The Implementation of Abstract Objects in a Capability Based Addressing Architecture

P. Corsini and G. Frosini
Istituto di Elettronica e Telecomunicazioni, Università di Pisa, Pisa, Italy
L. Lopriore
Istituto di Elaborazione dell'Informazione, Consiglio Nazionale delle Ricerche, Pisa, Italy

The problem of implementing abstract objects in a system having a capability based addressing is analysed in some detail. Hardware features needed to support a classical capability environment are first pointed out, and a simple implementation of abstract objects is presented. Then a generalization of the classical capability environment is given, which allows one to solve the stated problem in a more efficient way.

1. INTRODUCTION

In this paper we analyse the problem of giving a run-time support to the implementation of abstract objects, which constitute a well known salient feature of modern programming languages.[1,2] The characteristic of this approach, with respect to a compilative implementation, is that the data encapsulation is maintained in a protected environment during all the life of the object.

The protection environment to which we refer, recalled in the first part of the paper, is based on a capability addressing mechanism.[3–7] With this mechanism, the protected entities consist of memory segments, and the active entities able to perform access attempts (subjects) are provided with tickets, called capabilities, which give a subject the proper access rights to memory segments. More precisely, a capability is a pair (*AR*, *ID*), where *ID* is a unique identifier for a memory segment and *AR* is a set of access rights on that segment. The set of capabilities each subject is provided with represents its protection domain.

It is first shown that in a classical capability environment the implementation of abstract objects leads to a noticeable memory waste (owing to the replication of protection domains), and a consequent proliferation of memory segments.[8] Thus, a generalization of the concept of capability is proposed, by introducing pseudo-capabilities and a mechanism of access right amplification, thereby allowing a more efficient solution to the stated problem.

2. PROTECTING MEMORY SEGMENTS VIA CAPABILITIES

Let us refer to a system with many virtual processors VP_1, VP_2, ..., VP_r (each one allocated to a specific process), that have access to a common segmented virtual memory CVM, and let us consider the problem of protecting the segments of CVM from malicious or accidental access attempts.

Since the common virtual memory is segmented, the virtual address of a word has two components: the segment identifier *ID* and the offset *W*. The virtual memory is mapped into a physical memory by an addressing mechanism that provides, for every segment identifier *ID*, the base address and the limit address of the segment in the physical memory. Such an addressing mechanism also adds the offset *W* to the base address and checks that the address so obtained does not exceed the limit address.

Depending on the nature of the information stored in a segment of CVM, different *access rights* are associated with the segment itself. As an example, if a segment contains data, the possible access rights are READ and WRITE; if a segment contains a code, a further access right is EXECUTE.

A pair (*AR*, *ID*), where *AR* is a set of possible access rights and *ID* a segment identifier, is called *capability*. A set of capabilities specifies the way a subset of the segments of CVM can be accessed, i.e. specifies an access *domain* d_h.

At given time, a process p_i, $i = 1, 2, \ldots, r$, must operate in a specific domain d_h, in the sense that it can access segments only as specified by d_h itself. A pair (p_i, d_h) is called *subject* $s_{i,h}$, and represents the active entity whose access to segments must be validated. During its evolution, a process p_i can operate on different domains, and, when it switches from domain d_h to domain d_k, we shall say that subject $s_{i,h}$ *enters* subject $s_{i,k}$. In order to access a segment ID° in a manner α, a subject $s_{i,h}$ must possess in the pertinent domain d_h a capability (AR°, ID°), where AR° contains an access right consistent with the action α. Subjects are prevented from forging capabilities, but, if they need new segments, can ask a special protection monitor for new capabilities. Moreover, subjects can be authorized to transfer capabilities from one domain to another.

Since it is not viable for the protection monitor to keep trace of all the capabilities transferred among different domains, we will suppose that CVM is so large that each new request for segments by a subject can be honoured without reusing previously allocated segments. In other words, capabilities with the same identifier will never be provided by the protection monitor in the system life. As an example, a segment identifier of 48 bits allows the monitor to provide a new capability every 10 μs for about 75 years.

Referring to the problem of transferring capabilities,

CCC-0010-4620/84/0027-0127$04.00

a special access right COPY can be associated with any other access right R in the AR field and, when this happens, we shall say that the copy flag of R is set. Only access rights having the copy flag set can be passed when a capability is transferred from one domain to another.

3. IMPLEMENTING A CAPABILITY BASED ADDRESSING

A segment of the virtual memory CVM can store, besides data and codes, also capabilities. The set of all the capabilities defining a domain is structured as a rooted tree. Each node of the tree is stored in a segment of CVM: the segment storing the root is called the *base capability segment* (BCS), and the segments storing the other nodes are called *auxiliary capability segments* (ACSs). In the simplest case, all the capabilities are stored in one segment only, that is in the BCS. Access rights relative to a segment containing capabilities are TAKE and GRANT, that allow a subject to read and to store a capability, respectively. Owing to the rooted tree structure of the set of capabilities defining a domain, in a capability segment relative to a node N (excluding the segments relative to the leaves) there are capabilities for the capability segments relative to the sons of N, with access rights TAKE and/or GRANT. This allows a subject to use capabilities stored in any capability segment, as will be explained in the following.

