testing," Univ. California, San Diego, Computer Science Tech. Rep. 16, 1977.
[23] P. Henderson and R. Snowden, "An experiment in structured programming," *BIT*, vol. 12, pp. 38–53, 1972.
[24] W. E. Howden, "Algebraic program testing," Univ. California, San Diego, Computer Science Tech. Rep. 14, 1976.
[25] M. Geller, "Test data as an aid in proving program correctness," in *Proc. 2nd Symp. Principles of Programming Languages* (1976), pp. 209–218.
[26] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156–173, 1975.
[27] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554–559, 1975.
[28] R. M. Burstall, "Proving correctness as hand simulation with a little induction," in *Proc. Int. Federation of Information Processing Societies 74*. Amsterdam, The Netherlands: North Holland, 1974, pp. 308–312.
[29] L. P. Deutsch, "An interactive program verifier," Ph.D. dissertation, Univ. California, Berkeley, 1973.
[30] W. E. Howden, "Lindenmayer grammars and symbolic testing," *Inform. Process. Lett.*, to be published.
[31] W. Hetzel, "An experimental analysis of program verification methods," Ph.D. dissertation, Univ. North Carolina, 1976.

William E. Howden, for a photograph and biography, see p. 73 of the January 1978 issue of this TRANSACTIONS.

Dynamic Restructuring in an Experimental Operating System

HANNES GOULLON, RAINER ISLE, AND KLAUS-PETER LÖHR

Abstract—**A well-structured system can easily be understood and modified. Moreover, it may lend itself even to dynamic modification: under special conditions, the possibility of changing system parts while the system is running can be provided at little additional cost. Our approach to the design of dynamically modifiable systems is based on the principle of data abstraction applied to types and modules. It allows for dynamic replacement or restructuring of a module's implementation if this does not affect its specification (or if it leads to some kind of compatible specification). The fundamental principles of such "replugging" are exhibited, and the implementation of a replugging facility for an experimental operating system on a PDP-11/40E is described.**

Index Terms—**Data abstraction, domain architecture, dynamic address spaces, dynamic modification, modules, replugging.**

I. INTRODUCTION

ONE of the fundamental facts in software engineering is that large systems, once put into operation, are usually subject to frequent modification. A system is modified for several possible reasons, e.g., error correction, efficiency improvement, changes in or amplification of functional capabilities. Thus, to produce high-quality software, such modifications must be anticipated. The notion of *modifiability* comprises structural properties of software that facilitate fast and secure modification.

While requirements for software modifiability have been studied to some extent (cf. Parnas [11], Bauer [1]), the problem of how to modify a long-running system "smoothly," i.e., without stopping it and replacing it by a new version, has attracted only little attention. Shaw has mentioned the problem from the viewpoint of data abstraction [15]. Fabry was motivated to attack the problem by considering data base systems; he uses the term "changing modules on the fly" [2].

Throughout this paper, program modification during runtime will be called *dynamic modification* (as opposed to "static modification") in order to stress the analogies to similar uses of "dynamic" versus "static," e.g., static/dynamic memory management, linking. We were led to study, in a real system, dynamic modification and the design of a dynamic modification facility by a project to construct an experimental operating system. The system is to be used in operating systems research and education at a university. It is called a DAS (dynamically alterable system), and is being implemented on a PDP-11/40E. The system is designed in such a way that dynamic modifications are possible in all

Manuscript received November 15, 1977.
The authors are with the Technische Universität Berlin, Berlin, Germany.

0098-5589/78/0700-0298$00.75

system components except in the "kernel" which contains part of the modification facility.

In our design effort, we have consistently relied on the principles of data abstraction. The next section will clarify our use of the notions "type" and "module." The basic idea is (cf. Shaw [15]): since consequent data abstraction makes it possible to replace one implementation of a data type by another, without affecting the users of that type, why not do this at runtime—why not even convert the representation of already existing data objects?

We have confined ourselves to such dynamic modifications which we call *replugging*: a different implementation is plugged into the invariant specification "socket" of a type or module.

Fabry started off in a similar way. He describes the design for a system that allows for dynamic replacement of "modules," i.e., code segments that are used to manipulate data objects of a certain "type" (in fact, the notions of type/module coincide in his approach). The data representation within these objects is changed by appropriate conversion routines in order to suit a new code version; this is done on a demand basis, i.e., only when the object is actually being used by a process (and if the code has been replaced since the last use).

Data conversion is not always necessary. There are special cases where different code implementations operate on identical data representations. One can also think of the concept of a "current version" that determines the implementation only of a newly created object; replugging the current version would never affect existing objects, but would nevertheless have to be considered as a dynamic system modification. For these reasons we introduce the notion *dynamic restructuring*, which means dynamic modification that necessarily implies data conversion, either on demand or on request.

We will concentrate almost exclusively on replugging mechanisms for dynamic restructuring. The essentials of our design as opposed to Fabry's can be summarized as follows.

1) Types and modules are very different; each is replugged in its own way. A special "replug module" supplies the different replug functions.

2) Data restructuring is done on request, not on demand.

3) Data restructuring does not require conversion routines that take into account the representational details both of the old and of the new implementation. Restructuring is accomplished by "pairing" special information-transfer procedures of the old and new versions. The functional behavior of these procedures is described by the common specification of all versions.

