellaneous descriptive information about the programs, for the following purposes:
  - on line use for documentation in the DE and the EUE
  - off line use: manuals are generated from the information in the DE
  - manipulation support. One common reason for documentation of software systems is to provide the programmer (i.e. maintainer of the software) with enough information that he can translate what he really wants to do to the program, into a number of local operations of various kinds (text edits, recompilations, data base updates, etc). Instead of using a documentary data base to generate the tables that support the programmer in this way, the data base may be used to support the operation, so that the programmer can specify in his own problem oriented terms what he wants to do, and have the details of the manipulation be performed automatically.

- modularity: self-description is achieved by having the contents of the DE organized as modules, each of which is a chunk of information (programs and data) which can be referred to and manipulated as an entity whenever appropriate. - The contents of an Interlisp 'file' as described by the fileCOMS, is a well-known example of such modules, but we are using the module

structure for more purposes than in Interlisp.

- design modules are different from production modules. Usually, the software for one sub-system is maintained in one DE, and modules are then shipped to EUE:s, and possibly to other DE:s. However, the modules that are used internally in the source DE are not always the same as are sent to the usage environments: the DE may re-organize the modules and their contents, e.g. do partial evaluation on procedures in order to shorten access paths. This is a way to solve a common problem with strict modularization, i.e. that it tends to introduce excessive traffic across module boundaries.

Finally, a characteristic property in both EUE:s and DE:s is their regenerative ability. This is most easily explained in Interlisp terms. A 'sysout' in Interlisp is an entity which is initially obtained by 'loading' a number of 'files' into a 'fresh Lisp', and then working with it for a number of interactive sessions, and preserving it between sessions using the procedure 'sysout'. Programs and data are edited within the 'sysout', but backup is kept by dumping 'files' from time to time. At some time, the 'sysout' must be discarded (for various technical reasons), and a new 'sysout' is started up. Often the start-up involves not only loading files, but also other operations e.g. setting memory allocation parameters.

The environments in *SCREEN* are roughly equivalent to 'sysouts' (details below), but they usually contain enough information that they are able to generate an equivalent descendant. That is, the environment knows what 'files' are to be loaded into a 'fresh Lisp' (or some other predefined base), in order to create a fresh copy of the environment itself, and contains routines for performing such a bootstrap. This is a minor convenience when a single programmer is in charge of an environment, but it is significant for EUE:s where contributions may come in from various sources, and the owner of the EUE does not know what modules are in it.

It is also of significant value when several people are working on the same DE, since one then does not need to go through manual administrative routines to document what modules are in the environment. The DE is perceived like a piece of equipment: you can go to it, look around to determine its current status, and set out to improve it. In fact, it is better than most physical equipment, since it is able to tell you what parts you may or may not touch.

3. A discussion of software production technologies

The *SCREEN* system represents a new approach to the production of software. The purpose of this section is to discuss the conventional production technology, and compare it to the new approach represented by *SCREEN*.

The classical production line for software is organized around the compiler. 'Before' the compiler stage, there are text editors, program generators, and various documentation tools. 'After' the compiler, there is a linkage editor (for connecting separately compiled modules into programs) and a host environment within which programs are executed. The latter may be an operating system which performs a dialogue with the user, and enables him to select programs to be executed, or it may be a fixed set of JCL routines which define which programs are to be run at which times.

This pattern of work is illustrated in figure 1. We shall refer to it as the compiler-based production technology (CBPT) for software. Production is then divided into design (programming, up to the compiler stage), and manufacturing (after the compiler, and including installation into the host environment). (We are using the term 'technology' in a sense whereby the assembly line would be called a technology for manufacture of mechanical equipment.)

There are a number of problems with this compiler-based production technology for software, both from practical and theoretical points of view:

(1) Testing of a design is usually done by running down the whole manufacturing line. Debugging compilers and incremental languages (such as Lisp and APL) are the only exceptions.

(2) A considerable number of systems need to be used in this production process, such as compilers, text editors, job control languages, etc. Each has its own language, which is a nuisance for the operator, but more significantly, the information manipulated by the separate systems does not interface well. For example, the fact that a call from one subroutine to another, is specified by having the source code of the caller explicitly mention the name of the callee, severely restricts the flexibility of the post-compiler tools.

(3) CBPT is not well adapted to situations where a producer wants to have one product which is realized in a number of similar software installations, i.e. a number of variants of a piece of software which are to run in different environments, and which have partly different specifications, but which also have great similarities. (Other complex technical systems, such as aeroplanes or power stations, usually have that character). In such situations, one needs to distinguish a design stage, where a common architecture and a reportoire of exchangeable modules for the product are built up, and a manufacturing stage where modules are plugged into the architectural frame according to a specification explicitly or implicitly provided by the customer. CBPT allows such a distinction in principle, with a set of source code programs for various modules as the module reportoire, but does not seem to support it in practice. Lisp and similar programming systems are not oriented towards supporting the distinction between design and manufacturing either; the main idea in the SCREEN system is to extend Interlisp to providing that support.