Let us now examine in some detail a way to implement a capability based addressing. Let us suppose that each virtual processor, besides the classical resources that allow the process to perform its action, is provided with some special registers CR_0, $CR_1, \ldots, CR_{n-1}$, called *capability registers*, whose aim is to store those capabilities that are actually used in order to access memory segments. By hardware convention, when a subject is entered, two of such registers, say CR_0 and CR_1, are properly initialised, that is:

(a) CR_0 is loaded with the capability for the code segment to be executed first.
(b) CR_1 is loaded with the capability for the BCS of the entered subject.

The program counter PC must be more powerful than the conventional one, as it must contain (a) the index of the capability register containing the capability for the code segment in execution, and (b) the offset W_0 of the instruction in such a segment. The previous hardware conventions imply that PC, when a subject is entered, is loaded with the index of CR_0 and with the offset zero. The initial loading of CR_0, CR_1 and PC is performed by the special instruction *Enter*, as will be explained later.

In order to better understand the behaviour of the system, let us consider a virtual processor VP_i allocated to a specific process p_i, and let us suppose that a subject $s_{i,h}$ is active and is executing the fetch phase of an instruction. The following actions are performed (Fig. 1):

(1) Consider the capability in the register whose index is stored in PC, say (AR_0, ID_0).
(2) If AR_0 contains the access right EXECUTE, the fetch phase continues, otherwise an access violation occurs.

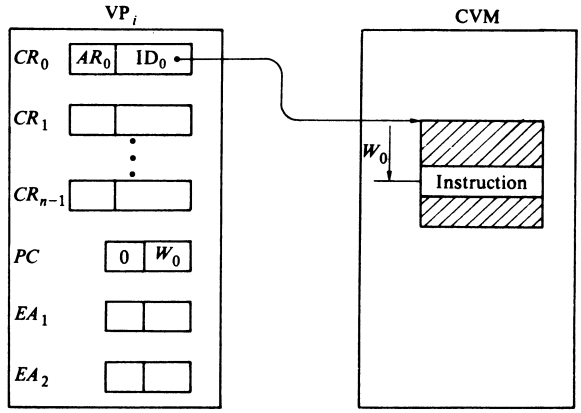

Figure 1. Actions involved in the fetch phase of an instruction.

(3) By using the pair (ID_0, W_0) find in the corresponding segment the addressed word, that contains the instruction to be executed.
(4) Evaluate the logical addresses of the operands and store them into appropriate registers $EA_1, EA_2, \ldots$ of the virtual processor.

In the execution phase, operands are accessed in the virtual memory by using the contents of the registers $EA_1, EA_2, \ldots$, each one specifying:

(i) the index of the capability register containing the capability for the data segment
(ii) the offset in the data segment of the operand.

The first instructions to be executed must be special instructions that load into the capability registers capabilities taken from capability segments. An instruction of such a type has the special form

Loadcap CR_i, W, CR_j

and its execution is performed according to the following actions (Fig. 2):

(1) Consider the capability stored in CR_i, say (AR, ID).
(2) If AR contains the access right TAKE then the execution phase continues, otherwise an access violation occurs.

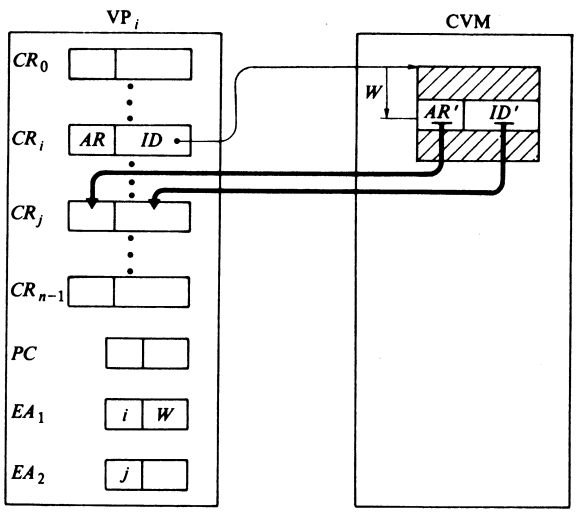

Figure 2. Actions involved in the execution of the *Loadcap* instruction.

(3) By using the pair (*ID*, *W*) find, in the addressed capability segment, the capability to be loaded, and store it into the capability register CR_j.

Note that, when a subject becomes active, the *Loadcap* instruction that is executed as the first instruction must have $CR_i = CR_1$. Note also that, by executing successive *Loadcap* instructions, capabilities stored in auxiliary capability segments of any level can be loaded into the capability registers (recall the rooted tree structure of the set of capabilities defining a domain).

When the active subject has loaded the appropriate capabilities into the capability registers, it can execute standard instructions having the form

Do α to CR_i, *W*

whose execution involves the following actions (Fig. 3):

(1) Consider the capability stored in CR_i, say (*AR*, *ID*).
(2) If *AR* contains an access right consistent with the action α, then the execution phase continues, otherwise an access violation occurs.
(3) By using the pair (*ID*, *W*) find the addressed word in the corresponding segment and perform the action α on it.

