Procedural Encapsulation: A Linguistic Protection Technique

Stephen N. Zilles
Cambridge Systems Group
IBM Systems Development Division
Cambridge, Massachusetts

Introduction

System reliability is an important aspect of operating system construction. Because of this, a number of researchers have proposed design methodologies [e.g., 1, 2, 3] which are intended to produce more reliable software. Although the methodologies differ, there seem to be some important properties that they have in common.

One property is the decomposition of the system into components or modules which make as few assumptions as possible about the other components. We will call this the "minimal assumptions" property. Minimizing the assumptions one component makes about another component means that subsequent maintenance and extensions can be done with a smaller probability of introducing errors. A programmer is less likely to overlook critical, obscure interconnections if such interconnections are severely restricted.

Another important property of the methodologies, although somewhat less universal, is that the correctness of each module should be demonstrated. That is, it should be possible to prove, or at least convince someone else, that the component works as intended.

The development of the paper is based on the assumption that the best way to support these methodologies is to provide a system building language, called the OS language, which is closely integrated with the operating system features. This concept is not new. For example, Multics [4] was implemented in PL/I, and the operating system runs in and provides a PL/I environment in which most operating system facilities are available as PL/I callable functions.

The main issue addressed in this paper is this: What set of language features are helpful in obtaining the properties desired of the methodology, and how should these features be integrated with the operating system facilities? The thesis is that procedures, particularly procedures which can return procedures as their result, are the proper mechanism for modularizing both programs and data. Furthermore, the natural access control a procedure provides should be enforced by the protection mechanism of the operating system; thus, providing the user with a natural, easy-to-use presentation of the protection facilities. We will use the term procedural encapsulation to refer to the technique of representing system components in terms of one or more procedures such that interactions among compoments are limited to procedure calls.

An Example

Using procedures to modularize programs is well known and will not be discussed here. The example will illustrate how procedures can be used to represent another class of system components, data objects, which are not normally expressed as programs. We have chosen to use the stream objects of OS6 as our example because they are fairly typical of the data objects that one finds in an operating system. OS6 is an experimental operating system developed by the Programming Research Group at Oxford [5, 6]. The stream objects, which are used for all input and output of information on OS6, are intuitively a sequentially ordered collection of data units. They are defined, however, as any object to which a certain given set of operations is applicable. The operations and an informal definition of each in terms of an internal current position indicator are:

NEXT	returns the data unit designated by the internal position indicator and moves the position indicator to the sequentially next data unit.
OUT	puts a data unit into the sequentially next place in the stream and moves the internal position indicator.

PUTBACK takes the data unit given as an operand, changes the internal state so that that data unit appears to immediately precede the data unit currently indicated by the position indicator, and then moves the position indicator to the newly added data unit.

ENDOF returns true if the end of the stream has been reached, and false otherwise.

There are also several housekeeping operations whose exact function is not relevant to the example.

A stream is uniformly represented by a vector of entry points, one for each of the above operations. And, because the OS6 system language does not provide true function returning functions, the state information needed by these entries (a position indicator, a pointer to the stream data and the information necessary to represent data which has been put back) is also stored in the vector.

Within a program, a use of a stream operation is translated into code to choose the appropriate entry point from the stream object vector and to call that entry point with the arguments given to the stream operation. A pointer to the vector is included among the arguments so the state information will be accessible.

A new type of stream object can be defined by choosing a representation for the data in the stream and the state information, and then writing procedures to implement each of the stream operations on that representation. A particular stream object of that type can then be created by obtaining sufficient space for the representing vector, putting the entry point of each of the new procedures in the appropriate vector element, and initializing the state information used by those procedures.

Encapsulation

The representation of each stream, and in fact the way a programmer would think about a stream, is in terms of the set of procedures which implement the operations defined on streams. The representation of the stream state and contents, although accessible, need not be manipulated directly by the programmer. When only the defined operations are used, a program is independent of the particular representation chosen for a stream. The term "procedural encapsulation" is derived from the fact that the procedures encapsulate or isolate the representational aspects of the data.

Support for the Methodologies

Using the technique of encapsulation, a data type is completely characterized by the set of operations defined on the data of that type and the observable relationships between those operations. The available operations can be divided into two classes. There are observation operations, such as ENDOF, which make portions of the content or state of the data object visible, and there are modification operations, such as OUT and PUTBACK, which change the content or structure of the data object. It is sometimes the case, however, that a single operation, such as NEXT, combines a modification (to the position indicator) with an observation.

