DYNAMIC RESOURCE MANAGEMENT IN A LANGUAGE FOR REAL TIME PROGRAMMING

P. Ancilotti – Istituto di Elaborazione della Informazione del C.N.R. – Pisa
M. Boari – Istituto di Automatica. Facoltà di Ingegneria, Università di Bologna
N. Lijtmaer – Istituto di Elaborazione della Informazione del C.N.R. – Pisa

Abstract

Reliability and efficiency of programs are main goals of each high level language for concurrent programming and more specifically for real time programming.

Keeping these goals in mind, any extension of the Concurrent Pascal language is proposed.

A mechanism is introduced through which processes can get capabilities to accede reusable system resources. Capabilities can be assigned either statically or dynamically.

The proposed mechanism allows compile time checking of access control, and can be incorporated in any *object oriented* language.

1. Introduction

Reliability and understandability of programs have been considerably improved by the introduction of abstract data types in sequential programming and monitors in parallel programming Correctness verification has been also simplified.

Recently some real time oriented languages which implement such concepts have been proposed, as Concurrent Pascal [1, 2] and Modula [3, 4].

The principal aim of these languages is to provide support for designing highly reliable concurrent programs. The syntactic and semantic definition of these two languages makes easier the detection at compilation time of most of time dependent errors and the static control of the accesses to shared data. This is obtained by means of a set of constraints on the utilization of linguistic objects (abstract data types, procedures, etc.).

However these languages have some properties limiting their applicability to real time programming. Moreover such limitations result more stringent than is necessary to obtain the proposed goals. In particular neither Concurrent Pascal nor Modula allow, consistently with their goals, to solve the problem of dynamic allocation of shared resources.

An extension of Concurrent Pascal to solve the problem of dynamic resource allocation has been presented in [5]. That solution, however, is not satisfactory as regard reliability; in fact it does not allow to statically verify the correct sequence of resource request, use and release. Besides, as far as protection is concerned, that solution does not allow different processes to accede to the same resource with different access rights.

A different solution to the problem of dynamic resource allocation is presented in this paper. The goal is to define a new feature allowing to get over the above mentioned drawbacks. The proposed feature may be embedded in any *object oriented* language for real time programming providing abstract data type definitions. Concurrent Pascal has been chosen in this work as a reference language for this purpose.

In the first part of the paper some properties of Concurrent Pascal are critically reviewed having the resource allocation problem in mind; then the goals to be reached are pointed out and the proposed solution is presented.

2. Resource management strategies in Concurrent Pascal.

Concurrent Pascal provides two language notations, monitor and class, in order to assure compilation time control of accesses to system resources. The former is used to represent shared

resources; the latter to represent resources private to the processes. Both the notations are but particular implementations of the abstract data type concept; in fact they define a logical resource by means of a data structure and the meaningful operations on it.

In the case of a monitor, the operations must guarantee a mutually exclusive access to the controlled resource. In the case of a class, the controlled resource cannot be concurrently requested by several processes; this property is obtained by imposing that an instance of the class may appear only as a local variable in monitors and processes and not among their access rights [1, 2]. The scope rules of the language assure in this case that no concurrent activity on the objects of type class is possible. In fact if an object is declared local to a process it represents a private resource to which no other process can accede; if it is declared local to a monitor, only a process at a time – that is the one which accedes to the monitor – can accede to the object.

Using the monitor notation for some kinds of shared resources can strongly affect the system efficiency. This is the case, for instance, of serial I/O devices, that is devices on which a process executes a sequence of reading or writing operations keeping an exclusive access.

If a monitor is used to represent a serial device, then two kinds of monitor procedures must exist:

a) assigning and releasing procedures;

b) device operating procedures.

Each operation on the device therefore requires an access to the monitor, causing a loss of efficiency, since the synchronization imposed by the monitor is more stringent than is necessary.