(4) The classical thinking about 'structured programming' and 'software engineering' argues for modularity for the purpose of the design phase. But it is more important to have modularity for the purpose of manufacturing and maintenance, so that modules can be exchanged while an installed system is in

operation. The fact that classical 'structured' programming languages provide separate compilation of procedures only very half-heartedly, is a symptom that this aspect of modularity is ignored.

(5) The significance of operating systems for software structure is overlooked. Textbooks about operating systems focus only on their role as administrators of shared resources (e.g. Brinch-Hansen, 1973, ref. 6). Even papers on the approachment of programming languages and operating systems tend to emphasize how programming-language concepts can be imported into operating systems (Jones, 1977, ref. 7). This trend overlooks and buries the fact that operating systems are large software structures which are designed in order to fit a class of applications, and whose structure allows modules to be added incrementally, which gives them a crucial role as the skeleton of every complex piece of software. Unfortunately, because of this neglect, they still perform that role in a very primitive fashion.

Software development tools such as program generators, programming languages implemented by pre-processors, or the Programmer's Workbench (Dolotta et al., 1978, ref. 8) attempt to solve some of the above problems by embedding the compiler in additional shells of software for administrating software, but the basic assumption of having a compiler as the central tool, creates artificial separations which make for an overly complex technology.

So much for the compiler-based production technology. Programming environments for incremental programming languages, such as Lisp and Apl, remedy some of these problems, but they do not provide support for a separation of design and manufacturing.

The technology described in the previous section approaches these problems by an extension of the programming-environment approach, and we shall therefore call it the Environment Based Production Technology (EBPT). With this technology, the following method would be used for implementing a new 'product', i.e. a family of similar software systems:

(1) Investigate the market and specify the range of desired systems, i.e. the desired flexibility of the design

(2) Design the standard EUE for the product, i.e. the common frame for all delivered software installations. It may be viewed as a special-purpose operating system, for the product. Using a term from other branches of engineering, it may be called a ***chassis*** onto which installation-specific modules may be mounted.

(3) Design a reportoire of modules which may be fitted into EUE:s, and/or program generators for such modules. Also, design languages for describing specific installations, i.e. for expressing customers needs, and implement tools which translate from that/those languages, to an appropriate selection of modules, and to parameters to those modules, and to programs which have been generated from the description language

(4) For each order, attempt to express the customer's specifications in the description language, and to manufacture the desired system by attaching selected modules (previously existing, or automatically generated from specifications) to the chassis. If unsuccessful, generalize the product by adding modules and/or extending the description language, without sacrificing what was in it before.

(5) Retain the specifications for the order, in the software tool that was used to generate the delivered system. Throughout the life-cycle of that software system, requests for extensions or modifications are serviced by issuing change requests to the generating software system, which will then send the necessary updates to the generated and delivered system.

The *SCREEN* system is an experimental software tool for supporting that production technology.

4. The structure of an environment from the user's point of view

The strategy of repeatedly sending updates from development environments to end-user environments, makes it necessary to have a well-defined global structure inside environments. The need arises both in order to help the user find his way around in a changing environment, much like the street map in a city serves as a framework for the inhabitant in spite of the construction that takes place locally. The need also arises in order to make it possible for a message which is sent from one actem to another, to attach its contents to the right places in the receiving environment.

The concept of a *context* provides a structure of the environment's contents from the user point of view. The idea is simple: the user dialogue in the EUE will first offer the user a choice between several different contexts (using a menue technique or a command technique), and then inside the context there may be one or more additional levels of choice. For example, the environment for a secretary may contain one context for "address directories", with sub-commands such as:

- new entry in address directory (person and address)
- delete entry
- change of address
- add person to (one of several possible) mailing lists
- delete from mailing list
- produce labels for mailing list

The tree structure of contexts, and commands (or other choices) within contexts, is represented explicitly as a data structure not only in DE:s but also in EUE:s. In that data structure, each context name has attached information about what commands are available in it, what local variables have to be bound, and what expressions have to be executed when the context is entered, etc. Furthermore, this information is specified using a frame-oriented data base, i.e. it can be given explicitly, inherited, computed or augmented using data-driven procedures, etc.

On lower levels below contexts, the user frequently encounters other kinds of choices, for example the choice of agenda features which were given as examples in section 2, or the choice of format (letter, memo, contract, etc) that is offered by the text handling context, or again the choice of languages supported by the same text handling context. (Natural languages differ with respect to e.g. choice of alphabet, hyphenization rules, and choice of standard phrases generated by a text formatter, such as the word "Chapter"). These choices are also represented by named objects.

