Retained objects and operating system interfaces

K. Hopper

Department of Computer Studies, University of Leeds, Leeds, LS2 9JT, UK

The general nature of all types of object which a user may wish to retain between computing sessions is shown to be that of a file. The requirement that a transportable control language provide identical semantic effects everywhere leads to the idea of a universal file system. The specification of a file in such a system is discussed before considering the effects of widely varying underlying file systems on the mappings and facilities needed in a control language processor to support such a universal file system. Discussion of file manipulations and network connection requirements completes the design requirements for the use of retained objects in a control language processor.

(Received February 1980)

Introduction

The word interface, when used in relation to computer operating systems, is often associated with two particular boundaries—that between users and the operating system, and between users' processes and the operating system, since these are the two boundaries with which the majority of computer programmers come into contact. If the operating system is considered only to be a specially privileged process, however, then three separate operating system interfaces can be identified: (*a*) the processor hardware, (*b*) the processor peripherals and (*c*) processes. While purists may wish to suggest that the first two should be considered to be one interface, it is argued that peripherals are outside the processor and therefore fall into a different class. Although they are indeed a hardware interface they are the data communication devices for input, output or both on behalf of any process.

Peripherals are therefore very much concerned with any object retention which a process may wish to do. The physical peripherals themselves must therefore be considered as retained objects—objects retained from one computer use to the next—since the operating system must know about them and their properties in order to be able to use them for communication purposes.

The suggestion that the third operating system interface is a boundary with processes is not meant to imply that a user is one of those processes which interfaces directly with the operating system. It does imply that users must have available a communication process which has a direct operating system interface, enabling them to instruct and receive responses from the operating system. This process will be called the control language processor (CLP). The two interfaces which this CLP has—with user and with operating system—are both of interest when discussing operating system interfaces since the CLP must convert between the user's view of actions or responses and the operating system view of them. These actions and responses may concern such things as controlling other processes, doing calculations or otherwise manipulating various types of object; in particular they will involve the manipulation of objects which may be retained between computing sessions on behalf of the user. The provision of a transportable control language (TCL) in this context implies that the CLP must provide a mapping between a 'universal' view of computer systems understood by the user using the TCL and a particular view of a particular system known to the underlying operating system.

The provision of a universal view for all users using the TCL has certain profound implications for the retention of objects between computing sessions. First, the operating system must retain its own view of these objects between sessions and second it must also on behalf of the CLP retain the user's (TCL) view of the same objects—all in addition to retaining the actual objects themselves!

Retained objects

From the computer users' viewpoint the objects which he is most likely to wish to retain are files containing data, programs, etc. As already suggested the operating system will wish to retain peripherals or, perhaps more accurately, at least its description of peripherals. Users may also wish to retain peripheral descriptions, accounts, budgets or even complete abstract machines from one session to another.

Whatever the semantics of the objects to be retained may be, or whoever wishes to retain them, the act of retention involves permanent storage of some sort. The name conventionally given to storage entities is file and therefore the discussion of retention of any type of object between sessions reduces to a discussion of retaining files, the nature of these files themselves and the mappings which must be provided by the CLP.

Universal file system

In order for a TCL to have standard semantics for retained objects and their manipulation it must view them in a standard way. Before considering the way in which retained objects may be viewed by an underlying operating system it is essential to describe this standard TCL view before discussing the precise nature of a file as seen by the CLP.

The universal file system provided by the TCL should provide all the facilities which any possible user may require when using his computer (whether or not it provides all of these facilities for all of its users). This very sweeping statement has, however, some natural restrictions which may not at first sight be obvious:

(*a*) The TCL user, particularly in a network environment, may have no knowledge of the underlying physical storage facilities provided for retained objects. A TCL can never provide standard facilities for describing these because they are highly machine dependent; the TCL would not be transportable if it did provide them!

(*b*) No TCL can provide direct access facilities to any contents of a retained object. It may, of course, include a mechanism to call procedures (which are machine dependent in content but transportable in use) in order to appear to provide this facility for certain classes of user. The TCL must, however, directly provide facilities for describing the contents of a retained object, i.e. it must provide the description which enables the machine dependent access procedures to correctly access the retained object's contents.

CCC-0010-4620/81/0024-0331 $04.00

(*c*) As a corollary of the inability of a TCL to provide direct access to the contents of a retained object, it can have no knowledge of the meaning of those contents. In particular, it can never provide the user with a guarantee that the contents correspond with the description given to the universal file system. The assurance of this correspondence must therefore remain with the user and the way in which he fills the retained object.

As a result of these natural restrictions, the universal file system can only provide a set of logically homogeneous retained objects which are containers for some sort of data. Each of these containers has certain properties (to be discussed in detail later) in accordance with which the CLP allows the user to handle them, irrespective of whether the allowed handling is meaningful in terms of the actual container contents! For each existing retained object the universal file system implementation within the CLP must therefore maintain the following property information on behalf of the user (irrespective of the facilities which the TCL provides for any particular user or class of users):

(*a*) *Unique identification.* This must be sufficient to separately identify every container in the system. Where a particular computer system is a node in a network then this identification must also contain the node identification in order to ensure proper homogeneity throughout the network.