By adopting the notation introduced in [1], we have the following access graph (fig. 1):

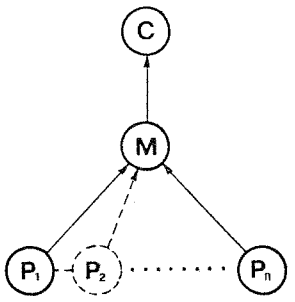

Fig. 1

where $P_1, P_2, \ldots, P_n$ are the processes requesting the resource, M is the monitor and C is a class whose procedures, called by the monitor procedures, execute the input-output operations on the device.

To improve efficiency the phase of resource allocation must be separated from the resource utilization.

The syntactic rules of classes and the resource management strategy adopted in Concurrent Pascal do not allow to solve such a problem in a satisfactory way.

In fact an instance of a class cannot appear among the access rights of a process; therefore no solution, like that shown in fig. 2, is possible. Then a process, to which the monitor has exclusively assigned the resource, can not use it without calling monitor procedures.

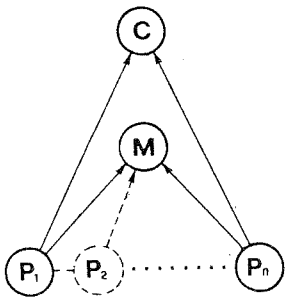

Fig. 2

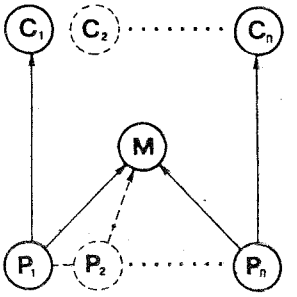

Fig. 3

The only possible solution, shown in fig. 3, is to declare local to each process P_i an instance C_i of the class C; the process P_i, to which the monitor has exclusively assigned the resource, may now directly accede to it through C_i.

Such solution, adopted by P. Brinch Hansen in the implementation of the SOLO operating system, presents some drawbacks limiting its validity. First of all, let us note that this way of using classes is not consistent with the properties of the classes themselves. In fact any instance of a class, local to a process, represents, by definition, a resource to which only that process can accede. In the case of fig. 3, on the contrary, all the instances C_i of the class C are used to represent the same resource, namely the device shared among the processes. Therefore the proposed solution requires that the data structure representing the device is not declared in the class definition. In fact it is contained in the definition of the virtual machine on which the Concurrent Pascal programs run. [7].

Let us consider, for instance, the solution given to a typewriter management problem (the solution is obviously simplified).

```
type Resource = monitor;
        var Free: boolean; Resourcequeue: Fifoqueue;
        procedure entry Request;
                begin
                        if not Free then delay (Resourcequeue);
                        Free: = false;
                end;
        Procedure entry Release;
                begin
                        Free : = true;
                        continue (Resourcequeue);
                end;
        begin Free: = true end;

type Typewriter = class (Device: Iodevice);
        procedure entry Write (Text: Line);
                var Param: Ioparam; C: char;
                begin
                    :
                    io (C, Param, Device);
                    :
                end;
```

```
        procedure entry Read (var Text: Line);
            var Param: Ioparam;  C : char;
            begin
                :
                io (C, Param, Device);
                :
            end;
        begin  end;
type User-A = process (Access: Resource);
        var Unit: Typewriter; Text: Line;
        begin
                :
                Access · Request;
                :
                Unit · Write (Text);
                :
                Unit · Write (Text);
                :
                Access · Release;
                :
            end;
            begin init Unit (Typedevice) end;

type User-B = process (Access: Resource);
            var Unit: Typewriter; Text: Line;
            begin
                :
                Access · Request;
                :
                Unit · Read (Text);
                :
                Unit · Read (Text);
                :
                Access · Release;
                :
            end;
            begin init Unit (Typedevice) end;

                                /* Initial Process */

    var  Typeuse: Resource;
         Writer:   User-A;
         Reader:   User-B
    begin init Typeuse, Writer (Typeuse), Reader (Typeuse) end;
```

Note that some types used in the program are supposed to be previously defined; this is the case, for instance, of the type Fifoqueue which represents an array of variables of the type queue used as a first-in first-out queue.