4) The replugging facility is not built onto an existing, capability-oriented operating system. Rather, indirect access to segments is a built-in feature of the system's kernel, and the necessary synchronization (reader/writer exclusion) is achieved by means of special lock/unlock primitives.

5) Thus the replug module is functionally independent of all other system modules (apart from the kernel). Virtually all system components can be subject to replugging, even the replug module itself.

There are several issues concerning repluggable systems that will *not* be considered in this paper, yet cannot be ignored if such systems are to be developed to practical success. Among these are the following.

1) An alternative version of a system component must be separately compilable. The compiled segment has to be provided with links (e.g., capabilities) that connect it to the proper environment.

2) Installation of a not yet validated version is likely to produce disaster. Who would then rely on a correctness proof? If it is not possible to certify the new version by using another installation of the system or a special test environment, the newly installed version should be backed up by the old version for some time.

Our treatment of the subject is organized as follows: in the next two sections, we will introduce some terminology and explain the basic principles of replugging in a programming language framework. Section IV contains an overview of how address spaces are managed in the DAS system. Address space management forms the technical basis of our replugging facility, which is presented in Section V. This is followed by a general discussion of structural issues in Sections VI and VII. Efficiency considerations are included.

II. The Linguistic View of Replugging

The concept of data abstraction is only reluctantly supported by contemporary programming languages. The main principle, separation of abstract *specification* and concrete *implementation* is frequently stressed yet never consequently followed; the only exception we know of is MESA (Geschke *et al.* [3]).

Algol 68, although somewhat dated now, contains a remarkable feature that can serve as a guideline: A "procedure mode," as opposed to other modes, merely represents a *syntactic specification* for procedure objects ("routines"). The mode is by no means bound to one particular implementation, as is the case with other modes.

Suppose the mode of a procedure is augmented with a suitable formal description of its functional behavior, i.e., with a *semantic specification*: Such a mode would perfectly realize the idea of procedural abstraction within the framework of a programming language.

Let us play this game with data abstraction. The abstract specification of a data type is called a *mode*.[1] We will give a syntax for the denotation of modes which will allow the description of both syntactic and semantic specifications; the latter will always be given informally throughout this paper. Here is an example which gives an impression of a possible syntax:

```
MODE queue IS (size: nat)
              ⟨enter (item: string);
               remove: string;
               nonempty: boolean⟩
              {fifo}
          SI
```

[1] Since formal specification techniques and their theory are not the subject of this paper (cf., e.g., Guttag *et al.* [5] and Goguen *et al.* [4]), we do not answer the question of what an abstract data type really is.

This is a parametrized mode. Parameters are enclosed in parentheses. The "operational" specification is enclosed in angle brackets; it consists of a list of operation names for queues and the syntactic specifications for these operations. The semantic specification is enclosed in braces. Note that the operations 'enter'/'remove' are parametrized only with the objects to be entered/removed, not with the queue object.

An implemented version of a mode is called a *type*. In fact, our type is more like a SIMULA *class* than, e.g., the Pascal *type*. Here is an implementation of the mode 'queue':

```
TYPE  circular queue = queue
BEGIN OBJECT cell: array (1, size) OF string;
      VAR    head, tail, count: int;

      PROC enter =
      BEGIN cell [tail] := item;
            tail := tail MOD size + 1; count:+1
      END   enter;

      PROC remove =
      BEGIN remove := cell[head];
            head := head MOD size +1; count:-1
      END remove;

      PROC nonempty = count ≠ 0;

      head := 1; tail := 1
END   circular queue
```

The type consists of a mode, declarations of the component objects of the objects of this type (*representation*), and procedures that operate on the component objects. By reason of their occurrence in the mode, some of these procedures are exported as operations of this type.[2]

An *object* of a certain type can be introduced through a declaration, e.g.,

```
OBJECT seesaw: circular queue (2)
```

In this framework, each object is of some invariant type and of a corresponding mode. An object can be manipulated only through the operations defined in its mode. Notational example:

```
seesaw.enter (eof)
```

Thus, to use an object, the programmer is not required any knowledge of its type, nor shall he rely on this type in any way.

An object possesses a *value* at any time. The value cannot be "seen" by the user of the object. For the user, the value of an object is determined by the object's future behavior. Of course, this value has a concrete representation, the structure of which is determined by the object's type.

Since it is possible to define different types of the same mode, objects of different types yet of identical modes may coexist. So, if some value is to be "copied" from one object to another, the question arises: how can this be implemented? —since this process will by no means be a simple copying.

We may also ask: why does an object have to be bound to a specific type, although the user does not care about the type?

If we allow an object to change its type during its lifetime, we must provide an algorithm to accomplish the necessary restructuring of representations. Furthermore, if the object is to be used and restructured concurrently, we will have to provide an exclusion mechanism.

The operation of modifying an object in the way just described will be called *object replug*. We will denote it by using an arrow '→', as in the following example:

```
TYPE linear queue = queue
BEGIN "type body" END linear queue;
OBJECT buffer: circular queue (8);
  .
  .
  .
buffer → linear queue (8)
```

Until now, we have assumed that a type does not manage any nonlocal data. If there is nonlocal data to be managed and/or if types of different modes are conveniently supported by common mechanisms, the *module* concept serves as a means of encapsulating data, procedures, and types. In fact, there seems to be no precise and generally accepted definition of "module." We will clarify our use of the notion "module" by means of an example.