This view of data can be formalized by formalizing the relationships between the characterizing operations. First, it is necessary to formally specify, in terms of this or other defined data objects, what will result from the application of each observation operation. Then the modification operations can be formally defined in terms of their effects upon subsequent applications of the observation operations*. This formalization makes it possible to define and prove the correctness of a particular representation of the data object. Thus, procedural encapsulation satisfies the "demonstration of correctness" property.

Procedural encapsulation also supports the "minimal assumptions" property. When each module is a set of related procedures, the assumptions about one module which can be exploited by another module are limited by the ways in which each procedure can access other procedures. In most higher level languages, the semantics of the language limit access between procedures to calls on the specified entry points of the procedures. If the use of block structured or nested procedures is allowed only within a module, then data declared local to one module is inaccessible to all other modules. Therefore, the only assumptions a module can exploit are assumptions about the names of entry points in other modules and the arguments each entry point requires.

Note that in discussing access restrictions we are referring only to the types of access actually defined in the language and not to the types of access allowed by an implementation.

*Operations which combine modification with observations yield multiple results. Following established mathematical practice, such functions can be represented by a set of functions, one for each component of the range. Each component function can then be defined as a simple observation or modification function as the case may be.

In many cases, the operating system on which the code for the procedures is run allows unlimited access or only weakly controlled access to procedures and their local data. An approach to operating system support for limiting the accessibility of procedures is discussed below.

Language Features for Procedural Encapsulation

One problem with the OS6 representation of a procedurally defined data object is that the vector which represents the system object is accessible to the user as a vector. There is no protection for the state information, and the data representation is exposed to the user's view. A solution to this problem is to encapsulate that information in the procedures which represent the data object. If such procedural objects are to be constructable while the program is in execution, then we are limited to approaches which allow procedures to be returned as the result of a procedure.

Because information must be preserved between invocations of the procedure or procedures which encapsulate the information, it cannot be represented by ALGOL-like local variables which are reallocated on every call. This problem may be solved in one of two ways depending on whether or not the OS language is block structured. In both approaches to the problem, we will assume that new instances of the data type being represented are returned as the result of invoking a procedure which constructs a procedural representation of the data.

Block Structured Languages

Suppose the OS language is block structured and implements retention of variables. Then, the definitions of the procedures representing the operations on a given data type can be nested within the procedure or block that is the constructor for those data objects. The variables representing the information that is required by the operation procedures can be declared local to the constructor procedure or block and referenced from each operation procedure in terms of free variables which are resolved to the constructor. Retention implies that the storage for those shared variables is not released when the constructor returns, but is held as long as the user has access to any of the operation procedures which reference the variables.

The body of the constructor consists of the code to initialize the variables shared by the operation procedures, and the code to return the representation for the data object being constructed. Initialization is based on the arguments passed to the constructor.

The data object can be represented procedurally in at least two ways. The method which is closest to the OS6 approach is to return a vector of entry points for the procedures, one per operation, which are defined in the constructor. Note that a new generation of local variables is created each time the constructor function is entered. Therefore, the content and state of each data object is completely independent of any other. However, for economy, they would probably all share the text of the operator defining functions.

Returning a vector of functions still exposes a piece of the representation, namely, the vector. Therefore, another representation for the result of the constructor is a single procedure which takes a variable number of arguments. The first argument is a coded value which identifies which operation is to be performed, while the remaining arguments are the operands of that operation, omitting, of course, the data object itself. With this approach, each use of an operator defined on that data object is translated into a call on the procedure representing the data object with the appropriate code for that particular operation.

The extra coded argument could be eliminated if the OS language allowed a procedure to have multiple entry points. In that case, the stream operators would be translated into calls to the appropriate entry point in the stream object. This latter approach is similar to the handling of classes in SIMULA 67[7]. Unlike SIMULA, however, the local variables and some of the nested entry points are not made accessible outside the procedure in which they are defined.

Non Block Structured Languages

If the OS language is not block structured, then another technique can be used. Most major languages have a class of storage, called *own* in ALGOL, which is local to the procedure but is retained between calls on the procedure. If the language provides a way for generating a copy of a procedure with new instances of all the own variables, this could be used in place of the local variables of an enclosing block. Since the own variables are strictly local to the procedure in which they are defined, the only possible procedural representation is one with multiple entry points in each procedure or one where an entry code is used to distinguish which operation is to be performed. Initialization can be handled by providing a special entry code or entry point which is used by the constructor function before returning the new procedure.

System Considerations

Although the formal semantics of a procedure imply that the procedure body is protected, most implementations of procedures do not provide protection at all. It is usually possible to read the text, to enter the procedure at any point, etc. If access to procedures is to be limited to the types of access defined in the language then the system must provide some elementary protection features.