Each instance of the class Typewriter local to the processes accedes by means of the procedures Read and Write to the same physical device. In fact the parameter Device in the io instruction assumes for all the instances of the class the value Typedevice. The data structure relative to such device is contained in the virtual machine and then it is implicitly common to each instance

of the class. Therefore the validity of the proposed solution depends on the properties of the virtual machine; in fact the solution is valid only for those resources whose data structure is contained in the virtual machine.

To conclude let us note that it is not possible to verify at compile time that a process uses the monitor procedures to request the device before using it and to release it at the end of the I/O operations; then deadlock conditions may arise. Moreover, since the access rights of the processes to the class objects are bound in a static way during the initialization phase, it is possible that more processes simultaneously operate on the same resource.

The same problems are present if we wish to allocate dynamically several resources of the same type, for instance a pool of buffers, I/O devices, etc. Also in this case two solutions are possible. The first one is obtained by using a monitor in which an array of objects of the same type of the resource is declared; some monitor procedures dynamically assign the objects to the processes and others execute the relative operations. As previously said, this solution is too expensive in terms of efficiency, since each access to a single object takes place through the resource managament monitor. Moreover, the parallelism degree is reduced since simultaneous accesses of the processes to different objects are prevented. Finally, in the case the objects are of monitor type, the solution is not proposable since a deadlock may occur. In fact let us consider the graph of fig. 4; if a process is suspended within the monitor R_i, the monitor M becomes blocked and prevents any other process to accede to R_i and to activate the suspended process.

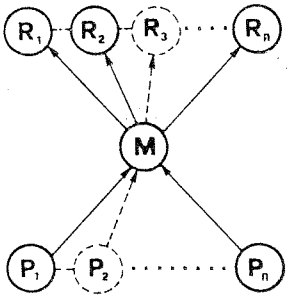

Fig. 4

The second solution is obtained by assigning to the monitor M only the task of providing for resource allocation and by allowing the processes to operate directly on them. In other words the monitor handles a data structure containing information about the allocation state of the resources and provides each process with the information on the resource to be utilized.

Resources of type class are declared as local variables inside each process; resources of type monitor are declared in the main program and the processes can accede to them through their access rights (fig. 5 and fig. 6).

The drawbacks of this solution are the same of the case of fig. 3. Let us note in particular that, if the resource is of type class, it is possible to dynamically allocate only the resources whose data structures are contained in the virtual machine; this means in practice that the solution may be adopted only for I/O devices. Moreover, processes may directly accede, in a non-controlled way, to any resource. Furthermore changes in the number of the resources will concern not only the monitor but also all the processes acceding to them.

If the resource is of type monitor, the constraint of defining the data structures relative to the resources inside the virtual machine is avoided. In fact, in this case, the object declarations are outside the processes.

In conclusion, we may observe that it is never possible, no matter of the technique adopted for the resource management, to obtain in Concurrent Pascal the following goals:

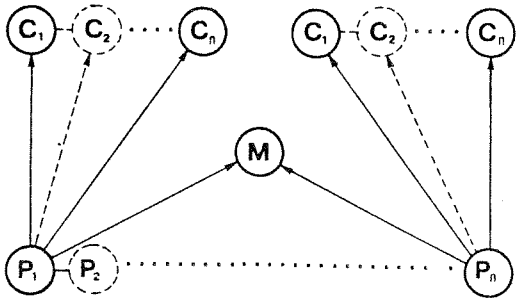

Fig. 5

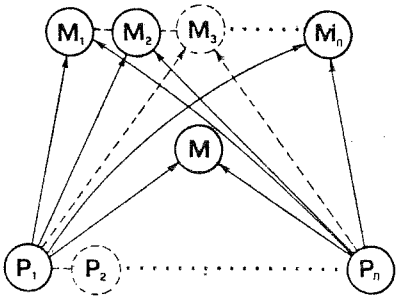

Fig. 6

a) to assure at compile time that the sequence of the operations of resource acquisition, utilization and release is correctly carried out;