Elementary process management in an operating system can be conveniently formulated as a module which provides, e.g., a type 'process,' a type 'semaphore,' and a procedure 'current process.' Here is a simplified version:

```
MODULE process management =
                ⟨process ⟨procid: nat⟩;
                 current process: process;
                 semaphore (initial: nat)
                          ⟨wait; signal⟩⟩
                {multiprogramming
                 with fifo scheduling}

BEGIN OBJECT ready list: list OF process;
      VAR    last id: nat;

      TYPE   process =
      BEGIN  OBJECT id: nat := last id :+1;
             PROC   procid = id;
             ready list.enter (ME)
      END    process;

      PROC   current process = ready list.first;

      TYPE   semaphore =
      BEGIN  OBJECT waiting list:list OF process;
             VAR    counter: int := initial;

             PROC   wait =
             BEGIN  counter :-1;
                    IF counter < 0
                    THEN p: ready list.remove;
                         waiting list.enter (p);
```

[2]The "definition" in MESA (Geschke *et al.* [3]) is similar to our "mode," the "program" corresponds to our "type" (or "module" as introduced below). In Parnas *et al.* [12], our "mode" is called "spec-type," and our "type" is called "mode"!

```
                    current process.resume
                FI
            END wait;
            PROC signal =
            BEGIN counter:+1;
                IF counter ⩽ 0
                THEN p: waiting list.remove;
                    ready list.enter (p)
                FI
            END signal
        END semaphore
    END process management
```

The example exhibits significant differences between our view of the module concept and other views [cf. SLAN (Koster [7]), MODULA (Wirth [17]), EUCLID (Lampson *et al.* [8])]. Several comments are appropriate.

1) The above is a module definition. Of course, it is intended that there be only one instance of this module in the system. We could have replaced the definition by a declaration (as it is possible in EUCLID).

2) There is no difference in the syntax of a module, a type, a procedure. (We could even have replaced the keywords TYPE and PROC by MODULE. This would sharply contrast usual language concepts.)

3) In the same way as a procedure and a type belong to a mode, so does a module. The mode of 'process management' consists of the "submodes" of the exported 'process,' 'current process,' and 'semaphore.'

We will not discriminate between "module" and "instance of a module" below. So a module can be considered as a "hyper-object," since it will usually possess own data. We will use the notion of *module replug* for the operation of module replacement; i.e., replacement of all the module's types and procedures, including own data restructuring (if necessary). Clearly, module replug is done on request. If, as a consequence of module replug, object restructuring is required as well, this will be done on demand.[3] As opposed to object replug on request previously referred to, there is only one "valid" type for each of the module's submodes at a time.

III. Data Restructuring

Data restructuring as a possible consequence of replugging applies to object data as well as to own data. Let us first attack the restructuring problem with regard to simple object replug on request as mentioned previously (no own data). For technical reasons, we will not restructure the value of an object *in situ*; rather, we will use a restructuring data-transfer algorithm which will "copy" a value from a 'source' object to an initialized 'destination' object of the same mode but of different type. Thus the latter is "filled" with information, whereas the former is "emptied" (i.e., it is left in its initial state).

[3] *Note:* Fabry's notion of "module" corresponds to our "type" or to our "module" without own data. Although not necessary, Fabry supports only one valid version at a time, so his "module change" is a special case of our "module replug."

We reject solutions that require a special conversion algorithm for every possible pair of types. Our attitude is as follows: Either of the two types involved shall contribute to the common goal, exploiting their own specific knowledge of the representational details of their objects.

The basic idea is that the mode be augmented with two operations 'in' and 'out,' the duty of which is the piecewise filling and emptying with information. 'In'/'out' and an additional operation 'nonempty' are specified in such a way that the loop

```
WHILE source.nonempty
DO    destination.in (source.out)    OD
```

implements the data transfer previously mentioned. Note that *by definition* this mechanism works independent of the types of 'source' and 'destination.' 'Out' and 'nonempty' are responsible for the source object, and 'in' is responsible for the destination object. The gap between the different fine structures of the source and the destination object is closed by means of a certain "grain of information" that is common to both, since it is a part of their common mode; this grain of information is determined by the mode of the item delivered by 'out' and accepted by 'in.' We require that this mode be a "basic" mode (e.g., 'integer') with only one type (which will never be subject to change).

The simplest possible example for in/out operations is given by the mode 'queue' which was presented earlier. The procedure 'nonempty' is already present, and the procedures 'enter' and 'remove' exactly represent the required in/out operations.

Of course, the in/out operations will be a real addition in the general case. For a somewhat more interesting example than the queue, look at the following mode 'symbol table':

```
MODE symbol table
IS ⟨insert (item: symbol; attr: attribute): bool;
    lookup (item: symbol; VAR attr: attribute): bool;
    in (item: symbol; attr: attribute);
    out: (item: symbol; attr: attribute);
    nonempty: bool⟩
  {'insert'   deposits '(item,attr)' in the table;
              returns 'false' if 'item' is
              already present in the table or if
              the table is full, otherwise 'true';
   'lookup'   delivers 'attr' of 'item';
              returns 'false' if 'item' is not
              present in the table, otherwise 'true'}
SI
```

'in' can be implemented by using 'insert,' but 'out' is a genuine, additional operation.