Thus, the general strategy when software is grown, is to arrange it so that concepts which are known by name for the user and the programmer, or which can easily be named, are also introduced into the environment's data base, associated with the (declarative and procedural) information which is needed to characterize it; and this information is looked up each time it is needed. As far as possible, then, new features are added by introducing new named objects of already existing types, and providing them with the information that characterizes them.

Consequently, the end user who operates an application will very frequently encounter situations where s'he has a choice of options, be they commands, formats, data set, language, or whatever, and where changes during the evolution of the software are often merely reflected by additions to a manue, or warning texts that a certain, previously existing item has been changed. This provides a natural structure whereby the user may understand the system, and at the same time as it serves as the backbone of the software architecture.

5. Software structure within environments

Environments contain programs and data, which exhibit the traditional structures such as the invocation structure of procedures, or the relation of data to their description (declarations) which are of course stored in the DE. Such structures are often independent of the borders between contexts, since procedures and data often have a generality which allows them to be used for several user services.

The procedural contents of actems are designed using the technique that is wide-spread in the Lisp community, i.e. to use general routines with many parameters and procedural handles. Such procedures define a certain general behavior, whereas the details of the behavior, or exceptions to it, can be defined using procedures which are associated with objects which are known to the routine, or which are given as arguments to it. These objects may be the user-oriented concepts described in the previous section, but very often they are also concepts which are internal to the software, for example concepts used in the declarations: type names, slot names (analogous to field names in records), etc.

In accordance with ref. 9, we use the term *superroutines* for such general routines which are parameterized on information stored in a data base, and which may dispatch to procedures stored there. Each SCREEN environment contains a limited and fairly fixed reportoire of superroutines (for command dialogue, form-oriented dialogue on the screen, report generation, search in the data base, etc., as well as for a number of system-oriented purposes), together with a large structure of conceptual entities which stores information ranging from specifications and documentation, to information which directs the superroutines.

In the data base that stores the parameters and data-driven procedures for the super-routines, the elementary piece of information is the ASSIGNMENT of a value to an OBJECT. (This is an assignment in a static sense - the object has a value - rather than in the dynamic sense of an assignment operation such as the algolish :=). One object may have several SLOTS, and an assigned value in each of the slots.

Therefore, the modules that are sent between actems e.g. to contribute a new service, or an improvement on an existing service, are not always new programs: quite often they are parameters, additional objects, and additional slots or slot assignments that make use of existing superroutines. This does not mean that the modules are un-procedural, because some of the slot assignments may be data driven procedures, but their main structure is not that of a program.

Since the number of objects and slot assignments grows quite rapidly, conceptual tools and software tools for administrating it are necessary. *SCREEN* therefore suggests and supports a certain uniform structure on the data. This structure is useful both inside actems, and for defining the extraction of modules from actems and the incorporation of modules into actems. The structure is based on two orthogonal concepts, the *block* concept and a fairly conventional *type* concept.

The type concept is the familiar one: each object has a type, which implies the existence of certain slots in the object, and sometimes also restrictions on the possible value in the slot. We allow for both the conventional type mechanism, where the type name is a new object whose type is "types", and for A.I.-style IS-A hierarchies where slots and slot values may be inherited from superior objects in the hierarchy.

Some measure of the need for structuring the data base is given by the fact that presently almost 100 types have been defined and used in the course of developing the *SCREEN* system and the applications that are presently implemented in it.

The mechanism of adding more named objects is however not always sufficient as a tool for organizing updates and the transmission of updates between environments. In general, an environment provides a framework within which modules may affect each others' behavior: one module with a certain 'standard' behavior, obtains a modified behavior because of the presence of some other module(s). The simplest case is when the behavior is dependent on the presence of certain objects of certain types in the actem, and the modifying module may add more objects of that type in such a way that they are visible. The addition of contexts or commands is an example where the behavior of the general-purpose command-mnue dialogue handler is affected.

That mechanism is natural when there is a named object, which is needed anyway, and to which the incremental information can be attached. However, sometimes there is no such object. The mechanism of INCREMENTAL ADVISING may then be useful. With this mechanism, a procedure may contain a named ADVISE POINT, i.e. a point to which incremental code ('advise') may be added, and other modules may contain advise to the advise point. Each piece of advise therefore has the form 'module M gives advise to advise point P that <code>'. (The module here is in fact a block in the sense of the next section). Several pieces of advise may be given from different modules to the same advise point, and these pieces of advise are re-separated to their respective sources when charts are generated for each of the advise giving modules.

Advise points may be either named objects (like ordinary procedures) or slots of named objects which are used for other purposes (like data driven procedures). This means that contexts, commands, and many other types of objects may have one or more slots to which incremental advise can be given.