b) to control at compile time that a process accedes to a resource only through a certain subset of its procedures. For example in the case of reader and writer processes operating on a same buffer, the compiler is not able to control if the two classes of processes are acceding to the buffer only through appropriate operations.

3. Dynamic resource allocation and access control

3.1 *Goals of the proposal*

Some modifications and extensions to the syntactic and semantic definition of Concurrent Pascal are introduced. The main goals of this proposal are:

1) to guarantee controlled access to system resources at compile time;

2) to allow dynamic allocation of reusable resources to precesses.

As suggested in [5] these goals can be achieved if the language has the following properties:

a) It should allow the definition of multiple, identical instances of an abstract data type, mutually disjoint in the address space, to be managed as a resource pool.

b) Resource instances should be dynamically allocated from the pool to customer processes, but in such a way that a customer process can neither determine the identity of the particular instance that it has been allocated, nor alter its allocation.

c) It should ensure that at most one process can have access to the address space of a resource instance at any time.

d) Synchronization blocking should not take place when two processes simultaneously attempt access to distinct, previously allocated instances of a common type.

c) Conditions 2 and 3 should be guaranteed to hold by the semantic definition of the programming language.

Nevertheless to fully achieve the first goal one more property must be added to the five previous ones, namely:

f) Selective limitations of processes access rights to resources must be allowed according to the operations performed by processes on each resource.

This property must be embedded in the programming language. Scope rules of most of current programming languages enforce accesses from program units to objects in an all-or-nothing way.

A partial solution to this problem is provided by languages that permit the definition of abstract data types (CLU [8], Alphard [9], Concurrent Pascal [1], etc.). In fact, in this case, objects local to an abstract data type can be manipulated only by the procedures exported from the type. The names of such procedures may be interpreted as access rights to the objects of that type. In the following, as far as Concurrent Pascal is concerned, access rights to operate on the objects of an abstract data type will be represented by names of its **entry** procedures.

However, in those languages it is impossible to assign to a program component only a subset of access rights associated with an abstract data type.

Some other languages (Euclid [10], Modula [3], etc.) provide a mechanism to define selectively:

a) objects, local to a program unit, whose names can be exported, i.e. made known outside;

b) objects, not local to a program unit, whose names can be imported, i.e. made known inside.

Such a mechanism can be used to select, for each program unit, access rights to objects.

A mechanism, modeled after the capability protection mechanisms implemented in some operating systems, has been proposed by A. Jones and B. Liskov [11] for the static check of access rights. This approach will be adopted in the following.

3.2 *System types, system components, system variables*

Three types of abstract data are allowed in Concurrent Pascal: Monitors, classes and processes.

The objects of the first two types are passive while processes are the active components. In order to provide a mechanism for the dynamic resource allocation, another abstract data type, whose instances are passive objects, is proposed in this paper.

The syntactic and semantic definitions of this type, called manager, will be given in the following.

According to the Concurrent Pascal notation, abstract data types will be called *system types*, while *system components* will denote instances of system types.

In a similar way, each variable identifying a system component will be called *system variable.*

The distinction between variables and components is a basic language property for implementing both the dynamic resource allocation and the static access control. Such distinction is typical of any object-oriented language [11].