(*b*) *Size and shape.* This is the logical description of the contents—the way in which the user wishes the CLP to handle the retained object (e.g. a retained object which is a file of compiled code which the CLP may therefore allow to be executed).

(*c*) *Owner and access.* Although the identity of the owner of a retained object is normally likely to be considered a part of the unique identification, it is strongly related to the accessing facilities. The CLP can only provide secure handling of retained objects if the universal file system restricts access to every object in accordance with the wishes of a particular user whom it considers to be the owner. The CLP must therefore enable this owner to give whichever access permissions he wishes (e.g. READ, WRITE, SHOW, DELETE etc.) to any other users in the system (or in the network if applicable). The universal file system must therefore also retain this access information as a property of each retained object.

The question of how the universal file system organises the sets of properties for all retained objects in the homogeneous store which it provides is not relevant to the subject of this study although some form of hierarchy by owner will probably be the most efficient. What is important, however, is that all the property information about the homogeneous store is semipermanent and must therefore itself be retained on a long term basis—as a retained directory object within the homogeneous store! (NB the use of the term semipermanent is intended to imply that property information is subject to change from time to time and is not permanent in the sense of being immutable.)

The nature of a retained object

The description of the universal file system suggests that a retained object is a container with some three groups of properties—so far described only in general terms. Consideration of the mappings required by the CLP and the two differing views of retained objects maintained by the user and by the operating system requires the development of a complete formal specification of the retained object as seen internally by the CLP before these mappings can be properly described.

The specification of a retained object is built up from considerations of the ontology of data which the user may wish to store. In the arguments which follow it is worth noting that they apply to any retained object whether or not computer storage and use is involved.

Unstructured data elements

Definition

At each level of discourse about any subject there exists one or more indivisible entities about which it is not necessary to know any finer detail to continue a 'discussion' at that level. At any level, therefore, these entities may be referred to as elements, whatever their fine structure may be at some lower level. Those elements which are data (DE) exist in a domain which is the cartesian product of their name and value, where value itself exists in a domain which is a product of four other domains—abstraction, denomination, mode and relationships. That is

$$\text{DE} = \text{IDENTITY} \times \text{VALUE}$$

where

$$\text{IDENTITY} = \{\text{name of objects in the subject of discourse}\}$$
$$\text{VALUE} = \text{ABSTRACTION} \times \text{DENOMINATION} \times \text{MODE} \times \text{RELATION}$$

where

$$\text{ABSTRACTION} = \{\text{the conceptual ideas involved}\}$$
$$\text{DENOMINATION} = \{\text{the names given to the abstractions}\}$$
$$\text{MODE} = \text{TYPE} \times \text{UNIT}$$

where

$$\text{TYPE} = \{\text{classes of abstraction}\}\ [\![\text{e.g. truth, number etc.}]\!]$$
$$\text{UNIT} = \{\text{scales of the abstractions}\}\ [\![\text{e.g. miles, hectares etc.}]\!]$$

$$\text{RELATION} = \{\text{relations to other DE}\}$$

where curly brackets are used to delineate a set.

Representation

At whatever level the current 'discussion' is taking place it must always be possible to represent the data elements involved in some physical way if they are ever to be stored or moved from place to place. A data element may therefore be considered not only to have a logical (abstract) existence, but also a physical one. Since it may, of course, be represented in numerous different ways it necessarily has a different physical representation dependent upon the medium in use [piece of paper, clay tablet(!) etc].

A physical data element (PDE) may therefore be characterised in terms of the medium and inscription used as

$$\text{PDE} = \text{MEDIUM} \times \text{INSCRIPTION}$$

where

$$\text{MEDIUM} = \{\text{physical media involved}\}$$
$$\text{INSCRIPTION} = \{\text{marks used to signify DENOMINATION}\}$$

The physical and logical data elements thus defined correspond closely to the concepts of datalogical and infological elements introduced by Langefors and Sundgren (1975). Following this development a little further necessitates the introduction of transformations converting from the physical to the logical forms and vice versa, i.e.

$$\text{LPCONV} : : \text{PDE} \rightarrow \text{LDE}$$
$$\text{PLCONV} : : \text{LDE} \rightarrow \text{PDE}$$

These may be thought of as *Write* and *Read* respectively since the 'thinking' mechanism (at some 'lower' level of meta-discourse, naturally) must have an 'abstract' copy of the data element about which to think. In this sense, of course, the abstract 'logical' data element has a representation of its own,

but it is a cardinal point in the remainder of this discussion that this logical representation is a private characteristic of the 'thinker' which does not concern any discussion of data storage and access beyond these reading and writing transformations [see Milne and Strachey (1976) for similar concepts in programming language theory].

Structured data elements

Definition

A commonly used name for a structured data element is *record.* Using this short name for convenience a record is commonly thought of as a group of data elements relating to some particular event, action or state. A more general informal definition must apply to all possible structured data elements and a better one would be

A *record* is a group of one or more unstructured data elements all of which bear a relationship to one other data element which is unstructured at a higher level of discourse.