Incremental advising is used for many purposes in the software for the *SCREEN* system itself. For example, there are advise points for the entry and exit of a context, for temporary detachment from an environment (Interlisp 'sysout' function) and for the corresponding resumption; for the regeneration of an actem, and so forth. Various services give incremental advise to these advise points; for example, the resumption of an environment has incremental advise for bringing it up to date with what may have happened while it was asleep, e.g. picking up various kinds of messages, as contributed by modules that make use of the respective kinds of messages.

Several application oriented routines which may receive advise have already been mentioned, such as the personal agenda service and the mail receiving service. These are usually implemented using either of the two mechanisms that have been described, i.e. adding new objects with information attached to them, or incremental advising.

6. Global software structures in the operation of the system

The concept of environment is actually somewhat different as seen from the user and from the system. For the user, an environment is one software individual with which s'he interacts when s'he turns on the computer. For the system, an environment is a data set containing procedures and data, represented at least between sessions as a file in the directory. (In Interlisp, such a 'system' environment is of course a 'sysout', i.e. an image of the virutal address space).

The reason why these two concepts are not completely equivalent is that sometimes one 'system' environment needs to make subroutine-like calls to one or more other 'system' environments, while still giving the user the impression of communicating all the time with the same software individual. This is useful in order to allow several environments to share space consuming or often updated services, and not only because of the limitations of the DEC-20 address space.

When the distinction is significant, we shall reserve the term 'environment' for the software individual perceived by the user, and we introduce the term 'actem' (for 'active system') for the environment in the software sense, thus avoiding the implementation-specific term 'sysout'. Much of what has been said in previous sections about environments applies in fact more precisely to actems. In particular, the characterization of *SCREEN* as a system of communicating environments is slightly imprecise: what we have today is a system of communicating actems. Actems are the significant entities from the point of system architecture, and in particular it is actems that are able to send and receive messages that contain programs or other data.

Actem-to-actem communication is necessary for purposes of software maintenance. It will probably also be desirable to superimpose environment-to-environment communication for more user-oriented purposes, but that has not been done yet (except for specific services such as mail).

It is useful to think about actems as individuals, which may have a considerable life-length; which may interact with human users or with other actems; which may transfer their knowledge to other actems; and which also may refer a human user to another actem which is to render specific services, when necessary. At least so far, the environment has relatively little cohesion: it is a dynamically formed tree of actems which have happened to invoke each other.

As usual in Interlisp, there is a duality between actems on one hand and 'makefile files' (i.e. files that contain textual representations of program structures and data structures from actems), on the other hand. We use the term 'chart' for such a 'makefile file'. Thus each actem usually maintains a set of equivalent charts, regenerates by re-loading those charts into a fixed 'base actem' (e.g. the Lisp system) and transfers information to other actems by telling them to load certain charts, and (in most cases) to augment their chart list with the newcomer, for regeneration purposes. However, the charts in *SCREEN* has additional structure besides the Interlisp standard, for several purposes including the support of these transfers.

Some information must be available in common for several actems, for example the description of the structure of actems and the charts they contain. Besides actems and charts, *SCREEN* also uses 'data bases', each of which is a mapping from atoms or pairs of atoms (similar to a property-list), to S-expressions or strings. Implementation-wise, each data base is a direct-access file ('file' in the sense of the operating system) with a specific structure. Often the strings are 'semi-texts', i.e. 5-10 lines of text for description of a procedure, command, or other named

object. In particular, there is one SYSTEM DATA BASE which contains descriptions of all actems and charts (and of members of certain other categories to be described below), and a PRIVATE DATA BASE for each user which contains information about him that he wants to make public, or for some other reason he does not want to keep in his private actem.

The data bases are also used for purposes similar to the 'blackboard' in the HEARSAY II system (ref. 10), i.e. as a global means of communication between actems.

In principle, data bases could of course have been implemented as just another kind of actem, using message-passing for each data base access, but in the TOPS-20 and Interlisp environment that was considered as too expensive.

Finally, besides actems, charts, and data bases, an application also consists of a number of text files for documentation purposes, both for end-user documentation and programmer (systems developer) documentation. Documentations usually consist of a fixed text part, which covers general introductions to a program and the concepts used therein, and variable parts e.g. descriptions of specific procedures or commands. The information for the variable parts is stored in data bases and is aggregated by a special program.

Since different environments contain different sets of information, there must be some mechanism for arranging that each environment contains the right set of information. This is done by collecting slot assignments into *blocks*. (This concept is not reminiscent of algol-ish blocks). In principle, a block is a set of slot assignments, i.e. a set of object/slot/value triples. When the block appears in a chart, these triples are printed textually after each other, i.e. the block is defined by physical proximity. In actems and data bases, on the other hand, the slot assignments are dispersed, and the block is represented as a catalogue, which is stored in a slot of the block name, and which specifies which objects the block has as members, for each of several types.