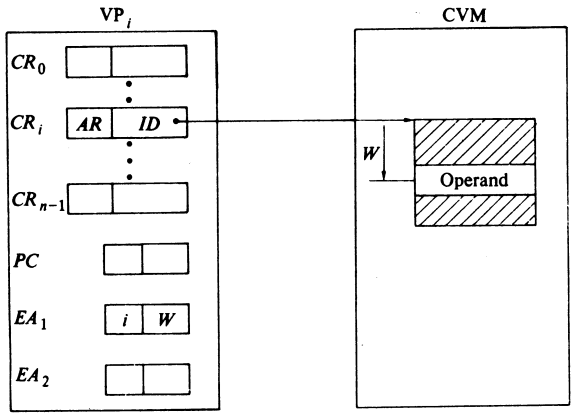

Figure 3. Actions involved in the execution of the generical instruction.

If the instruction to be executed is a *Jump* instruction, having the form

Jump CR_i, *W*

the execution simply consists in loading the index *i* and the offset *W* into the program counter. The *Jump to Subroutine* and the *Return from Subroutine* instructions are executed in a similar way.

In executing all the previous instructions, the active subject always remains the same, that is the process does not change domain. Let us now introduce two special instructions, called *Enter* and *Reenter*, that allow a subject $s_{i,h}$ to enter subject $s_{i,k}$, and the subject $s_{i,k}$ to return to the initial subject $s_{i,h}$. A further access right, called ENTER, is also introduced, that allows the *Enter* instruction to be executed. The *Enter* and *Reenter* instructions require the existence of a stack for each process, in which the linking information is pushed and popped, respectively. The *Enter* instruction has the form

Enter CR_i, *C'*

and its execution involves the following actions (Fig. 4):

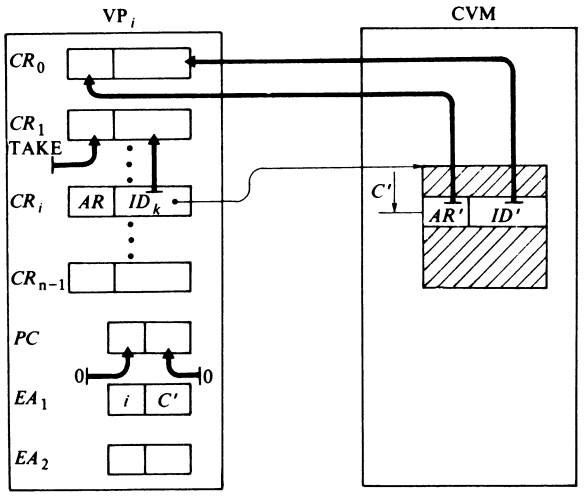

Figure 4. Actions involved in the execution of the *Enter* instruction, in addition to the saving of the contents of capability registers and *PC* in the process stack.

(1) Store the current values of CR_0, CR_1, . . . , CR_{n-1} and of *PC* in the process stack.
(2) Consider the capability stored in CR_i, say (*AR*, ID_k).
(3) If *AR* contains the access right ENTER, then the execution phase continues, otherwise an access violation occurs.
(4) By using the pair (ID_k, *C'*), find in BCS_k the addressed capability, say (*AR'*, *ID'*), and load it into CR_0.
(5) Load the capability register CR_1 with the new capability (AR_k, ID_k), where AR_k is constituted by the access right TAKE.
(6) Load the program counter with the index 0 and the offset 0.
(7) Destroy the contents of all the capability registers, excluding CR_0 and CR_1.

Note that the aim of point (7) is to prevent the entered subject from using the capabilities of the entering one.

A subject $s_{i,k}$ entered by a subject $s_{i,h}$ by means of an *Enter* instruction can return to the initial subject $s_{i,h}$ by issuing the instruction

Reenter

whose execution consists in loading the registers CR_0, CR_1, . . . , CR_{n-1} and *PC* with the values popped from the top of the process stack.

In a capability environment, capabilities can be transferred from one subject to another. A special instruction is provided for this aim, having the form

Transfer MSK, CR_i, *AC*, CR_j, *AC'*

Its execution involves the following actions (Fig. 5):

(1) Consider the capability stored in CR_i, say (*AR*, *ID*), and the capability stored in CR_j, say (*AR'*, *ID'*).
(2) *If AR* contains the access right TAKE and *AR'* contains the access right GRANT, then the execution phase continues, otherwise an access violation occurs.
(3) By using the pair (*ID*, *AC*), find in the pertinent capability segment the capability to be transferred, say (*AR**, *ID**).
(4) Construct an access right field AR^0 by (a) taking from *AR** only the access rights having the copy flag

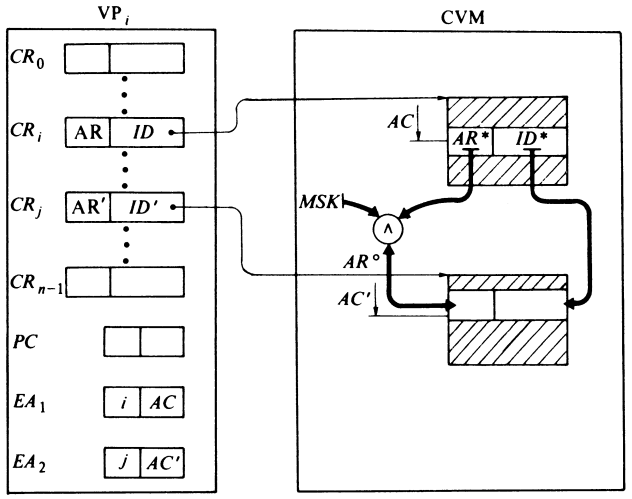

Figure 5. Actions involved in the execution of the *Transfer* instruction.

set, and (b) masking such access rights with the *MSK* field.