This approach to restructuring is still too simplistic. Note that the restructuring loop is mode-dependent: the type and size of the information unit transferred by the loop body may vary considerably with different modes. It may even happen that in/out operations with a unique parameter/result type just do not exist.

We will therefore adopt a somewhat different approach, which incorporates the restructuring loop in the source type. We introduce an operation 'restruct' which is claimed to have

the property that the restructuring process is represented by

```
source.restruct (destination)
```

instead of the above loop. In the simplest case, this new 'restruct' will be implemented as

```
PROC restruct = (destination : thismode)
                WHILE nonempty
                DO destination.in (out) OD
```

'restruct' can, of course, be much more complicated, by virtue of its specific knowledge of the mode, and can use perhaps more than one in/out pair, etc. Note that 'nonempty' and 'out' are now local procedures of the type.

We have tried this approach on several examples. We have, however, no final results concerning the existence of restructuring operations for arbitrary modes.

Data restructuring, as previously introduced, is applicable to object data as well as to own data. In some special cases, module replug can be achieved by obvious use of restructuring.

1) No types, no own data:
code replacement without restructuring.

2) No types but own data:
own data restructuring on request using a 'restruct' procedure.

3) One type, no own data:
object data restructuring on request, or on demand, as preferred, using a 'restruct' procedure.

4) Several types, no own data:
object data restructuring on request, or on demand, as preferred, using one 'restruct' procedure per type.

The general case, one or more types plus own data, is not handled that easily. We will discuss the problem in the context of the DAS system in Section V.

IV. Address Spaces in the DAS System

In most contemporary operating systems, a process exists in a fixed virtual address space which contains permanently allocated code and data. An alternative approach is to allow the process to move the "window" of its virtual address space over arbitrary segments of code and data which are identified by means of capabilities.

Some motivation for this approach is given, e.g., in Parnas *et al.* [13]: compensation for a small virtual address space and support for the "need-to-know principle" (see also Spier *et al.* [16] and Habermann *et al.* [6]). The DAS system developed at our university uses such address space management for an additional purpose: the segmentation structure supports the desired replugging; precisely expressed, certain code and data segments are the units of replugging.

The DAS system is structured as a collection of modules as they have been described in Section II. When a process executing in some module, calls upon another module, an *address space transition* is performed by means of a special CALL mechanism. The process leaves its current address space and enters a new one; this is accomplished by replacing some segments in the virtual address space by other segments. The new virtual address space will consist of:

1) a *code segment* containing the code for all the new module's procedures, including the operations for the module's types; the code is always pure and write-protected;

2) an *own segment* containing the module's own data (if any);

3) an *object segment* containing the data of the object to be manipulated (if any);

4) a *stack segment*, which is identical to the stack segment of the previous address space (before the CALL); it can be used for communication purposes between the calling and the called address space.

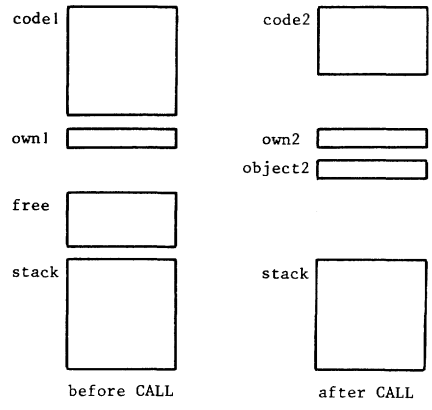

Fig. 1. Segments in the process's virtual address space.

One of the parameters of CALL is either a capability for a code segment, or also for an own segment (which is "connected"—as explained below—to the corresponding code segment), or also for an object segment (which is connected to the corresponding own segment and code segment). Thus a set of at least one and at most three segments serves as a "template" for the new address space which is actually built by combining the template segments with the stack segment.

Reestablishment of the calling address space is achieved by a special RETURN mechanism. In addition to CALL/RETURN there are operations that allow a process to change its address space incrementally by attaching or detaching other segments which are called "free segments." CALL and RETURN are implemented in the system's kernel.

Fig. 1 demonstrates the effect of an address space change caused by calling an object, provided that the calling space contains a free segment and no object segment. Note that the address space directly mirrors the different storage classes which result from the static nesting of procedures/types/modules as presented earlier. Procedure locals are located in the stack segment, type locals in the object segment, module locals in the own segment; the whole code is concentrated in the code segment.

The system kernel keeps track of segments by means of segment descriptors. Look at the simplified picture of a descriptor (requiring three machine words on a PDP-11) given in Fig. 2.

'kind' is an integer in the range of 0 . . 7, which indicates the segment kind as follows:

0: code
1: own
2 . . 5: object (a module may manage several submodes)

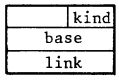

Fig. 2. Descriptor format.

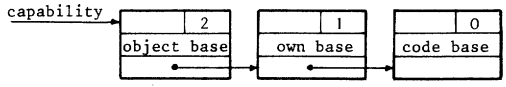

Fig. 3. Typical descriptor chain.

6: free
7: stack

'base' is the segment's physical address. 'link' is a pointer to another descriptor; it is used to establish connections between object, own, and code segments as previously mentioned. For example, starting with a capability for an object segment we can follow the descriptor chain given in Fig. 3. Typically, many object descriptors will point to one own descriptor.