With this definition in mind, the formal definition of a logical record, following the formal definition of a data element, can be given as

$$\text{LRECORD} = \text{IDENTITY} \times \text{VALUE}$$

where

$$\begin{aligned}\text{IDENTITY} &= \{\text{names of records in the subject of discourse}\}\\ \text{VALUE} &= \text{ABSTRACTION} \times \text{DENOMINATION} \times \text{ELEMENTS} \times \text{RECORDRELATION}\end{aligned}$$

where

$$\begin{aligned}\text{ABSTRACTION} &= \{\text{conceptual structures involved}\}\\ \text{DENOMINATION} &= \{\text{names given to the abstraction}\}\\ \text{ELEMENTS} &= \{\text{LDE}\}\\ \text{RECORDRELATION} &= \{\text{relations to other LRECORDs}\}\end{aligned}$$

Bearing in mind that an unstructured data element is structured at some lower level of discourse, however, the ELEMENTS of this definition and the MODE of the unstructured data element definition are seen to serve the same function of specifying the kind of object concerned, the name used here merely serving to distinguish between structured and unstructured object definitions.

Representation

Exactly in the same way that each logical unstructured data element was shown to have its physical counterpart, so must the same be true of logical and physical records for similar reasons. It is therefore possible to write, as before

$$\text{PRECORD} = \text{MEDIUM} \times \text{INSCRIPTION}$$

and, for the appropriate transformation functions,

$$\begin{aligned}&\text{LPREALIZE} : : \text{LRECORD} \rightarrow \text{PRECORD}\\ &\text{PLINTERPRET} :: \text{PRECORD} \rightarrow \text{LRECORD}\end{aligned}$$

which may be thought of as *Write*(LRECORD) and *Read* (LRECORD) in a similar way to the *Write* and *Read* discussed for unstructured data elements.

Virtual record definition

The analogy between structured and unstructured data elements is satisfactory only up to this point in the development of the argument leading to a retained object specification. Whereas a 'discussion' may either use or not use some unstructured data element, it is at liberty to consider only parts of a complete logical record as being pertinent. This partial logical record, however, is still a structured data element in its own right and may therefore usefully be given the name *virtual record* since it is never given a separate *stored* physical existence although, of course, a physical copy may be required in order to manipulate it, however transitory such a copy may be.

Although a virtual record may be identical to the logical record in any particular instance it is in general a proper subset showing only such detail as is necessary to the 'discussion' taking place. A virtual record may therefore be formally defined as

$$\text{VRECORD} = \text{IDENTITY} \times \text{VALUE}$$

where

$$\begin{aligned}\text{IDENTITY} &= \{\text{names of the records in the subject of discourse}\}\\ \text{VALUE} &= \text{ABSTRACTION} \times \text{DENOMINATION} \times \text{VELEMENTS} \times \text{RECORDRELATION}\end{aligned}$$

where

$$\begin{aligned}\text{ABSTRACTION} &= \{\text{conceptual logical structures involved}\}\\ \text{DENOMINATION} &= \{\text{names given to the abstraction}\}\\ \text{VELEMENTS} &\equiv [\text{ELEMENTS} \vee \text{ELEMENTS VALUE of logical record}]\\ \text{RECORDRELATION} &= \{\text{relation to other LRECORDs}\}\end{aligned}$$

A close comparison of this definition with that of a logical record shows that, apart from the names used, they are identical. While the notion of a virtual record in this sense is in common usage, it seems therefore that its existence really corresponds to a *separate* logical record existing in some notionally different logical retained object. The virtual record concept is therefore more properly left to the area of implementation rather than specification (see the Appendix for a brief discussion of further development of the idea for implementation purposes).

Retained object contents definition

Following similar arguments the contents of a retained object could be defined by substituting FILE wherever RECORD appears and substituting LRECORD (PRECORD etc.) wherever LDE (PDE etc.) appears. The contents of a retained object, however, are really only a structured data element at a higher level of discourse! They need not therefore be considered separately any further.

File contents access

Corresponding to the definitions given for reading and writing logical records, reading and writing of a complete contents could be defined by

$$\begin{aligned}&\text{LPFILECONV} :: \text{LFILE} \rightarrow \text{PFILE}\\ &\text{PLFILECONV} :: \text{PFILE} \rightarrow \text{LFILE}\end{aligned}$$

corresponding to the notions of writing and reading the entire contents at once.

Unfortunately, these two definitions are not realistic from a practical point of view. User processes cannot in general access the complete contents of a retained object at one time due to storage constraints. It is equally rare that a practical process logically requires the entire contents of a retained object at one time. It is necessary therefore to decompose the theoretical transformations to enable selection of a part of the contents to be made by the using process. This requires definition of the rather more practical functions *Writetofile* (LRECORD) and *Readfromfile*(LRECORD). Using functional notation the theoretical decomposition of these transformations, including the necessary selections, may be given as

Writetofile(LRECORD) = LPREALISE (LPSELECT (PFILE))

Readfromfile(LRECORD) = PLINTERPRET (LPSELECT (PFILE))

Physical access considerations

In any practical situation, each of the transformations referred to in the theoretical discussion requires some physical representation of the data in order that they may be manipulated. Each partial transformation therefore must incorporate any necessary alterations to the physical representation of the data.