In the very simplest case, then, the contents of a block in an actem or data base are defined by three nested loops: for all types t used in the block, for all members m of t in the block, for all slots s that m may have in type t in the block, according to declarations stored in the data base, operate on <m,s>

Several superroutines in *SCREEN* make wholesale operations on blocks according to that pattern, for example for transfer between actems, data bases, and charts, for data entry, presentation, and for checking of data. Also, the purpose of having the right information available in each actem is served by providing it with the right blocks (rather than by feeding one object or one slot assignment at a time). In order to keep track of dependencies between block, and to determine which blocks are needed in which actem, there is a higher-order structure which will be described in the next section.

Each routine (e.g. each superroutine in the sense described above) assumes a certain structure, not only in its object data, but also in its parameters. Since modules that are transmitted between actems often consist essentially of parameters for superroutines, this stipulated structure is significant for the modules. We take the following view: the superroutine is a block, each set of parameters (of which there may be seeral) is another block, and the declarations that describe the structure of the parameters, is either included in the block of the superroutine itself, or is a separate block, but in a one-to-one correspondance to the superroutine block. There is a META relation between the parameter block and the block that contains the declarations.

That structure is sufficient when new objects are introduced in the parameter block, together with assignments to some slots. But it also happens frequently that a superroutine wants its parameter blocks to contain additional slot assignments for objects that existed before. It is particularly common that additional slots are assigned to type names and slot names. For such situations, we use OVERLAY blocks. An overlay block is named not by a single object, but by a pair of objects ⟨M B⟩, where B is a previously existing block whose catalogue is also the catalogue of ⟨M B⟩, and M is an operator which specifies for each type, what slots shall be defined for that type in the block ⟨M B⟩.

An example of these structures is given in the appendix. Let us just make two general observations:

(1) The different relationships which exist within the actem, such as the block-metablock relation, the object-type relation, the block-members relations, and so forth, are all used in various way by the superroutines in the actems. There is no single hierarchy in the actem;

(2) We have given the impression that a block contains exactly the same set of slot assigment, in actems, charts, and data bases. This is not completely true: handles on the dumping and reloading procedures enable some slots to stay in the actem, and others to be generated for the chart, or be created when the chart is loaded. This is significant as one of the methods whereby the contents of blocks in a chart can be strongly integrated into an actem, and later extracted again to form a clean module (with no extraneous information) that can be sent to another actem.

7. Software structure for purposes of software management

From a programming-language perspective, one might describe the contents of an actem as "a program organized by attaching program fragments to named objects, and invoking them from superroutines". However, another description which may be more useful is to say that it is "a data base for storing specifications, descriptions, documentation, and declarations of software, and within which one also adds some procedural information in order to provide the detail that is necessary for actual execution". In this section we take the latter perspctive.

Some documentation has a local character, and may be associated with each command name, procedure name, or other similar entity. Examples of local documentation which is presently maintained by the *SCREEN* system is

- help text associated with command names and other user-oriented concepts
- commentary describing the purpose of a procedure
- an accumulated sequence of notes about the maintenance of procedures or other similar object, including bug complaints, notes about desired improvements, and notes made when bugs have been fixed and improvements performed.

However, there is also a need to associate documentary information about larger software structures. For each software development undertaking, we want to store which person is responsible for it; the relationship to the text file(s) for the free-text documentation, and so forth.

For these purposes, we have a higher-order structure of FACILITIES and TOOLS. Each tool consists of a set of charts (irrespective of the actems that the charts may be in), and a facility is a set of charts and tools. Each facility and each tool is (represented by) a named object, just like the various types of user-oriented objects.

Thus we have a heterarchical structure, which contains several hierarchies that provide partial structure from different points of view. Actems define one hierarchy, and facilities and tools define another. The following analogy may be helpful: Each actem is like a city, which contains several houses (= charts). Identical houses may appear in several cities (e.g. for certain restaurant chains; = the same chart is loaded into several actems). A corporation (=facility, tool) is represented by a set of houses in one or more cities. Identical houses in different cities belong to the same corporation, but one corporation may also have several kinds of houses.

The semantics of these concepts is that a facility should be the result of one programming undertaking; something which is designed and developed as an entity, and usually by one or a small number of people. A computer mail handling system, and a text formatter, are examples of facilities which have been developed within *SCREEN*.

In some cases, the facility is implemented only as a set of charts which are direct members of the facility. But sometimes, in the course of developing a facility, one also develops certain modules which serve a defined technical purpose, but which may also be separated out and used for other applications. For example, the data base routines used in *SCREEN*, were developed as an auxiliary for the mail handling facility, but have later also been put to other uses. Such parts of a facility are called TOOLS. We select to keep the association between the tool and

the facility where it was first developed, even though it may later be used by other facilities as well, simply because that encourages the user to identify possible tools within what at first seems as just a homogenous application. The association between the tool and its originating facility is then maintained as long as it reflects the actual responsibility of maintenance. Later on, if the tool obtains an independent life of its own, it may be transferred to another facility.