The heart of the CALL mechanism is a routine that determines a new address map for the address mapping hardware, using the descriptors arrived at by following the chain. After the old map has been pushed on a process specific stack, the mapping hardware is loaded with the new map.

V. Fundamentals of the DAS Replugging Facility

If replugging a module and restructuring own data and/or object data is to go unnoticed by the users of the module and its objects, there must be an invisible indirection facility that makes it possible to replace one segment by another. The use of chained descriptors, as previously described, will show up as a sufficient means to accomplish this indirection.

Let us first study the simplest case of replugging: module or object replug without any data restructuring. Look at the above descriptor chain. Suppose that the object descriptor or the own descriptor is present. Then the code segment is simply replaced by changing the own descriptor's link field contents (or object descriptor, if there is no own descriptor).

We assume that this special kind of replug will frequently be used, e.g., to extend the functional capabilities of a module without changing the organization of data. So we have provided a special operation called 'chlink' which affects that which a descriptor just points to one code descriptor, then points to another.

chlink (data,newcode)

Note that this operation if applied to an own segment implements module replug, whereas if applied to an object segment (only if there is no own segment) implements object replug (on request).[4] (See Fig. 4).

Replacing a "stand-alone" code segment is effected by replacing its descriptor contents. There is an operation called 'repcod' that does so by exchanging the contents of the descriptors of the old and new segment.

repcod (code,newcode)

When 'repcod' has been executed, 'code' refers to the new code segment and 'newcode' to the old one. Note that 'repcod'

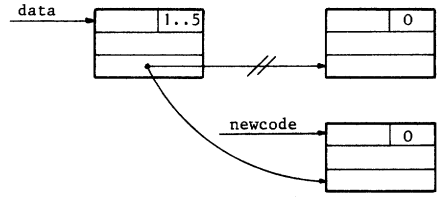

Fig. 4. chlink (data,newcode).

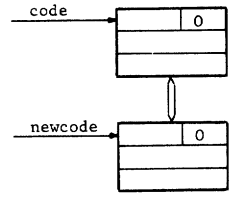

Fig. 5. repcod (code,newcode).

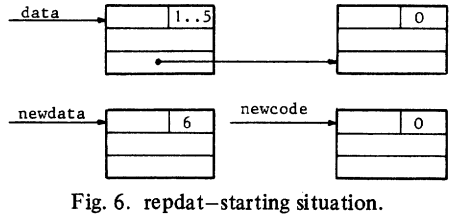

Fig. 6. repdat–starting situation.

represents code replacement "by value" and 'chlink' "by reference." (See Fig. 5). This has consequences for synchronization that will be discussed in Section VI.

Another operation called 'repdat' serves the purpose of general replugging-on-request, including data restructuring. 'repdat' has three parameters, 'data,' 'newdata,' 'newcode,' that give access to the following:

1) an object segment or own segment containing the data to be restructured ('data');

2) a free segment which will be filled with the restructured data ('newdata');

3) a code segment which represents the new type or the new module, respectively ('newcode').

Figs. 6–8 demonstrate the details of the 'repdat' operation.

repdat (data,newdata,newcode)

Step 1: A new object or module is created: the 'kind' of 'data' is copied to 'newdata'; 'newdata' is connected to 'newcode.'

Step 2: 'data.restruct(newdata)' is executed; this starts with initialization of the object (module) 'newdata' and thus is slightly different from Section III.

[4]The operation may also be used in order to assign some arbitrary mode to a given value representation (cf. Parnas *et al.* [13]).

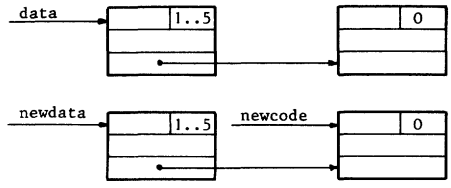

Fig. 7. repdat–effect of step 1.

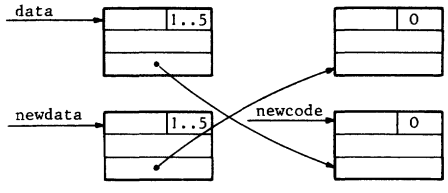

Fig. 8. repdat–final situation.

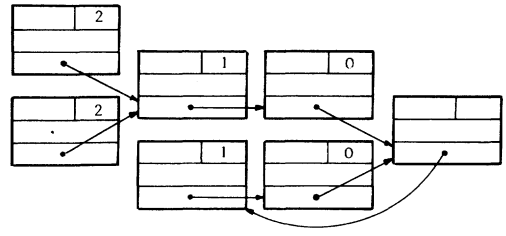

Fig. 9. repmod.

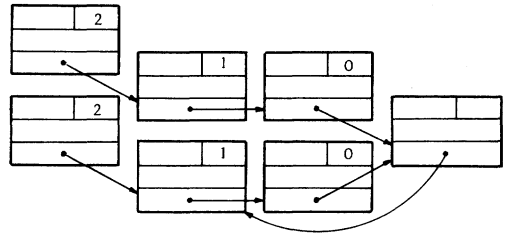

Fig. 10. Demand object replug following repmod.

Step 3: The contents of the 'data' and 'newdata' descriptors are exchanged. After this, 'data' identifies the restructured object (module), whereas 'newdata' identifies the old data segment, which is now emptied and can be discarded.