It appears to be necessary for the system to treat procedures as a primitive object which can be separately protected. It suffices to be able to partition storage into regions, each of which is individually protected. Then each procedure could be represented by a region containing a list, called a capability list [8], of regions together with the type of access that is allowed to that procedure. One of the regions, for example the first, would hold the code for the procedure.

Only two types of access restrictions are required. A region may either be "freely accessible" or it may be "enter only." By freely accessible, we mean that any operation that is defined on the contents of the region by the system may be performed. In most current systems, freely accessible would correspond to having read, write, and execute access. Each procedure sees other procedures as "enter only" regions.

Two classes of access are sufficient because a program can only access procedurally encapsulated data by way of the operations for which it has entry points. By controlling which entry points or entry codes are made available when an object is constructed, the access to that object is controlled. This implies that the representation of a data object may admit or contain only a subset of the entry points for operations defined on that data type. For example, a "read only" stream object might not have an entry defined for OUT.

The system must also supply enter and return operations. The enter operation provides the only way to call another procedure and to pass accessibility to the regions that hold the arguments. Upon the execution of an enter, the system would suspend the current procedure and its capability list, and would begin execution at the appropriate entry in the region holding the code for the procedure being entered. The newly entered procedure would be given access only to the regions in its capability list plus those passed to it by the calling procedure. The state and capability list of the calling procedure is saved on a hidden stack to be reactivated when the newly entered procedure returns. The return operation can optionally return a capability list to be added to the capability list of the caller when it is reactivated.

These facilities make it possible to treat each procedure or set of procedures defining a single component as a separate domain [8]. In fact, procedural encapsulation is a good way to use hardware which implements domains. Because each system component is individually protected, the support for domains should be designed to handle a number of relatively small domains.

Some systems do not implement domains but, instead, group capabilities with processes. In these systems, it is necessary to switch processes to change the active capability list. It would appear that placing each component (i.e., the procedures representing the component) in a separate process involves more overhead than using separate domains because each procedure call would require a process switch. Hence, on such systems, practical use of procedural encapsulation may be restricted to those components whose execution time would normally hide the increased overhead.

Conclusions

Procedural encapsulation is a useful technique for building modular, reliable operating systems. With appropriate support from the operating system, it can provide protection for data objects. Both accidental destruction or modification, and intentional subversion are prevented. This form of protection is easy to use correctly since it is integrated with a natural building block for system components: procedures.

The set of assumptions that can be made about a module is minimized. Because representational details are encapsulated within procedures, the representation cannot be manipulated nor its form discovered. Therefore, the representation can be modified to fix bugs or to enhance performance without introducing problems caused by invalidating assumptions made in other modules.

The correctness of a particular representation is easier to establish because the set of applicable operations is strictly limited. The inaccessibility of the local variables also makes correctness demonstrations simpler. A purely local argument suffices to establish correctness. Finally, the procedural approach gives the OS language a powerful form of extensibility and provides a framework for structured programming. The author believes facilities for procedural encapsulation should be part of every OS language and its supporting operating system.

Acknowledgments

The work described in this paper was prompted by the work of J. H. Morris [9]. The helpful suggestions made by Barbara Liskov, J. E. Stoy, and Leo Rotenberg are gratefully acknowledged.

References

[1] Dijkstra, E. W., "Notes on Structured Programming" in Structured Programming (C.A.R. Hoare, ed), Academic Press, New York-London, 1972

[2] Parnas, D. L., "Information Distribution Aspects of Design Methodology," IFIP Congress Preprints, (1971)

[3] Liskov, B. H., "A Design Methodology for Reliable Software Systems," FJCC Proceedings, Vol. 41 (1972), 191-199

[4] Corbato, F. J., Saltzer, J. H., and Clingen, C. T., "Multics-The First Seven Years," SJCC Proceedings, Vol. 40 (1972), 571-583

[5] Stoy, J. E., and Strachey, C., "OS6-An Experimental Operating System for a Small Computer. Part 1: General Principles and Structure," Computer J. Vol. 15, No. 2 (1972), 117-124

[6] Stoy, J. E., and Strachey, C., "OS6-An Experimental Operating System for a Small Computer. Part 2: Input/Output and Filing System," Computer J. Vol. 15, No. 3 (1972), 195-203

[7] Dahl, O. J., Myhrhaug, B., and Nygaard, K., the SIMULA 67 Common Base Language, Norwegian Computing Center, Oslo, Publication No. S-22, 1970

[8] Lampson, B. W., "Dynamic Protection Structures," FJCC Proceedings, Vol. 35 (1969)

[9] Morris, J. H., Jr., "Protection in Programming Languages," Comm. ACM, Vol. 16, No. 1 (January 1973), 15-21