The declaration of a system variable, as well as of any other variable, implies the declaration of the type of the component identifiable by such variable (system type). A subset of the access rights dependent on the type is also declared together with a system variable. The pair (type – subset of access rights) will be called *qualified type.*

The syntax of a variable declaration is:

$$\textbf{var}\ v: t\ \{ x_1, x_2, \ldots\ldots, x_n \}$$

where: $t\ \{ x_1, x_2, \ldots\ldots x_n \}$ is the qualified type of the variable v, while $x_1, x_2, \ldots\ldots, x_n$ are the access rights associated with v. The semantics of such a declaration specifies that a system component of type t is accessible through the variable v; $x_1, x_2, \ldots\ldots, x_n$ are the only procedural accesses available to refer or alter that component through v. The compiler can detect any attempt to per-

form accesses not declared in the qualified type.

The specification of access rights may not appear in the declaration of a system variable:

var v : t

in this case the entire set of access rights dependent on the type t will be considered associated by defualt with the variable v.

In this paper the concept of system variable does not correspond to the traditional concept of variable as an object containing a value that can be modified by means of assignements. A system variable can be conceived rather as a pointer bound to a particular object (system component) containing the value.

Rules to bound system variables to system components are different from assignement rules involving traditional variables. In fact the former allow sharing of system components while the latter implie copying values into the objects. This notions of binding corresponds to assignement involving variables of type pointer. A system variable is equivalent to the concept of capability typical of operating systems.

Two different system variables can be bound to the same system component with different access rights; using one variable rather that the other changes the way the component can be manipulated.

The assignement between system variables:

a : = b

implies the binding of the variable a to the same component referenced by the variable b; in other words it means the creation of a new access path to that component. Such a binding is correct only if the set of access rights associated with the variable a is a subset of the access rights associated with the variable, b furthermore the types of variables a and b must coincide. Then new rights can not be obtained from the new path to the component. The correctness of an assignement between system variables can be checked at compile time.

3.3 *Static and dynamic assignement of access rights*

A system component x of type X (for instance a process) can refer and use a componenf of type Y only if x owns a capability, that is a system variable, bound to that component.

There are two methods to provide capabilities to system components: a static and a dynamic method.

Static method: A capability for acceding a component of type Y can be statically assigned to a component x in two different ways:

i) In the case of a private component of x it is sufficient to declare a system variable y : Y { y_1, y_2, ..., y_n } local to the system type X. Owing to the declaration of the system component x the compiler allocates, among other things, the system variable y in the name space of x. During the initialization of x, its local variables, included the variable y, are initialized. That implies the creation of a component of type Y and the binding of the variable y to that component.

ii) In the case of a shared component, let y: Y {y_1, y_2, ..., y_n} be a system variable declared in the main program and bound to the shared component of type Y. Also in this case both the creation of that component and the binding of y to it occur during the initialization of y. In order to statically assign to a component x of type X a capability for the component referred by y, it is sufficient to associate to the system type X a formal parameter of type Y, for instance:

type X = ... (z : Y {z_1, z_2, ..., z_m });

and to declare y as the actual parameter during the initialization of x: init x(y)

In this way, a system variable z: Y $\{z_1, z_2, ..., z_m\}$ is allocated in the name space of x, and during the initialization of x, the variable z is bound to the component referred by y. Obviously the compiler can check whether $\{z_1, ... z_m\}$ is a subset of $\{y_1, y_2, ..., y_n\}$, that is the correctness of this binding.

Dynamic method. This method is used only for dynamic assignement of capabilities to active components (processes).

In order to dynamically assign to a process x of type X the capability for a component of type Y, the system variable:

$$y: Y \{y_1, y_2, ..., y_n\} \textbf{ dummy}$$

must be declared local to the type X. In this way a system variable y, whose qualified type is Y $\{y_1, ..., y_n\}$, is created in the name space of any component x of type X. During the initialization of x, all ist local system variables, identified by the key word **dummy**, are initialized to **nil**, that is they are not bound to a particular component.