The 'repdat' operation requires that the data segment be directly connected to the code segment. This does not exclude application of 'repdat' to an own segment which one or more object segments point to. What happens if 'repdat' is applied to such an own segment?

After execution of 'repdat,' the objects will be connected to a restructured module with new code. Additional restructuring of object data is not necessary, provided the following conditions hold.

1) The new code does not differ from the old code as far as treatment of object data is concerned.

2) Representations of object data and own data are mutually independent (!).

(Note that condition 2 is violated if, e.g., object segments contain pointers into the own segment.)

If these conditions hold, module replug is accomplished by applying 'repdat' to the module's own segment. The object segments remain untouched.

The first condition can be dispensed with if additional object data restructuring on demand is provided as part of the CALL mechanism. For this purpose, CALL must be able to detect that the representation of the object to be called does not meet the actual implementation of the module. The problem cannot be solved by bare introduction of version numbers (as it is done in Fabry's approach). Our restructuring philosophy requires that the old code segment be discarded only when all corresponding objects have been restructured. However, this requirement is not fulfilled by the 'repdat' operation.

We propose the following approach (which is *not* implemented in the DAS system): any code descriptor points to a quasi-descriptor, the only purpose of which is to point to the *current version* of the module, i.e., an own descriptor (or code descriptor if there is no own data). Fig. 9 presents the situation after execution of a hypothetical operation 'repmod' (instead of 'repdat'); there are two objects which have not yet been called since then.

A CALL operation for 'victim' will detect that the object has to be restructured towards the current version before it can be operated on. Restructuring is performed using an additional scratch segment (like 'newdata' for 'repdat'). After exchanging the two object descriptors and deleting the scratch descriptor, CALL can activate the replugged object 'victim' (see Fig. 10).

It should be noted that retaining not only the old code segment but also the old own segment is by no means foolish. Until now we have tacitly assumed that restructuring own data is "separable" from restructuring object data, i.e., we have presupposed that restructuring of "abstract own data" can be accomplished by looking at own segments only and that the same is valid for object data. For many applications, however, this attitude is wrong.

Suppose a module exports one type and imposes a limit on the number of coexisting objects of this type. Then the object data can partially or completely be implemented as part of the own segment. (In fact, this is usually the reason for imposing that kind of limit! Example: process control blocks are part of the data of process objects; for efficiency reasons, however, they use to be maintained as own data of the process management module.) In this case, separability as previously postulated is not achievable. In/out operations for object restructuring will have to deal both with object segments and with own segments. On the other hand, module replug does *not* leave the old own segment in the empty state in this case; there remains some data pertaining to the currently existing objects, and this will successively be removed as the objects are replugged.

VI. Synchronization

If we allow concurrency in using a system component and replugging it, we are faced with synchronization problems.

For analysis and solution, let us start with modules that are not subject to any synchronization proper.

In the general case, using a module and replugging it must certainly be excluded from each other. Here we encounter a typical reader/writer exclusion problem, the users of the module representing the readers and the replugger(s) representing the writer(s); we will use the term "use/replug" exclusion.

Since we have refrained to burden all code segment entries and exits with clumsy synchronization operations (requiring extra address space changes for using the process management module!) we have decided to incorporate the users' contribution to the use/replug exclusion in the CALL/RETURN mechanism. Consequently, the most primitive synchronization technique is used, viz., busy waiting. (Note that this is unusual on a uniprocessor, but the problem cannot be solved by mere interrupt inhibition.) The use of busy waiting as opposed to processor preemption is also justified by the fact that replugging can be considered as an exceptional event and that system efficiency should not permanently be degraded due to the mere possibility of this event.

The necessary synchronization data are located in an additional descriptor word which has not been mentioned in the previous sections; every descriptor contains this additional word. The synchronization data consist of an integer 'usecount' (initially 0) and a Boolean 'repkey' (initially 'false'). The basic operations to be executed before/after using and before/after replugging are

enteruse / leaveuse ,
enterrep / leaverep .

They operate on the synchronization data of a descriptor. 'enteruse' is executed as a part of CALL for object, own, code descriptor; 'leaveuse' is executed as a part of RETURN. The 'enterrep'/'leaverep' pair is executed as a part of both 'repdat' (for the 'data' descriptor) and 'repcod' (for the 'code' descriptor). CALL, RETURN, 'repdat,' 'repcod' are all executed under interrupt inhibition.

The enter/leave operations are implemented as follows:

```
enteruse :                  enterrep :
  WHILE repkey                WHILE usecount>0 OR repkey
  DO enable; disable OD;      DO enable; disable OD;
  usecount:+1                 repkey := true

leaveuse :                  leaverep :
  usecount :-1                repkey := false
```

There are three points that should be noted about these operations.

1) Admitting interrupts during the busy waiting is obligatory. CALL, RETURN, 'repdat,' 'repcod' are programmed in such a way that this does no harm.

2) Repluggers are excluded from each other.

3) The users have priority over the repluggers! This kind of scheduling is quite unusual for readers/writers; an explanation will follow in Section VII.

Remember the operation 'chlink' which may be used instead of 'repcod' for the special case of module replug without restructuring. The implementation of 'chlink' does not require any synchronization. It is sufficient to execute 'chlink' in noninterruptible mode.