The partial transformations given in the above function definitions may be described as:

(*a*) LPREALISE. Realise the shape and representation of a logical record (in the thinker's 'private' representation) as the corresponding physical record (in the physical retained object representation).

(*b*) PLINTERPRET. Interpret the physical retained object representation of a record as the corresponding logical record shape in the thinker's 'private' representation.

(*c*) LPSELECT. Select the logical record in the logical organisation from the corresponding physical organisation.

While the partial transformations defined do not form a unique sequence in which the overall transformation must be effected, they have the virtue that they separate the transformation of data shape from the transformation of organisation (selection). This particular virtue will generally enable a reduction in the number of representation translations to be made with a corresponding saving in the access overhead. With this in mind only PLINTERPRET and its converse may need built-in representation translations incorporating into them.

Retained object specification

A retained object may be specified simply as a *box* containing *something* in some (unknown) *representation*. A formal definition must incorporate these simple ideas in specifying the properties both of the box and of its contents. It must be remembered that the user cannot be forced to use the contents in accordance with the given specification of them since this will restrict his freedom of action. Since it is intended that the mechanism which makes use of the specification shall do so dynamically, the user will be at liberty to alter the specification of the retained object contents without altering the actual contents either directly using the TCL or through a program.

Remembering that the contents of a retained object may be considered as being either physical or logical (or virtual) it must be possible to include all aspects of contents definition simultaneously in the specification to ensure its completeness. It must be realised that a logical object (or a virtual one derived from it) need not correspond to any single physical container, nor necessarily to the complete contents of one or more such physical containers. This physical fragmentation, which is of no concern to the CLP, could be considered to relate to a sort of aggregate retained object which would therefore require multiple interrelated transformations. Further continuation of this line of argument would mean that the boundaries between individual transformations would become blurred and that they would merely be 'nice' theoretical ideas with little possibility of practical implementation.

In order to avoid this blurring of transformations it is necessary to think of a retained object as being first and foremost a (possibly disjoint) set of physical boxes for physical data objects, i.e. more a conceptual container than a real one in any sense. If this is true then the partial transformation LPSELECT will always associate a logical record in the container with some specific physical part(s) of the container. Since the required user functions are *Readfromfile*(LRECORD) and *Writetofile*(LRECORD) the logical object never needs to exist as a separately realisable physical representation of the data structure involved and remains, as its name implies, an abstract concept. Because of this a logical retained object can contain any number of records with some relation to another common data element (as described for a structured data element) and it may at the same time be spread among parts of any number of physically separate containers. This enables logical retained objects to be specified with overlapping contents without complication of the relevant selection transformations. The logical properties must therefore be used with the transformation LPSELECT. Similarly this transformation must form part of the specification of the contents in order for a user process to properly access the logical retained object.

Hertweck (1978) expresses the need just outlined in terms of some type of file dictionary and goes on to suggest that each record should similarly contain its own specification. This notion of self-defining contents is generalised in the partial transformations being described here; in particular a dynamic specification is considered. LPREALISE and PLINTERPRET which are part of the retained object specification rather than its contents are, of course, processes and also contain a representation translation from the physical medium 'code' to the logical 'code' used for manipulation—without requiring that each record in the contents contain its own dynamically variable specification.

A formal specification for a retained object must therefore contain the following parts which are an extension of the general ideas discussed for the universal file system:

(*a*) Identification—both as a name by which the retained object may be known to a user process and also the identity of the container itself.

(*b*) Specification of the logical object.

(*c*) Specification of the physical container.

(*d*) Specification of the physical representation.

(*e*) The dynamic state of the physical/logical object.

(*f*) Transformations—a set of transformation process identities (or, possibly alternatively, a 'shorthand' for some standard transformations). NB if the actual transformations were to be specified, as opposed to merely giving their identities, any implementation of dynamic transformation change would be more complex. Using this method, any number of different transformations can be dynamically assigned to the specified identities.

(*g*) Retained object contents.