To account for tools which have become separated from their originating facility, or never had any, we have certain 'dummy' facilities which are only introduced for the purpose of containing tools.

The structure of facilities, tools, and charts is documented in the system data base, along with other global structures. In addition, each facility has its own DOCUMENTATION DATA BASE, which contains miscellaneous information about the entities in the facility, such as brief texts describing the purpose of each entity.

Orthogonally to the distinction between tools, the charts in a facility may be divided into several SEGMENTS, in order to make explicit which subsets of the charts are used by different actems. This will be further discussed below.

RESPONSIBILITY FOR CHARTS. The information in a chart changes over time, as the system changes in response to new users' needs. As usual in Interlisp, each actem has the responsibility for maintaining a consistent set of charts. The updated chart must then also be distributed to all other actems that contain the chart. Although in principle it is possible to allow all actems to change all charts they contain, it is usually better to let each chart have a 'home' actem, where the maintenance on the chart is made, and from which updates are distributed. Also, it is usually preferable to let all charts in a facility or tool, be supported from the same actem. We refer to that actem as the DEVELOPMENT ACTEM for the facility (or tool, or chart).

Each actem must contain information about what charts are contained in itself or in some other actems. That information is needed for several purposes, including:

- for documentation
- for distribution of updates of charts
- for determining which charts are to be used for regenerating an actem

However, since charts are fairly often added to a facility, when they are needed for some new feature, it is impractical to represent the explicit chart list, and we would prefer to specify only the facilities that are in the actem, and be able to derive the set of charts. On the other hand, actems often need only a subset of all the charts in a facility. The charts in a facility or tool are therefore divided into SEGMENTS, so that each chart belongs to exactly one segment, and each actem is defined to contain certain segment(s) of certain facilities. As long as additional charts in a facility fit into one of the existing segments, no actem descriptor needs to be changed.

There are also cases where a development actem will generate charts with information which is never re-loaded into the development actem, but only sent to certain other actems. That happens in particular when programs and data may be re-organized in order to increase performance in the receiving ELE, for example by shortening indirect access paths. Presently, such re-organization of programs is done using compiler macros, and with the regular Interlisp compiler as the driving program. Such generated charts have their own segments in a facility.

8. Conclusions.

The *SCREEN* system has been used as the basis for a significant amount of software, including a number of specialized software tools and a number of applications. At the same time, it is an experiment with a novel way of organizing software and software production. It is dissimilar from the conventional paradigm of Lisp systems, since it separates the programming environment from the user environment, and is designed to support the continous traffic of modules from one or more PE:s to one or more EUE:s. It is also dissimilar from conventional, compiler-based software production technology, since it introduces a new framework in place of the conventional roles of the compiler, linkage editor, and operating system.

The software architecture of *SCREEN* requires several kinds of depositories of information (actems, charts, and data bases), and a number of intertwined structures, within and across the borders between these depositories. These structures are different from the usual structures in classical programming languages, and so-called structured programming, in several ways:

- the described structure does not consist of one single, strict hierarchy. There are several hierarchies that intersect each other, such as facilities/tools/charts/blocks, types/members, actems/charts, the meta relationship and other relationships on blocks, generation and contribution relationships between actems, etc. Only one of them is textually explicit in the chart representation (by necessity), but they all exist in the data-structure representation in the actems, and various tools that are regularly present in the actems trace down one or more of those paths

- the structure described here relies very heavily on program/data equivalence. Objects that represent types, procedures, blocks, charts, contexts, commands, and so forth are all parts of the program structure, since they are used to describe data and/or are associated with one or more executable procedures. However, they are also used as data, both in the sense that data management tools are available for the programmer for operating on them, and in the sense that many of them occur in input and output when the program is executed for the final user.

RELATED TOPICS. Relating back to the discussion of environment-based production technology for software (EBPT) in section 3, the following issues are of particular interest when designing the software support for EBPT:

(1) Architecture of the DE and EUE
This includes issues of how to represent programs and other knowledge in the data base of the development environment; generally useful support for the reportoire of special-purpose languages; issues of modularity and other global structures in in the data base and the programs; and the structure of the supporting software for the two kinds of environments

(2) Dialogue techniques
Good methods for dialogues are essential in both DE and EUE. Dialogue techniques include special-purpose languages for describing dialouges; techniques for mixed-initiative dialogue and natural-language dialogue; etc.

(3) Facilities in the EUE
This includes 'fancy terminal' techniques such as graphics, multiple windowing and multiple screen techniques, non-standard keying devices, etc.

(4) Operations on application models
By virtue of the special-purpose languages and the dialogue techniques, models of the application are built in the development environment. Some significant operations on these models are for simulation and other performance evaluation, for generating data base schemata for use in a data base for the production system, and for non-trivial generation of program modules for the production environment.

The present report has covered the first of those issues, by describing the *SCREEN* system.