In order to guarantee the protection and the integrity of data, assignements between system variables (that is new bindings of variables to components) can appear only in well defined components, identified as components of type **manager**, with the following restrictions:

a) Each manager can modify only bindings between variables and components of a particular type.

b) Only a **dummy** variable can appear as a left member of an assignement between system variables. In this way, access rights statically assigned can not be modified.

3.4 *A new type: manager*

The dynamic metod to assign capabilities needs the introduction of a new system type: the manager. Components of this type can both bind a **dummy** variable y: Y { ..... } to a specific component of type Y – capability assignement – and assign the value **nil** to the **dummy** variable – capability reclaim –. Therefore, each component of type manager can allocate capabilities to processes in order to handle the dynamic allocation of resources of a well defined type. This type may be either a class or a monitor.

The syntax of the manager definition is similar to that of a monitor or a class definition. The main difference resides in the form of the heading. In fact, together with the identifiers of the defined system type, it is necessary to give also the identifier of the system type of the resources (components) handled by the manager:

```
type < identifier > = manager of < type identifier > (formal parameters);
     < local declarations > ;
                :
     < entry procedures > ;
                :
     < local procedures > ;
begin< initialization > end ;
```

In order to illustrate the use of the manager construct and the dynamic assignment of capabilities, the program for the typewriter management, presented in the paragraph 2, will be modified.

```
type Typewriter = class (Device: Iodevice);
        procedure entry Write (Text: Line);
            begin ..... end;
        procedure entry Read (var Text: Line);
            begin ..... end;
        begin end

type Resource = manager of Typewriter;
        var Terminal : Typewriter;
          Free: boolean;
          Resorcequeue: Fifoqueue;
        procedure entry Request (var x: Typewriter dummy);
          begin
              if not Free then delay (Resourcequeue);
              Free: = false;
              x: = Terminal;
          end;
        procedure entry Release (var  x:  Typewriter dummy);
          begin
              x: = nil;
              Free: = true;
              continue (Resourcequeue);
          end;
        begin Free: = true; init Terminal (Typedevice); end

type User-A = process (Access: Resource);
        var Unit: Typewriter {Read }dummy;
              Text: Line;
        begin
                  :
            Access · Request (Unit);
                  :
            Unit · Read (Text);
                  :
            Unit · Read (Text);
                  :
            Access · Release (Unit);
                  :
        end;

type User-B = process (Access: Resource);
        var Unit : Typewriter {Write }dummy;
              Text : Line;
        begin
                  :
            Access · Request (Unit);
                  :
            Unit · Write (Text);
                  :
            Unit · Write (Text);
                  :
            Access · Release (Unit);
                  :
        end;
```

```
/* Initial Process */
var Allocator: Resource;
    Reader: User-A;
    Writer: User-B;
begin init Allocator, Reader (Allocator), Writer (Allocator); end;
```

Let us consider the representation of the device as a component of name Terminal of the class Typewriter local to the component Allocator. This component is an instance of the manager Resource.

Both processes Reader and Writer have a capability to accede the component Allocator. This capability was statically assigned to them through a parameter. Furthermore the **dummy** variable Unit of type Typewriter is declared local to both processes. By means of this variable that unitially is not bound to any system component, each process can request access to the device calling the Request procedure of the manager. If the resource is free, this procedure binds the variable Unit to the component Terminal handled by the manager. Then, the process could accede the device by means of the variable Unit. Of course the only operation a process could execute on the device is specified by the access right (Read or Write) associated with its own variable Unit.

When the process ends its work with the device, the capability is released. In fact, by calling the manager procedure Release (Unit), the value **nil** is assigned to the variable Unit.

Then the access graph may be that of fig. 7 where the two dotted arrows represent the capabilities to accede the device. Since these capabilities are not assigned statically to the two processes, they could be requested and released dynamically.