With these ideas in mind a formal specification of a retained object may be given as a modified version of the specification given by Kugler *et al* (1978).

$$\text{FILE} = \text{IDE} \times \text{VALUE}$$

where

$$\begin{aligned} \text{IDE} &= \{id \mid id \text{ is a correct filename}\} \\ \text{VALUE} &= (\text{DESCRIPTOR} \times \text{DATA}) \backslash \text{RESTRICTIONS} \end{aligned}$$

where

$$\text{DESCRIPTOR} = \text{IDENTITY} \times \text{DESCRIPTION} \times \text{STATE} \times \text{ACCESS}$$

where

$$\text{IDENTITY} = \text{SITE} \times \text{OWNER} \times \text{TITLE} \times \text{GENERATION}$$

where

$$\begin{aligned} \text{SITE} &= \{id \mid id \text{ is a network node name}\} \\ \text{OWNER} &= \{id \mid id \text{ is a valid usercode}\} \\ \text{TITLE} &= \{id \mid id \text{ is a valid title}\} \\ \text{GENERATION} &= \{\mathbb{N}\} \end{aligned}$$

$$\text{DESCRIPTION} = \text{PHYSICAL} \times \text{LOGICAL}$$

where

PHYSICAL = {physical attributes of container}
LOGICAL = {logical attributes of container}

STATE = {state attributes}
ACCESS = {access specifications}

DATA = INFORMATION × TRANSFORMS

where

INFORMATION = ALPHABET
[[symbols defined in the logical retained object]]
TRANSFORMS = (SELECTIONS × SUBTRANSFORMS)
V STANDARDMAP

where

SELECTIONS = [*id* | *id* is the name of an LPSELECT transformation function]
SUBTRANSFORMS = PLTRANSFORMS

where

PLTRANSFORMS = [< *id*_1, *id*_2 > | (*id*_1 is the name of a LPREALISE transformation function), (*id*_2 is the name of a PLINTERPRET transformation function)]

STANDARDMAP = {STANDARDTRANSFORM, ORGKEYLENGTH, RECFORMAT}

where

STANDARDTRANSFORM = {implementation defined standard transforms}
ORGANISATION ∈ {none, random, serial, iseq, seq}
RECFORMAT = ORGANISATION × ORGKEYPOS × ORGKEYLENGTH
ORGKEYPOS ∈ ℕ
ORGKEYLENGTH ∈ ℕ
RESTRICTIONS = {*val* | *val* is a combination of attributes in which at least two are not compatible}

Underlying file systems

The range of retained object specifications used in the wide variety of operating systems—from the large and sophisticated to the small and primitive—which are to run under CLP control is potentially almost infinite in the range of properties considered together with the way that these are implemented. Fortunately when considering their file systems this range is not quite so wide as it may seem. It is even narrower when those systems are included in which for architectural reasons the file handling software is not formally part of the operating system but rather a separate set of programs, routines or both. Considering all retained object handling facilities in common, whether built-in or separate, is essential for this study, although considerations of specific details for specific systems only becomes necessary when the detailed design of specific mapping functions for a particular CLP is being worked out.

All file systems provide a mechanism for obtaining physical access to storage files, for creating them and for deleting them again in a physical sense (i.e. they contain at least physical description, contents and some form of identity as part of their retained object specifications). The majority of systems do not, however, consider input from or output to non-storage peripherals as being physical access to a retained object. Those which do allow stacks of cards, for example, to be considered as files only do so at the higher logical level. It is important to note that physical access to a retained object in even the most primitive of operating systems is provided in relation to some object identity provided by the user. This user-supplied identity may have to indicate not only the name but also the medium on which that named object is known by the user to be stored.

Excepting those operating systems which are no more than simple hardware supervisors and may therefore be deemed to fall outside the class of true operating systems being considered here, all operating systems, knowing where a retained object is physically stored, maintain at least *implicit* information about the possible logical organisation. To protect themselves they also maintain, as a minimum, information about whether the contents of a retained object is suitable to be executed as instructions by the machine, although in some systems this does not actually prevent the user from attempting to execute the contents of any object! The majority of systems do, however, maintain some other information regarding the logical organisation of the retained object contents, even if only certain restricted forms of mappings are provided, recognised or both (e.g. in many systems 'random' is often used to mean 'anything not specifically provided for'). The more user support functions which are provided by an operating system, the more logical mapping information has to be retained about the particular object—for example the provision of a unified file store (as seen by the TCL in the CLP universal file system) may require the provision of dumping, retrieving and archiving functions based upon date information about the retained object. Once this degree of sophistication is achieved it is usual (although not essential) for some form of central directory to be maintained for all retained objects within the unified file store. As was the case for the CLP universal file system, this can take the form of a single directory for all retained objects or some form of hierarchical system of directories.

Once some form of directory facility is provided it becomes possible for a form of access protection facility to be incorporated, based upon the owner of the retained object. In the simplest cases this access protection may provide for only two 'groups' of retained objects—those belonging to users and those belonging to the system—and then only by providing a general facility to prevent any form of access by any user to system objects. The variety of access protection schemes found in operating systems shows perhaps the widest discrepancies between one system and another of any file facility in use, most of them in no way approaching the security specified for the universal file system.

Retained object mappings

The outline of the universal file system and the above notes on possible underlying file systems tend to suggest that the retained object specifications used by them will be quite different, although in many cases they will be at least partially complementary. The command language processor must be able to map the underlying file system view into its own internal view and vice versa. This, however, is not the only set of mappings which it must provide. In particular, it should provide the TRANSFORMS of the retained object specification used by the operating system if possible, to avoid double transformations on reading or writing the contents of retained objects. Since the set of PHYSICAL and LOGICAL properties which are part of the DESCRIPTION and the set of properties which constitute the STATE may well differ between the universal file system and the underlying file system then a variety of mappings between these two sets of properties must also be provided in the CLP.