(5) Transfer the capability (AR^0, *ID**) into the location specified by (*ID'*, *AC'*).

Note that the *Transfer* instruction operates between capability segments, and so it is not possible to transfer a capability contained in a capability register. This fact prevents a subject from granting another subject the capability for its BCS, with the access right TAKE, which was constructed in CR_1 when the subject was entered.

Observe that it is not viable to give a subject $s_{i,h}$ the possibility of transferring capabilities into the base capability segment of a domain d_k: in fact, in this case $s_{i,h}$ could destroy any capability in d_k. Instead, the transfer must have an auxiliary capability segment as its destination. Moreover, if a subject $s_{i,h}$ wants to transfer all the capabilities contained in an auxiliary capability segment of its domain into another domain, it can simply transfer into that domain the capability for the auxiliary capability segment, with the access right TAKE.

4. A SIMPLE IMPLEMENTATION OF ABSTRACT OBJECTS

A system having a capability based addressing represents an environment well suited for implementing abstract objects of a generical type. An implementative approach, whose main merit is simplicity, will be given in this Section. As a working example, we will assume that a capability is 8 bytes long, 2 bytes for the *AR* field and 6 bytes for the *ID* field. Moreover, we will refer to abstract objects of the type *Regular_polygon*, which is defined as follows (an arbitrary self-explanatory notation is used):

```
type Regular_polygon is
  representation Regular_polygon is
    record
      edge_number: integer;
      edge_size: real;
    end record;
  function Init (edge_number: in integer; edge_size: in real)
                  return Regular_polygon is
    --returns a regular polygon with given edge number and
    --edge size.
    begin
      return (edge_number, edge_size);
    end;
  function Similar (polygon: in Regular_polygon; scale: in real)
                  return Regular_polygon is
    --returns a regular polygon similar to a given one.
    begin
      return (polygon . edge_number, scale * polygon . edge_size);
    end;
  function Perimeter (polygon: in Regular_polygon)
                  return real is
    --returns the perimeter of a given regular polygon.
    begin
      ......
    end;
  function Area (polygon: in Regular_polygon)
                  return real is
    --returns the area of a given regular polygon.
    begin
      ......
    end;
end type;
```

Let us suppose that an object *Pol* of the type *Regular_polygon* is defined, i.e.

declare *Pol*: *Regular_polygon*;

A simple implementation of *Pol* uses a base capability segment BCS_{Pol}, an auxiliary capability segment ACS_{Pol}, four code segments CD_1, CD_2, CD_3, CD_4, and a data segment IRS_{Pol} (Fig. 6). The BCS_{Pol} contains: (i) the capabilities for the four code segments, each with the access right EXECUTE (each code segment contains the re-entrant codes relative to an operation defined in the *Regular_polygon* type declaration); (ii) the capability for the data segment IRS_{Pol} with the access rights READ and WRITE (this segment is reserved for containing the internal representation of the object *Pol*); (iii) the

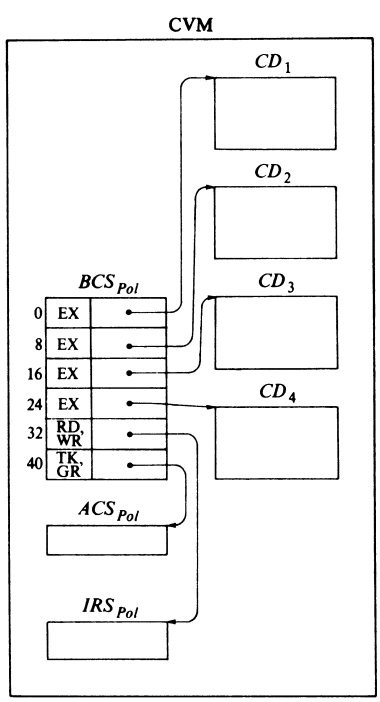

Figure 6. Memory configuration for implementing the abstract object *Pol*.