It is worthwhile to look at the differences between 'chlink' and 'repcod.' Suppose a process is just using the old code when 'chlink' is applied. The process will not be affected by making the own descriptor point to the new descriptor. The current memory map consists of pointers to the descriptors of the segments involved; it will not be changed by 'chlink.' Thus, even if the processor is preempted from the process for a while, the memory mapping hardware will always be loaded with the old code descriptor; this means that 'chlink' will only affect new users of the module.

If the necessity for use/replug exclusion would have been ignored in the implementation of 'repcod,' the following effect could occur: While a process is using the old code, the descriptor of the code segment is replaced by the new one. Although this does not directly affect a process that is just running in the corresponding address space, it will produce disaster if the processor leaves the address space (by means of CALL or a process switch) and returns to it later on: the old program counter will be dangling within the new code segment.

What is the runtime cost of our CALL/RETURN mechanism, especially concerning the code required for replugging? We have greatly benefitted by the microprogramming feature of the PDP-11/40E, which has been used extensively in order to make that cost acceptable. Address space transition is heavily supported by a bunch of so called "legalized instructions," i.e., microcoded routines that are referred to by unused operation codes of the original PDP-11/40.

A CALL/RETURN version that supports replugging differs from a nonreplug version by the synchronization only. The table contains execution times (in microseconds) for both versions. Note that the synchronization code is not yet supported by microprogramming in the present system version.

	CALL	RETURN
nonreplug	372	298
replug	442	388

Concerning the gain by microprogramming, note that a nonreplug version of CALL/RETURN requires 1875/1251 μs if not supported by legalized instructions.

VII. Design Issues

The replug operations 'chlink,' 'repcod,' and 'repdat' form a module which is called "replug module." This replug module does not provide any type. Its own segment contains the descriptor pool. This segment is also part of the fixed address space of the system's kernal, i.e., the kernel and the replug module[5] share a common data base.

For this reason, the replug module's own segment does in fact *not* contain own data in the sense of the previous sections. We see that different modules may share knowledge of repre-

[5] Actually, also the memory management module.

sentational details of a data base which ought to be managed by one module (cf. Habermann *et al.* [6] and Parnas [14]).

Which is the position of the replug module within the system's functional hierarchy? The module does not make use of any system feature except CALL/RETURN and may thus be located as low as its application by other system modules requires. But since no other module does apply replug operations, the replug module may reside in an arbitrary position within the hierarchy. We decided to make it swappable, i.e., to place it above the memory management module.

Replugging can be applied to any system module, including the replug module itself. It must be taken into account, however, that application of the 'repdat' or 'repcod' operation is delayed until the module at issue is not in use. Now, the lower a module ranks in the functional hierarchy, the more it will be demanded by the processes. Thus the low system levels will tend to resist replugging because they are frequently used.

An extreme example is the process management module. At any time, all processes but one (the running process, if any) are using the process management module; they have entered it on behalf of an explicit synchronization operation or a time slice (or other) interrupt, and they have not yet left it since they have been deactivated by a process switch. Here the ability of 'chlink' to perform at least code replacement with no exclusion required comes in handy. If we would not have provided this operation, replugging the process management module would always require deletion of all processes except the replugging process; for the user community this is tantamount to a system shutdown.

As we said earlier, the replug module is even able to replug itself. As previously explained, the module's own data can never be subject to restructuring, but the code can be exchanged. This is again accomplished using the 'chlink' operation; while the old code is executing 'chlink' the new code is being installed by just this execution of 'chlink.'

Remember the unusual scheduling discipline of our use/replug exclusion: the users are favored above the repluggers. Why? In presenting the synchronization requirements for replugging, we have intentionally ignored modules with inherent synchronization. The reason is most simple: it *can*, in fact, be ignored. Any synchronization proper that is executed on behalf of the module's code is irrelevant to the replugging issue.

There is one remarkable exception, though. Suppose a module has two procedures the one of which may cause the executing process to block; the other may have the effect of waking that process. A process that is blocked within the module is currently using this module. Execution of, e.g., 'repdat,' for the module's own segment will be delayed in this situation. Now, if repluggers were favored above users, execution of the procedure that would wake the blocked user would be delayed as well, and so both would be deadlocked!

Favoring the user has been chosen in order to avoid this kind of deadlock.

VIII. Conclusion

We have demonstrated that replugging is feasible and that incorporating a replugging facility in a real operating system which provides address space management can be achieved at low cost.

The present version of the DAS system provides a restricted replugging mechanism which does only allow *either* own data *or* object data restructuring within one module; this will always be done on request. We take the view that if a module manages both own and object data, we will mainly be interested in own data restructuring; this is one of the differences between Fabry's approach and ours. We have explained how the limitations of both Fabry's and our approach can be overcome.

The resistance of low system levels to replugging is somewhat annoying, but is clearly an inherent structural property.

Our work has been a first step towards a really usable system which has to provide support for replugging in several areas (e.g., separate compilation, error detection, replug commands, replug rights). Also a major research effort is still required in order to investigate the restructuring philosophy based on the in/out principle. A module can be restructured only if there are in/out procedures. As a comfort, the operation 'chlink' can be used for postsubmission of the 'restruct' procedure if this is not yet available in the active version of a module.