In order to enable the user and his programs to properly manipulate retained objects in its universal file system the CLP must also provide mappings for a number of using actions. These actions may either be performed directly by the user (in some cases) or by his programs (in all cases) dependent upon the facilities provided by his particular version of the TCL. The

mappings provided must apply at all times, irrespective of whether the user or his programs are attempting to manipulate elements of retained objects. If this rule is not observed then:

(*a*) A user could write programs (assuming that he were able and allowed to do so) to perform actions in relation to retained objects which would otherwise be considered to be illegal by the universal file system.

(*b*) The universal file system information about all retained objects would not reflect their true state as changed by the user program, say.

The variability of underlying file systems and the requirements for mapping the effects of user actions at all times in order to maintain the integrity of the universal file system implies that the universal file system retained object specification must be complete. This need, however, brings with it a problem! The TCL user, as already discussed, cannot manipulate directly in the TCL any physical attributes since these are machine dependent. Neither, of course, can he effect any physical transforms (i.e. LPSELECT, PLINTERPRET and PLREALISE). The question which therefore arises in considering the CLP, which must hold these parts of the physical specifications privately, is whether it should retain a 'universal' set of these properties or whether it should rely upon that set of physical properties used by the underlying file system (where available). In order to carry out any mapping of logical properties at all the three 'physical' sections of TRANSFORMS must obviously relate to the physical view of the retained object held by the underlying operating system. It is therefore logically consistent to design the universal file system specification maintained by the CLP to reflect the logical structure and properties as seen by the TCL user while maintaining the physical structure and properties as reflected in the underlying file system.

Practical mapping design considerations

Where the CLP is to map the universal file system on to an underlying file system which provides identical features (although, perhaps, in a slightly different form) then the mappings involved could be little more than some relatively straightforward renaming rules. At the other end of the scale, however, where only a few of the facilities provided by the universal file system are available in the underlying file system, then either

(*a*) a fully comprehensive set of system functions must be provided within the CLP to implement the complete universal file system specification and mapping needs; or

(*b*) the user, whatever his needs, is only provided with those functions contained in the underlying file system, mapped in a suitable way to his TCL's view of the universal file system, other facilities merely giving rise to a 'facility not available' type of error message; or

(*c*) some compromise mapping be provided between the two extremes just referred to.

The choice of which of these three implementation policies to adopt is not, however, open to an individual implementer in quite the simple way suggested by merely listing the three possibilities. The restriction on choice is contained in the philosophy behind the TCL itself—that its semantic effects be the same everywhere! While this may seem to imply that the fully comprehensive CLP implementation is an absolute requirement, practical considerations regarding very small computers limit the size of any extra facilities which could be provided without unduly overloading machine storage, particularly if no virtual memory facility is available. This leads to the conclusion that some compromise must be reached along the lines of the third option suggested.

Even at this stage, the selection of the compromise is again not entirely within the implementer's discretion since he must choose a compatible compromise. This need for compatibility arises, of course, from the need to provide identical semantic effects. In other words if a user (or his programs, of course) can do anything at all with retained objects then the resultant effect must appear identical to the effects achieved when using the same TCL on any other machine.

The discussion so far has thus led to a general notion that every implementation of the universal file system must contain three things:

(*a*) the users' view of their retained objects as described by the TCL—i.e. a universal file system fully comprehensive retained object specification as described above, together with some form of directory and supporting internal routines;

(*b*) a set of mappings for those facilities provided which is compatible with (*a*);

(*c*) this possibly empty section which consists of: (1) a set of auxiliary functions to ensure complete compatibility for those functions which are provided; (2) a set of 'not available' messages with a message production facility for those universal file system functions not provided.

Retained object manipulation facilities

The discussion so far has concentrated on the nature of a retained object and the relationship between the command language processor view and that of the underlying file system, i.e. the process/operating system boundary. Note has only been taken of the user/CLP boundary (i.e. the TCL) insofar as it affected the needs of the universal file system. Similarly, only passing mention has been made of the boundary between other processes running on behalf of users and the operating system.

The need for the universal file system to keep a complete and correct view of a user's retained objects at all times implies that all actions in relation to these objects must be carried out either with the active participation of the CLP or with the operating system informing the CLP as they are done. Whichever of these methods is involved depends upon whether CLP 'permission' checks are required befor the action can be done.

The manipulation facilities required by the user and his programs may be considered in four groups:

(*a*) IDENTITY. This group of manipulations contains only two functions, create and identify a retained object or find an identified retained object. In the general sense both of these functions are always available in all 'file' systems. The only differences between one system and another are: (1) The ability to uniquely identify a retained object within a complete set of system storage media. The resolution of any difficulty of this sort must therefore rely on the universal file system directory containing a unique map. In order to create retained objects in the worst case it will be necessary for the CLP to either provide or have access to information about unallocated storage; (2) The ability to provide for the creation of retained objects of all possible sizes and shapes specifiable in the universal file system. Any inabilities of this nature may be mapped into suitable 'not available' messages or into CLP additions dependent upon the practical needs of users of that machine and any restrictions of size which may be imposed.