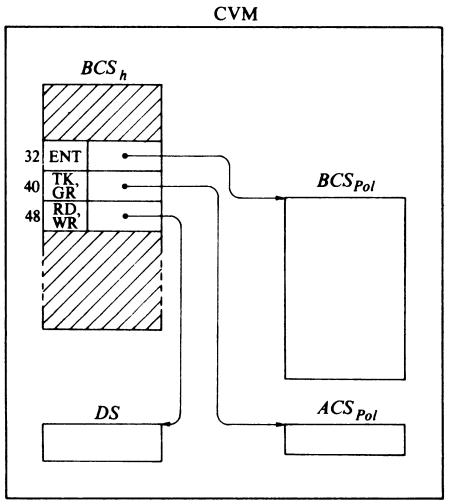

Figure 7. Structure of the base capability segment of a subject able to operate on the abstract object *Pol* implemented as in Fig. 6.

capability for ACS_{Pol} with the access rights TAKE and GRANT (this segment will be used for the transmission of the capability for the segment reserved for containing the actual parameter values).

A subject $s_{i,h}$ is able of operating on the abstract object *Pol* if its base capability segment BCS_h contains (Fig. 7): (i) the capability for BCS_{Pol} with the access right ENTER; (ii) the capability for ACS_{Pol} with the access rights TAKE and GRANT; (iii) the capability for a parameter data segment *DS* with the access rights READ and WRITE. When $s_{i,h}$ is active, the capability register CR_1 contains the capability for BCS_h with the access right TAKE, and CR_0 contains the capability for the actually running code, with the access right EXECUTE. If $s_{i,h}$ wants to perform the operation *Init* on the abstract object *Pol*, it must first store the actual values of the parameters *edge_number* and *edge_size* in the parameter data segment *DS*. Then it executes the following instructions (the memory situation shown in Fig. 7 is hypothesized):

Loadcap	$CR_1, 40, CR_2$
Transfer MSK WRITE,	$CR_1, 48, CR_2, 0$
Loadcap	$CR_1, 32, CR_3$
Enter	$CR_3, 0$

The first two instructions transfer a capability for the parameter data segment *DS*, with the access right READ, from BCS_h to ACS_{Pol}. The two remaining instructions perform a domain switch so entering the subject $s_{i,Pol}$, which executes the instructions relative to an *Init* operation. These instructions are the following:

Loadcap	$CR_1, 32, CR_2$
Loadcap	$CR_1, 40, CR_3$
Loadcap	$CR_3, 0, CR_3$
Move	$CR_3, 0, CR_2, 0$
Movelong	$CR_3, 2, CR_2, 2$
Reenter	

The first instruction loads the capability for the internal representation segment IRS_{Pol} into CR_2. The second instruction loads the capability for ACS_{Pol} into CR_3. The third instruction loads the capability for *DS*, from ACS_{Pol} into CR_3. The *Move* instruction moves the integer, which constitutes the actual value of the parameter *edge_number*, from *DS* into IRS_{Pol} (such an integer is supposed to be 2 bytes long). Similarly, the *Movelong* instruction moves the real, which constitutes the actual value of the parameter *edge_size* (such a real is supposed to be 4 bytes long). These two instructions correspond to the body of the function *Init*. Finally, the last instruction re-enters subject $s_{i,h}$.

It should be noted that the capability based environment allows a rigid encapsulation of the object's internal representation which cannot be directly modified by subject $s_{i,h}$ in any way: accesses on the internal representation can only be performed by using the operations defined in the data type specifications.

The noticeable disadvantage of this implementation of abstract objects is the wide memory waste which originates if several objects of the same type are to be implemented. As an example, if we want to implement *m* objects *Pol*1, *Pol*2, ..., *Polm* of the type *Regular_polygon*, we must introduce *m* base capability segments $BCS_{Pol1}, BCS_{Pol2}, \ldots, BCS_{Polm}$, *m* auxiliary capability segments $ACS_{Pol1}, ACS_{Pol2}, \ldots, ACS_{Polm}$ and *m* internal representation segments $IRS_{Pol1}, IRS_{Pol2}, \ldots, IRS_{Polm}$: only the code segments CD_1, CD_2, CD_3, CD_4, do not need to be replicated. In Fig. 8 the case $m = 2$ is shown.

The implementation of the generical object *Polj* requires 48 bytes for BCS_{Polj}, 8 bytes for ACS_{Polj}, and 6 bytes for IRS_{Polj}. So, the memory overhead associated with *Polj*, defined as $ov_j = (\delta_j - \gamma_j)/\delta_j$, where δ_j is the memory needed for the global implementation of *Polj*, and γ_j is the memory needed for its internal representation, is about 0.9. To overcome this wide memory waste,