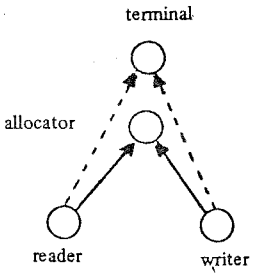

Fig. 7

Let us resume the most important characteristics of the proposed method for the dynamic assignement of capabilities and resource management:

a) A declared **manager** of resources of type Y is able to modify the bindings among variables and components of type Y. Then the manager must have the capabilities to accede a certain number n of components of type Y. . For this reason, n system variables of type Y are declared local to the manager.

b) Two **entry procedures**, each of them with a **dummy** variable of type Y as a formal parameter, are declared local to the manager that handles resources of type Y. The first procedure allocates one of the components, handled by the manager, and therefore in that procedure one of the system variables of type Y is assigned to the formal parameter. The second procedure deallocates the component by assigning the value **nil** to the formal parameter.

c) Like monitor procedures, manager procedures are mutually exclusive. In fact, several processes may try to accede these procedures concurrently.

d) If the declaration:

$$\textbf{var } y: \{y_1, \ldots, y_n\} \textbf{ dummy}$$

is part of the local declarations of a process then a system variable $z = Z\{\ldots\}$, whose type Z corresponds to a manager handling components of type Y, must appear among the formal parameters of that process. The process requests to the manager z the capability to accede a component of type Y in order to operate on it. This request is done by calling the allocation procedure of the manager and passing the actual parameter y. When the capability is no longer necessary it must be released by calling the deallocation procedure of the manager.

Since the components handled by a manager are all of the same type, they cannot accede mutually to each other. Furthermore, since the type definition of the components handled by the manager is external to the manager itself, these components cannot accede the local variables of the manager. Therefore the scope-rules of the language determine a protection hierarchy in the manager.

3.5 *Static access control mechanism*

Some drawbacks of the Concurrent Pascal have been pointed out in Section ?. The proposed mechanism for granting capabilities allows to remove such drawbacks, encreasing th reliability and efficiency of programs. In fact, the declaration of access rights for each system variable enables the compiler to check that a process refers to resources in a proper way, i.e. through the only procedures that are meaningful to its application domain. Moreover, the capability passing mechanism allows dynamic resource assignment. However, such mechanism does not allow to check whether a **dummy** variable was correctly bound to a system component before being used to accede that component. Similary, it is impossible to check that a capability is released after being used. In other words, it is impossible to check statically whether the sequence of calls to manager procedures is correct.

The above drawbacks can be removed by adding a pair of constraints in the use of the capability passing mechanism:

a) Each manager has exactly two **entry procedures**: A procedure **Request**, through which capabilities are assigned to accede a system component handled by the manager; A procedure **Release** through which capabilities are released.

b) If a process dynamically requests the capability to accede a system component, then each sequence of accesses to that component must be part of a procedure body. A **dummy** variable, through which the process accedes the component, must be declared local to the procedure.

In this way, for each **dummy** variable, local to a procedure, the compiler can place calls to the appropriate Request and Release, respectively before and after the procedure body. Obviously, Request and Release belong to a manager handling components of the same type of the declared **dummy** variable.

If in the body of a procedure there are more declarations of **dummy** variables, the compiler generates calls to the corresponding Request and Release procedures in the same order in which variables were declared.

Let the declaration:

$$\textbf{var } y: Y\{y_1, \ldots, y_n\} \textbf{ dummy}$$

belong to a local procedure of a process. If more instances, $m_1, m_2, \ldots, m_k$, of managers handling components of type Y are accessible to the process, then the name m_i of the manager instance, to which requests will be directed, must be passed as a parameter to the procedure.

Finally, since resources may be required in a nested way, nesting of procedure declarations must be allowed.