(*b*) ACCESS. This group of manipulations consists essentially of five functions which are considered vital to the integrity of any CLP and for which, therefore, facilities must be provided irrespective of their availability in the underlying file system. These functions are: (1) Keep the retained object as a permanent object belonging (in TCL terms) to the owner; (2) Delete the retained object—for a non-permanent object (i.e. one which has not been kept) this action never involves any security con-

siderations except for ensuring that only the retained object which it is desired should be deleted is in fact deleted. For a permanent retained object the facility must be provided to ensure that the person wishing to delete the object has been given permission to do so by the owner. This, of course, is a hidden sixth function not directly accessible to the user, either using TCL or in a program; (3) Allow other users to have certain specified access facilities to a retained object. This function is restricted to the owner of the object and represents the de jure capability described by Bishop and Snyder (1979) and is, of course, only applicable to a permanent object; (4) Remove one or more specified access facilities to the retained object from another user—again only applicable to a permanent object; (5) Show to the owner (and only to him/her) the current access rights to a retained object which he owns.

(*c*) DESCRIPTION. The functions in this group of properties are more in the nature of language facilities than conventional ideas of function since they involve altering the value of some component of the DESCRIPTION by assignment to it (if logically possible) and 'reading' the current value. These two basic actions in respect of each individual component may necessitate the provision of any or all of many to one, one to one, one to many and many to many mappings dependent upon the meanings attached to individual LOGICAL properties maintained by the underlying file system. It may be equally impossible to provide unambiguous mappings—or any mapping at all—if the CLP retained object specification attributes cannot be mapped on to underlying file system properties. It will then generally be practically sensible to provide 'not available' messages appropriate to the omitted facilities.

(*d*) DATA. The manipulation of retained object contents involves the actions known conventionally as opening, seeking, reading, writing and closing. These are subsumed in the functions *Readfromfile* and *Writetofile* described in deriving the specification of a retained object. The exact facilities provided for these functions are often dependent both on language system and the particular machine. They are of no direct concern to the CLP, but every action which a program makes with regard to a retained object is of concern to the universal file system either from the need to ensure security, from the need to ensure legality, from the need to ensure consistency, from any two of the three or, indeed, from all three reasons. When necessary the CLP must be able to prevent illegal or insecure operations of any sort![NB some types of retained object (e.g. a peripheral) have no DATA component and these manipulations are therefore not applicable to them.]

Implications of network connection

The design requirements for a universal file system and transportable control language CLP must consider the possible implications of the connection of the computer system concerned into a network with other computer systems. Since each node in a network will have its own CLP, no problems arise at the user level involved in any retained object connection between nodes. All difficulties which may arise concern the CLP and the operating system together with the communications software involved. Assuming that the communications software is able to convert user node identities to network addresses and pass messages transparently between nodes then the only problems lie in the CLP implementation of the manipulations which it controls internally for a remote user.

The facility provided by joining machines using a standard CLP in a network raises no problems concerning IDENTITY, ACCESS and DESCRIPTION which cannot readily be solved by the use of a standard message system [see for example Liskov (1979] between nodes. Manipulations involving access to the retained object contents does, however, raise some transfer problems when the machines joined in the network are of different types or, indeed, merely have different underlying file systems. The problems involve the need for a mechanism within the CLP to convert to and from some standard communicable code representation of data not only the contents of the retained object, but also the meaning of that code representation—the TRANSFORMS of the specification.

Fortunately, this need to include TRANSFORMS requires only that some standard code representation of this is available as a valid internode language form for transfer between CLPs. The conversion of ALPHABET to and from some standard communication code to pass it as messages can also be provided by relatively simple additions to the set of partial transformations described when developing the specification of a retained object.

Consider the function *Readfromfile* which is achieved by successive applications of partial transformations in the order:

(1) LPSELECT
(2) PLINTERPRET

PLINTERPRET involves representation translation which necessarily takes place at the 'sending' network node (at which the physical container exists).

Similarly the function *Writetofile* is achieved by successive application of the partial transformations in the order:

(1) LPSELECT
(2) LPREALISE

In this case, both the partial transformations are necessarily carried out at the 'receiving' node.

Since different nodes in a heterogeneous network are very likely to employ different representations it is necessary to provide some standard representation in which internode transfer of primitive data elements may be effected. The conversions from a node representation to this standard representation and vice versa may, of course, be made independent of the retained object or process concerned. The two functions *Readfromfile* and *Writetofile* may then be achieved by the following sequence of partial transformations:

(*a*) *Readfromfile*
(1) LPSELECT
(2) PLINTERPRET
(A) NODETONETWORKCODE
(B) NETWORKTONODECODE

(*b*) *Writetofile*
(A) NODETONETWORKCODE
(B) NETWORKTONODECODE
(1) LPSELECT
(2) LPREALISE

The 'universal' translation transformations NODETONETWORKCODE and NETWORKTONODECODE are common to all retained object data transfers between nodes and therefore need not form part of the specification of individual objects. The need to be able to specify all the other transformations in a standard manner may, of course, be achieved either using TCL messages or some standard low level code common to all nodes of a network.