APPENDIX

A figure and an example may illustrate the block-structure concepts introduced in section 6. We consider two facilities: one for text formatting (such as runoff) and one for administrating a directory of publications (published papers, technical reports, books, etc.) of interest to a research group. In the development actem used by the formatting facility, there are objects in at least two types: procedure names (in the formatting program), and names of commands to be used in the input file to the formatter. The actem also contains information about what slots are defined for each type.

As the facility for text formatting evolves over time ("is maintained"), the set of objects and their assignment in the development actem changes. More precisely, the way for the programmer to change the system is to interact with the development actem, change those assignments, and add more objects.

When the programmer wants to find out what is currently in the facility, he may either inspect specific assignments (interactively, at the terminal) or ask for a listing. In the latter case, it is often natural to command a listing of all assignments in a block.

As much information as possible in the facility is organized in terms of the objects. For example, consider what happens when we want to augment the formatter with support for multiple alphabets, in such a fashion that text may be entered and edited on conventional, single-alphabet terminals, but output in formatted form on devices that permit multiple alphabets. We then need to add an alphabet switching command. This is done by entering a new object and assignments to some of its slots, for example code to specify what to do when this command is encountered. (In fact, one needs at least two such procedures, one to be executed when the command is encountered as the input text is scanned, and another to be executed when the formatted line is output to the result file).

Furthermore, we need transcription conventions, so that e.g. a German ä may be represented according to conventions as "ae" in the input text, or so that a Russian ш may be represented as "sh". Thus we introduce a type for alphabet names, with a slot for the set of characters and another for the transcription conventions. This is sufficient if there is only one such convention for each language. If there are several, we need to introduce names for the different conventions, which means we need a type for such conventions names, and slots to indicate for each convention what alphabet is transcribed into what, as well as the details of the convention. Furthermore, we need additional commands for selecting conventions.

As another example, different languages need different hyphenization algorithms, so besides the type of alphabet there is the type of language (of the written text), the obvious relations between those (several languages may use the same alphabet, and occasionally the other way also), a source-file command for selecting language, and a slot in the language name for "its" hyphenization procedure.

After some time, the formatting facility has grown to include a considerable number of things. The need arises to provide subsets of it for certain users.

The sets of all objects and assignments have therefore been divided into blocks right from the beginning, with the intent that each specialized version of the facility can select some of the blocks, but not parts of blocks. For example, the multiple-language option will use one or a few blocks, which contain the

descriptions for each of the supported languages (probably in different blocks, so that one may pick one's choice), the alphabet information, the declarative information (meta-information) which specifies the structure of the descriptions of languages and alphabets, etc. Furthermore, they contain some additional assignments to "underlying" objects. For example, commands which print out canned texts, such as the command for generating a chapter heading, or the command for selecting a format for page numbers, will generate texts such as "Kapitel 4" or "Seite 85" in a single-language German system, but need to be parametrized for language if several languages are allowed. This additional slots for previously established objects also go into the blocks for the multiple-language facility.

The development actem contains knowledge about
- the names of the (defining and generated) charts of the facility;
- for each charts, which blocks it contains;
- for each block, which members it has of different types
- for each type, which slots are defined for objects of that type plus some other information to account for blocks that contain assignments for non-members (overlay blocks). This information together with the assignments themselves, contained in the actem, are enough to generate the charts.

We can now turn to figure 3, which shows how objects of different types reside in the development actem for the text formatting facility, how they are grouped into blocks already in the actem (although the borderline is not noticed in the internal operation of the actem), how the blocks go into charts which may be sent to EUE:s to augment their operation. Figure 4 shows how the full set of charts may be loaded into a base actem to re-generate a development actem. The figures assume a simple case of three charts: one for the regular set of commands (R), one for the extended set of commands using multiple languages and alphabets (L), and one for services used by the person in charge of the formatting facility, for example documentation information (M). The R chart contains the following blocks:

R contains the procedures of the (basic) formatter, together with declarative information about e.g. what slots are defined for formatter commands;

C contains the commands of the (basic) formatter, organized according to the conventions specified in R

The L chart contains the following blocks:

L contains the additional code, as well as the declarative information about new types, as well as additional slots for old types

LC contains the additional commands for supporting languages and alphabets

LE contains information about the English language and its alphabet

LS similarly for Swedish

L<C> contains the additional assignments to objects in the block C, as specified by L