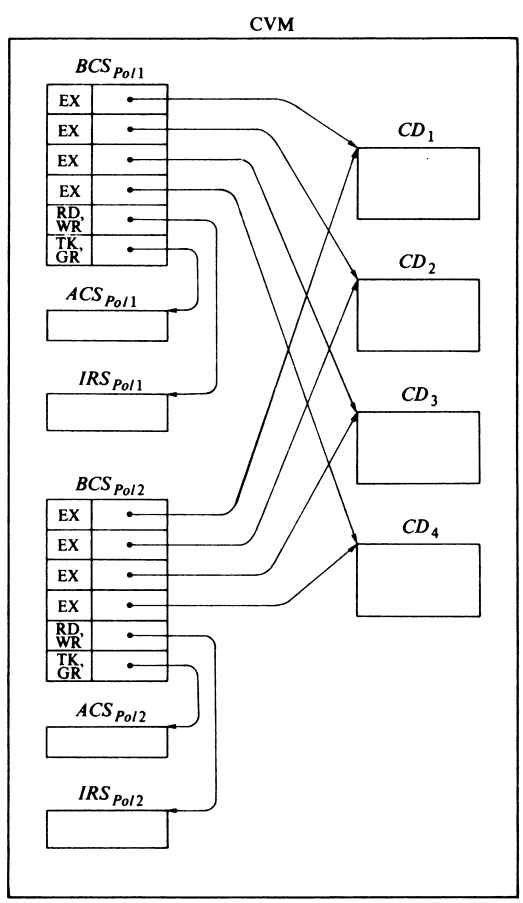

Figure 8. Memory configuration for implementing two abstract objects *Pol1* and *Pol2*.

we will generalize the concept of capability and introduce the mechanism of access right amplification as follows in Section 5. Successively, we will show that the above generalization and the new mechanism allow the implementation of several objects of the same type, without any replication of BCSs and of ACSs (Section 6).

5. GENERALIZING THE CAPABILITY CONCEPT

Let us now enlarge the capability environment described in Sections 2 and 3, by: (i) introducing the concept of extended capability; (ii) lightly modifying the previously described instructions; (iii) introducing the new special instruction *Amplify* and the new access right AMPLIFY.

An *extended capability* is a triplet (F_1, F_2, ID), where: (i) F_1 is a 1-bit field specifying if the extended capability is a *true-capability* or a *pseudo-capability*; (ii) F_2 represents an access right field if the extended capability is a true-capability, or an offset field if the extended capability is a pseudo-capability; and (iii) ID is a unique identifier for a memory segment. Capability registers must now be reformatted to contain also the F_1 field. True-capabilities have exactly the same aim as capabilities, as described in Section 2. Instead, a pseudo-capability is used in order to point to a specific entry (i.e. the entry specified by the offset field F_2) of a capability segment (i.e. the segment identified by the field ID). Pseudo-capabilities can only be loaded into capability registers, and transferred from one domain to another, but they cannot be used for effective memory accesses.

All the previously introduced instructions must be modified to treat extended capabilities. In particular, all instructions, excluding the *Loadcap* and *Transfer* instructions, must abort if they work on pseudo-capabilities.

In order to understand the aim of pseudo-capabilities, let us introduce the new special instruction *Amplify* and the new access right AMPLIFY. An *Amplify* instruction requires, as its operands, a true-capability for a capability segment with the access right AMPLIFY, and a pseudo-capability for an entry of the same segment. The result of the instruction execution is the loading into a capability register of the extended capability pointed to by the pseudo-capability. More precisely, the instruction has the form:

$$Amplify \quad CR_i, CR_j$$

and its execution phase involves the following actions (Fig. 9):

(1) Consider the extended capability stored in CR_i, say (F_1, F_2, ID), and the extended capability stored in CR_j, say (F'_1, F'_2, ID').
(2) If (F_1, F_2, ID) is a true-capability, if F_2 contains the access right AMPLIFY, if (F'_1, F'_2, ID') is a pseudo-capability, and if $ID' = ID$, then the execution phase continues, otherwise an access violation occurs.
(3) By using the pair (ID', F'_2), find the addressed extended capability in the pertinent capability segment, and load it into the capability register CR_j.

Observe that the loading of an extended capability from a capability segment CS into a capability register CR_j can also be performed by using a *Loadcap* instruction.

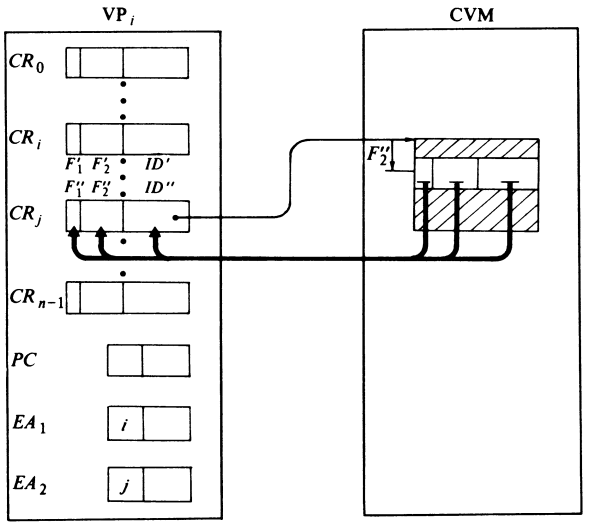

Figure 9. Actions involved in the execution of the *Amplify* instruction.

While the *Loadcap* instruction only requires a true-capability for CS, with the access right TAKE, the *Amplify* instruction requires both a true-capability for CS, with the access right AMPLIFY, and a pseudo-capability for the proper entry of CS. Moreover, while the true-capability for CS with the access right TAKE (required by the *Loadcap* instruction) allows the loading of every extended-capability contained in CS, the true-capability for CS with the access right AMPLIFY and the pseudo-capability for an entry of CS (both required by the *Amplify* instruction) only allow the loading of a specific capability stored in CS.