Note that this facility is not present in Concurrent Pascal. In fact, if there are two nested procedures A and B with the declarations: var y:Y $\{y_1, \ldots, y_n\}$ **dummy** local to A, and var z:Z $\{z_1, \ldots, z_m\}$ **dummy** local to B, the compiler inserts, automatically, calls to the appropriate Request and Release procedures. In particular, calls to the procedures of a manager of Y are placed before and after the procedure body of A, and calls to the procedures of a manager of Z are placed before and after the procedure body of B. In this way, when a process calls the procedure A, it obtains a capability to accede a resource of type Y, and when, during the execution of A, the procedure B is called, it obtains the capability to accede a resource of type Z. Capabilities are released at the end of procedure executions.

With this method, access control is guaranteed by the compiler for both the static and the dynamic methods of granting capabilities.

The two constraints introduced before to guarantee the static control of the correct sequence of manager procedures calls, are not too much restrictive and do not limitate the generality of the proposed method.

4. Conclusions

Reliability and efficiency of programs are main goals of each high level language for concurrent programming and more specifically for real time programming. Keeping these goals in mind, a new mechanism, by which processes can get capabilities to accede reusable system resources, has been introduced. This mechanism allows compile time checking of access rights. Moreover, capability assignements may be done statically or dynamically.

A significant property of this mechanism is that it can be embedded in any *object oriented* language. In particular, an extension of the Concurrent Pascal language has been proposed.

Both, system variables and manager concepts have been introduced.

A system variable identifies a system component and the associated set of access rights which are the only operations that can be performed on that component through that variable. There is an immediate analogy between system variables in high level languages and capabilities of protection mechanisms in operating systems.

The manager is a new system type with characteristics very closed to that of the monitor type. The main difference is that managers are able to perform dynamic bindings between system variables and system components and then to allocate dynamically a set of resources to processes.

A first approach to managers has been introduced by Sitberschatz, Kieburtz and Bernstein [5]. Nevertheless this approach presents some inconveniences, namely:

i) Sequences of calls to manager procedures can not be statically checked. That is, no guarantee can be provided for a process to request a resource before using it and to release this resource after its use. Deadlock conditions can arise due to a wrong sequence of procedure calls.

ii) No control can be done on access rights. That is, selective limitations of process access rights to resources are not allowed.

iii) Only one instance of a manager may be defined. This introduces a difference between the treatement of managers and other system types.

iv) Each process uses the same name to identify both a component of type manager and a resource handled by the manager. Then no distinction can be done between static and dynamic assignment of capabilities to a process.

The proposal presented in this paper removes the above inconveniences.

REFERENCES

[1] P. Brinch Hansen: – The programming language Concurrent Pascal – . **IEEE Transactions on Software Engineering**, June 1975.

[2] P. Brinch Hensen: – **Concurrent Pascal Report** – . Information Science. California Institute of Technology, June 1975.

[3] N. Wirth: – Modula: a language for modular Multigramming – **Software-Practice and Experience** – January – February 1977.

[4] N. Wirth: – Design and Implementation of Modula – **Software–Practice and Experience**, January-February, 1977.

[5] A. Silberschatz, R.B. Kieburtz, A. Bernstein: – Extending Concurrent Pascal to allow dynamic resource management – **Proceedings of the 2nd international conference on Software Engeneering** October 1976 – S. Francisco California.

[6] P. Brinch Hansen: – The Solo Operating System – **Software-Practice and Experience** – April-June 1976.

[7] P. Brinch Hansen: – **Concurrent Pascal machine** – Information Science . California Institute of Technology, January 1976.

[8] B.H. Liskov: – **An introduction to CLU** – Massachussets Institute of Technology, Laboratory for Computer Science, February 1976.

[9] W.A. Wulf: – **Alphard: Towards a language to support structured programs** – Department of Computer Science, Carnegie-Mellon University, April 1974.

[10] Report on the programming language EUCLID, **ACM, Sigplan Notices**, Vol. XII, n. 2, Feb. 1977.

[11] A. Jones, B. Liskov: – A language extension for controlling access to shared data. **IEEE Transactions on Software Engineering**, December 1976.