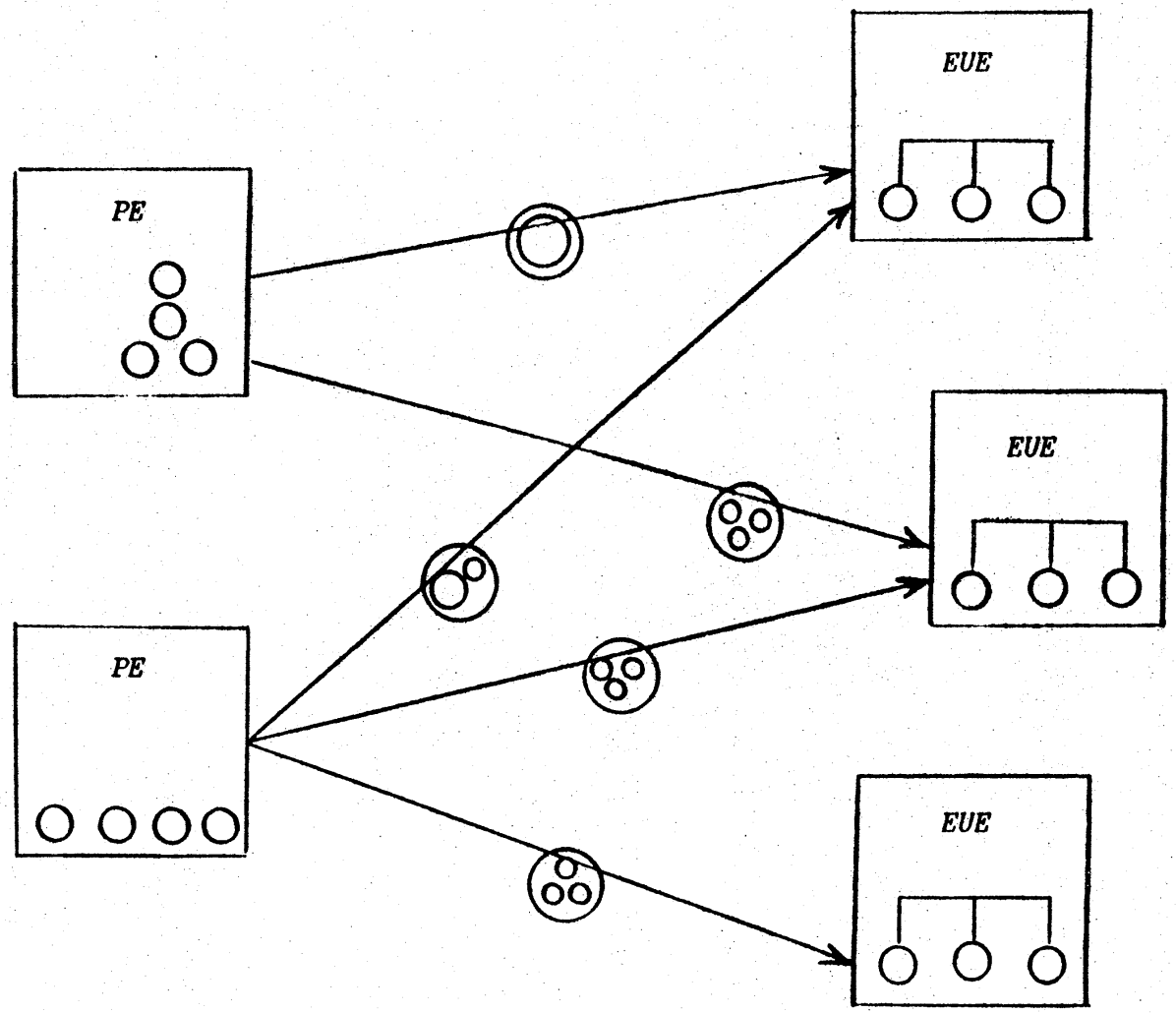
In this example, there is an obvious 'meta' relation between each block and the block(s) which specify the structure of its contents, namely:

from C to R
from LC to R
from LE to L
from LS to L
from L<C> to L

Notice that this 'meta' relation establishes another hierarchy than the one that goes facility - tool - chart - block, just as the object - type structure (and the 'ISA' property inheritance structure) run independently of the block hierarchy.

It is probably natural to have one single context for the text formatting facility. The object representing this context naturaly belongs to the block R (not to C; one way to see that is to notice that there may be several blocks C1, C2, ... giving commands in different languages, and then no one of them is guaranteed to be in the service actem). When the block R is sent to a service actem, it augments its menue of contexts as described above.

The user dialogue within the formatting context consists of asking the user for the name of the source file and the desired name of the result file, as well as for some parameters which one might not wish to keep in the source file (output medium, perhaps also choice of fonts). The procedure for that dialogue is of course in one of the slots of the object representing the context.



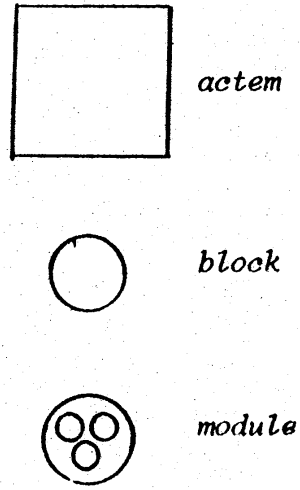


Figure 2