Acknowledgment

We appreciated the referees' suggestions for improving the presentation of our material. Design and implementation of the DAS address space management has been a common effort of the DAS project group, especially H. Mauersberg and J. Müller. P. Nellessen has made valuable contributions by clarifying basic replugging issues in his Master's thesis. We are especially grateful to N. Habermann who, as the head of the project, has been a permanent source of advice and encouragement.

References

[1] F. L. Bauer, Ed., *Software Engineering—An Advanced Course* in *Lecture Notes in Computer Science*, vol. 30. Berlin: Springer, 1975.

[2] R. S. Fabry, "How to design a system in which modules can be changed on the fly," in *Proc. 2nd Int. Conf. Software Eng.*, 1976.

[3] C. M. Geschke, J. H. Morris, and E. H. Satterthwaite, "Early experience with MESA," *Commun. Ass. Comput. Mach.*, vol. 20, 1977.

[4] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types," IBM Thomas J. Watson Res. Center, Yorktown Heights, NY, Rep., 1976.

[5] J. A. Guttag, E. Horowitz, and D. R. Musser, "The design of data type specifications," in *Proc. 2nd Int. Conf. Software Eng.*, 1976.

[6] A. N. Habermann, L. Flon, and L. Cooprider, "Modularization and hierarchy in a family of operating systems," *Commun. Ass. Comput. Mach.*, vol. 19, 1976.

[7] C. H. A. Koster, "Visibility and types," *Ass. Comput. Mach. SIGPLAN Notices*, vol. 11, 1976.

[8] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. L. Popek, "Report on the programming language EUCLID," *Ass. Comput. Mach. SIGPLAN Notices*, vol. 12, 1977.

[9] J. Müller *et al.*, "Address space management in the DAS operating system," TR78-20, Fachbereich Informatik, TU Berlin, 1978.

[10] P. Nellessen, "On replacing software system components at runtime," Diplomarbeit Fachbereich Informatik, TU Berlin, 1977.

[11] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. Ass. Comput. Mach.*, vol. 15, 1972.

[12] D. L. Parnas, G. Handzel, and H. Würges, "Design and specification of the minimal subset of an operating system family," *IEEE Trans. Software Eng.*, vol. SE-2, July 1976.

[13] D. L. Parnas, J. E. Shore, and D. Weiss, "Abstract types defined

as classes of variables," *Ass. Comput. Mach. SIGPLAN Notices*, vol. 11, 1976.
[14] D. L. Parnas, "Some hypotheses about the 'uses' hierarchy for operating systems," Fachbereich Informatik, TH Darmstadt, 1976.
[15] M. Shaw, "Research directions in abstract data structures," *Ass. Comput. Mach. SIGPLAN Notices*, vol. 11, 1976.
[16] M. J. Spier, T. N. Hastings, and D. N. Cutler, "A storage mapping technique for the implementation of protective domains," *Software–Practice & Experience*, vol. 4, 1974.
[17] N. Wirth, "Modula: A language for modular multiprogramming," *Software–Practice & Experience*, vol. 7, 1977.

Hannes Goullon, photograph and biography not available at the time of publication.

Rainer Isle, photograph and biography not available at the time of publication.

Klaus-Peter Löhr, photograph and biography not available at the time of publication.

End of Special Collection

Guest Editorial
Software Management: We Must Find a Way

Introduction

SOFTWARE management is a term which inspires many different reactions from the broad range of practitioners in the field of data processing. The term has many dimensions ranging from the first and second line manager of a software development activity who plies his trade with little to fall back on other than his prior experience, and native skills and intuition. Contrast this with a senior executive who, because of lack of understanding of software management, is frustrated in trying to determine if his company can produce a software program on schedule and within budget. The data processing landscape is littered with examples of major software developments which faltered and then expired. That this situation should exist, after almost a quarter-century of experience with implementing these systems, is one of the enigmas of the rapidly expanding data processing industry.

What Follows

This special section was inspired by the series of seminars [1] on software management organized by the American Institute of Aeronautics and Astronautics (AIAA) which addressed software management from the viewpoint of the Department of Defense (DOD). The first paper by De Roze and Nyman echoes the theme of this seminar–i.e., that DOD is launching a major effort to bring the management of software development under better control. Starting with this global view of the problem, succeeding papers by Cooper, Cave and Salisbury, and McHenry and Walston examine software management from the more restrictive viewpoints of the system developer within the framework of a major procurement agency, the program manager responsible for a system containing an embedded computer resource, and finally the system implementer who gambles his system expertise in an attempt to successfully develop a system on schedule and at a profit, respectively. A final paper by Putnam addresses a specific technique developed within DOD which appears to offer the senior executive a global view of what tradeoffs are available to determine the risk factors in a proposed or ongoing software development program.

Fuzzy Function of Management

Because of the lack of understanding of software management techniques, papers that address this topic must deal in generalities, rules of thumb, and other issues which may be regarded by some as banalities. Under these circumstances some readers may well question whether such an undefined subject should be addressed within this TRANSACTIONS. While the subject may be hard to bring into focus, the desire for available information is intense as was demonstrated by the interest shown in the AIAA seminars which drew large audiences not only in this country but overseas. The papers in this special section bring to the reader a description of key software management issues, strategies designed to cope with these issues, and advice on how better strategies can be evolved.

0098-5589/78/0700-0307$00.75