Note that the same true-capability with the access right AMPLIFY can be used together with different pseudo-capabilities, in order to load different extended capabilities taken from the same capability segment into capability registers. So, the true-capability behaves as an amplification device, whose input is represented by the pseudo-capability, and whose output is the true-capability that replaces the involved pseudo-capability in the capability register.

A typical application of the mechanism of access right amplification is the following: suppose we want (i) a subject $s_{i,h}$ to be able to authorize another subject $s_{i,k}$ to access a specific memory segment DCS, but (ii) $s_{i,h}$ itself not to be able to access DCS. In this case, we provide $s_{i,k}$ with a true-capability with the access right AMPLIFY for a capability segment CS, and $s_{i,h}$ with a pseudo-capability for a specific entry of CS, containing in its turn a true-capability for DCS. Neither $s_{i,h}$ nor $s_{i,k}$ can separately access DCS; but, if $s_{i,h}$ transfers the pseudo-capability into the domain of $s_{i,k}$, then $s_{i,k}$ can execute an *Amplify* instruction, in order to obtain the true-capability for DCS from CS.

6. AN EFFICIENT IMPLEMENTATION OF ABSTRACT OBJECTS

A different approach for an efficient implementation of abstract objects is now given. We suppose that the length of an extended capability is still 8 bytes, i.e. the F_2 field

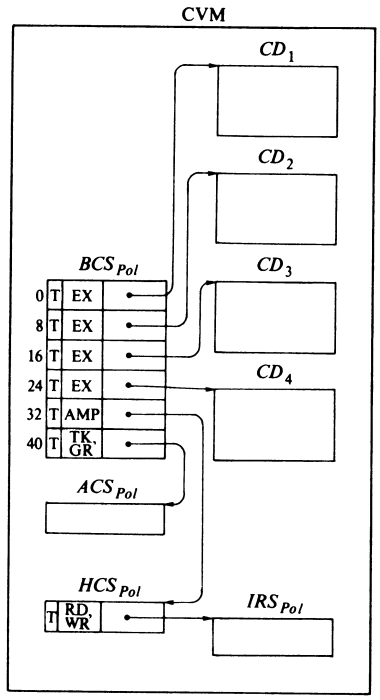

Figure 10. Memory configuration for an efficient implementation of the abstract object *Pol*.

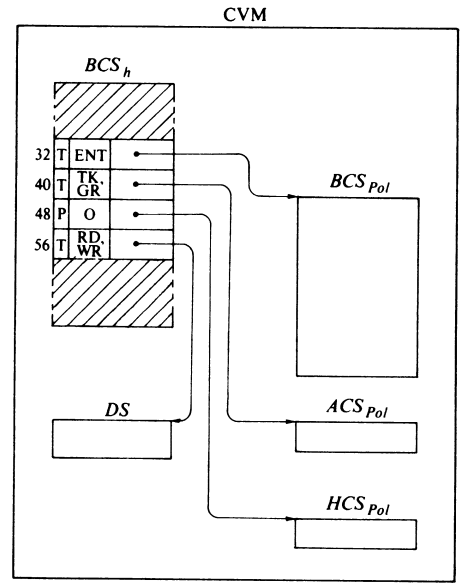

Figure 11. Structure of the base capability segment of a subject able to operate on the abstract object *Pol* implemented as in Fig. 10.

is now reduced to 15 bits. We will refer, as a working example, to the object *Pol*, of the type *Regular_polygon* introduced in Section 4. The implementation of *Pol* uses, besides BCS_{Pol}, ACS_{Pol}, CD_1, CD_2, CD_3, CD_4 and IRS_{Pol}, also a further capability segment, called hidden capability segment HCS_{Pol} (Fig. 10). The BCS_{Pol} structure is slightly modified with respect to the one shown in Section 4. It no longer contains a true-capability for the segment IRS_{Pol}, but a true-capability for HCS_{Pol}, with the access right AMPLIFY: HCS_{Pol} in its turn contains a true-capability for IRS_{Pol} with the access rights READ and WRITE. Moreover, ACS_{Pol} is widened to contain two extended capabilities.

The structure of the BCS_h relative to the subject $s_{i,h}$ able to operate on *Pol*, is also slightly modified: in fact BCS_h now contains, besides the true-capabilities for BCS_{Pol}, ACS_{Pol} and the parameter data segment *DS*, also a pseudo-capability for HCS_{Pol} with an offset $F_2 = 0$. If $s_{i,h}$ wants to perform the operation *Init* on the object *Pol*, it must first store the actual values of the parameters (i.e. the quantities *edge_number* and *edge_size*) into the parameter data segment *DS*. Then it can execute the following instructions, that assume the memory situation shown in Fig. 11:

```
Loadcap                  CR1, 40, CR2
Transfer   MSK WRITE, CR1, 56, CR2, 0
Transfer                 CR1, 48, CR2, 8
Loadcap                  CR1, 32, CR3
Enter                    CR3,  0
```