The introduction of a network connection to a computer system therefore only adds to the CLP design the need for facilities for converting all elements of a retained object specification into a standard message form and vice versa. The necessary representation code translations seem to be

more suitable for inclusion in the necessary network communications software processes.

Summary

The requirements for the retention of certain types of object between computing sessions involves their consideration as file-like objects which may have a null DATA component. This enables all object types to be considered in a common way in order to assess the impact of their retention on the command language processor for a transportable control language. The need to provide the user of a TCL with identical semantic effects whatever his real underlying processor and operating system led to the introduction of the need for a universal file system.

The nature of a retained object as a container of structured data elements was introduced in detail to enable a standard retained object specification as seen by the universal file system of the control language processor to be developed. Consideration of the wide range of possible views of retained objects which could be held by a variety of operating systems led to the need for mapping of both individual retained object properties and of the retained object semantic access functions within the CLP.

The need for the user's view of his universal file system to be always consistent with the 'real' state of the retained objects as maintained by the operating system led to the requirement that the CLP either take part in or be informed of all actions relating to retained objects, whether initiated directly by the user or indirectly by a process activated on his behalf. A review of the manipulations required enabled the problems of this integrity maintenance for the CLP to be introduced.

Finally, the CLP design requirements were reviewed in the light of the possible connection of the computer system as a node in some network. Apart from the introduction of a message system, this showed that, providing that all semantic transformations were expressible in a universal way, then the transfer of any components of a retained object could be effected by the addition to the CLP of suitable message generating features.

Acknowledgements

The ideas presented in this paper are the result of considerable discussion with friends and colleagues, in particular I should like to thank N. S. James, D. Jardine, P. C. Jenkins, H. J. Kugler and C. Unger.

Appendix—Virtual objects

A data base which permits the use of many different views of one set of logical data is only a particular manifestation of the general concept of a virtual object as a subset of some other logical object. Considerations of storage and accessing efficiency often dictate that these subsets should be derived from their logical counterparts rather than provided with separate physical storage in many practical implementations of any virtual object mechanism. The extension of the CLP retained object specification given below is a consistent mechanism to meet these practical constraints.

In order to implement virtual objects in this way it is necessary to introduce three additional transformations which correspond in a general way to the three already introduced

VLEXTRACT corresponding to LPREALISE
LVINSERT corresponding to PLINTERPRET
VLSELECT corresponding to LPSELECT

where the additional transformations may be described as:

(*a*) VLEXTRACT. Simulate the shape of the virtual record given the corresponding logical record.

(*b*) LVINSERT. Simulate the shape of the logical record given the corresponding virtual record.

(*c*) VLSELECT. Select the logical record in the virtual organisation from the corresponding logical organisation.

With these additional transformations a specification of a retained object for a CLP implementation requires the following modifications:

SUBTRANSFORMS = LVTRANSFORMS × PLTRANSFORMS

where

LVTRANSFORMS = [⟨*id*_1, *id*_2⟩ | (*id*_1 is the name of the VLEXTRACT transformation function), (*id*_2 is the name of the LVINSERT transformation function)]

SELECTIONS = ⟨*id*_1, *id*_2⟩ | (*id*_1 is the name of VLSELECT transformation functions), (*id*_2 is the name of LPSELECT transformation functions)]

With these implementation additions to the CLP retained object specification, the sequences of actions to provide the *Readfromfile* and *Writetofile* functions in a network environment must be extended to become:

(*a*) *Readfromfile*
(1) VLSELECT
(2) LPSELECT
(3) PLINTERPRET
(A) NODETONETWORKCODE
(B) NETWORKTONODECODE
(4) VLEXTRACT

(*b*) *Writetofile*
(1) VLSELECT
(2) LVINSERT
(A) NODETONETWORKCODE
(B) NETWORKTONODECODE
(3) LPSELECT
(4) LPREALISE

References

BISHOP, M. and SNYDER, L. (1979). The transformation of information and authority in a protection system, in *Proceedings of the 7th Symposium on Operating Systems Principles*, pp. 45–54. ACM, New York.

HERTWECK, F. (1978). An approach to a unified view of file handling, personal communication.

KUGLER, H. J. *et al* (1978). Project NICOLA—Progress report 2, August, University of Dortmund.

LANGEFORS, B. and SUNDGREN, B. (1975). *Information Systems Architecture*. Petrocelli, Princeton.

LISKOV, B. (1979). Primitives for distributed computing, *Proceedings of the 7th Symposium on Operating Systems Principles*, pp. 33–42. ACM, New York.

MILNE, R. and STRACHEY, C. (1976). *A Theory of Programming Language Semantics*. Chapman & Hall, London.