The first three instructions transfer from BCS_h into ACS_{Pol} the true-capability for the parameter data segment *DS*, with the access right READ, and the pseudo-capability for the entry 0 of HCS_{Pol}. The two remaining instructions perform a domain switch, thereby entering the subject $s_{i,Pol}$, which executes the instructions relative to the *Init* operation. These instructions are the following:

```
Loadcap     CR1, 40, CR2
Loadcap     CR2,  0, CR3
Loadcap     CR2,  8, CR4
Loadcap     CR1, 32, CR5
Amplify     CR5, CR4
Move        CR3,  0, CR4, 0
Movelong    CR3,  2, CR4, 2
Reenter
```

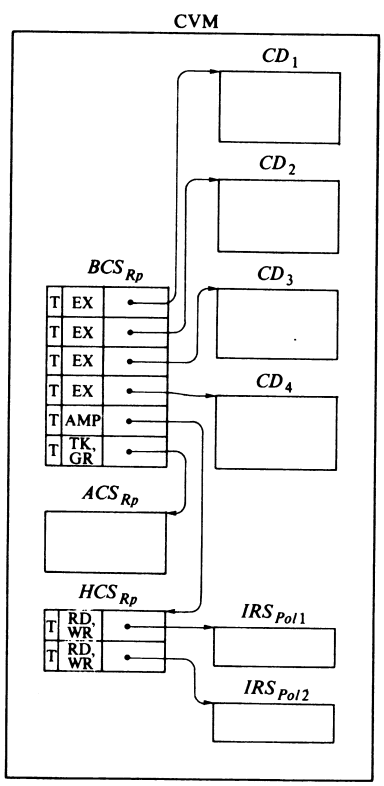

Figure 12. Memory configuration for an efficient implementation of two abstract objects *Pol1* and *Pol2*.

The first instruction loads the true-capability for ACS_{Pol} into CR_2. The second instruction loads the true-capability for the parameter data segment DS, from ACS_{Pol} into CR_3. The third instruction loads the pseudo-capability for the entry 0 of HCS_{Pol}, from ACS_{Pol} into CR_4. The fourth instruction loads the true-capability for IRS_{Pol}, from HCS_{Pol} into CR_4. The *Move* and *Movelong* instructions move the parameter actual values, from DS into IRS_{Pol}: these two instructions correspond to the body of the function *Init*. Finally, the last instruction re-enters subject $s_{i,h}$.

Let us now show a possible implementation of several abstract objects of the same type by referring, as a working example, to the implementation of m objects $Pol_1, Pol_2, \ldots, Pol_m$, of the type *Regular_polygon*. Only the internal representation segments must be replicated m times, so that we have a base capability segment, four code segments CD_1, CD_2, CD_3, CD_4, an auxiliary capability segment ACS_{Rp}, m data segments IRS_{Pol_1}, $IRS_{Pol_2}, \ldots, IRS_{Pol_m}$, and a hidden capability-segment HCS_{Rp}, extended to contain the true-capabilities for the m internal representation segments, with the access rights READ and WRITE. In Fig. 12 the case $m = 2$ is shown.

The subject $s_{i,h}$ is allowed to use the jth object *Polj* if its BSC_h contains, besides a true-capability for BCS_{Rp} with the access right ENTER, a true-capability for ACS_{Rp} with the access rights TAKE and GRANT and a true-capability for DS with the access rights READ and WRITE, also a pseudo-capability for the jth entry of HCS_{Rp}.

The memory waste is very much reduced in this approach, as shown by the following consideration. The implementation of the generical object *Polj* requires 6 bytes for IRS_{Polj} and 8 bytes for the jth true-capability in HCS_{Rp}: in such a way the memory overhead ov_j has now decreased to be about 0.4. Moreover, this approach leads to a noticeable diminution of the number of the required memory segments, so reducing memory fragmentation and simplifying memory management.

7. CONCLUSIONS

An efficient run-time support to the implementation of abstract objects has been given, which uses a capability environment. The efficiency arises from a proper generalization of the classical capability concept. This generalization consists in the introduction of pseudo-capabilities and of a mechanism for amplifying access rights.

REFERENCES

1. B. H. Liskov and S. N. Zilles, Specification techniques for data abstractions. *IEEE Trans. Software Engineering*, March, 7–19 (1975).
2. J. Guttag, Abstract data types and the development of data structures. *Comm. ACM*, June, 396–404 (1977).
3. G. S. Graham and P. J. Denning, Protection—principles and practice. *Proc. AFIPS 1972*, SJCC, 417–429 (1972).
4. R. S. Fabry, Capability-based addressing. *Comm. ACM*, July, 403–412 (1974).
5. J. H. Saltzer and M. D. Schroeder, The protection of information in computer systems. *Proc. IEEE*, Sept., 1278–1308 (1975).
6. P. Corsini, G. Frosini and L. Lopriore, Protection structures in computer systems. *Informatica*, January, 83–102 (1982).
7. P. Corsini, G. Frosini, F. Grandoni and L. Lopriore, Capability based addressing: an overview. *Proc. AICA Annual Conference '80*, AICA, Oct., 164–176 (1980).
8. D. C. Cosserat, A data model based on the capabilities protection mechanism. *Proc. Int. Workshop on Protection in Operating Systems*, IRIA, Aug., 35–53 (1974).

Received April 1983