PROTECTION IN LANGUAGES FOR REAL TIME PROGRAMMING

P. Ancilotti
Istituto di Elaborazione della
Informazione,CNR, via S. Maria
Pisa - Italy

M. Boari
Istituto di Automatica, Facoltà
di Ingegneria, Viale Risorgimento
Bologna - Italy

N. Lijtmaer
Istituto di Elaborazione della
Informazione,CNR, via S. Maria
Pisa - Italy

ABSTRACT

A protection mechanism which may be embedded in an object oriented language for real time programming permitting definition of abstract data types, is proposed in this paper.

This mechanism provides support for designing highly reliable concurrent programs; in fact it allows the detection at compile time of a large class of time dependent errors. To verify the versatility of the proposed mechanism it is firstly characterized abstractly in terms of a protection model; then some linguistic features enforcing protection are defined.

1. Introduction

The role of a protection mechanism in a programmed system is to prevent processes from acceding to the objects defined in the system in an unauthorized or undesirable way; a protection mechanism must therefore guarantee that each process may both accede only those objects for which it has legitimate rights and perform only meaningful accesses to those objects.

Protection mechanisms are usually supported by operating systems. Recently, however, the opportunity to incorporate protection facilities in programming languages has been recognized. In fact software reliability may be considerably enhanced if access control restrictions are expressed directly in the language and enforced by the compiler of that language. Thus, access control errors can be captured at compile time so that programs may be written to be well-behaved with respect to access control restrictions [Wul 74,76; Jon 76] .

A mechanism suitable for incorporation in object oriented languages was presented in [Jon 76]. Such a proposal is based on capability protection mechanisms provided by some operating systems.

Following this approach, a protection mechanism, which may be embedded in an object oriented language for real time programming providing abstract data type definition, is proposed in this paper. This mechanism also provides support for designing highly reliable concurrent programs; in fact it allows the detection at compile time of a large class of time dependent errors.

Unlike some real time languages recently proposed [Bri 75, Wir 77] , the possibility that objects may be dynamically allocated to processes is considered. The protection mechanism must guarantee, also in this case, a compile time checking of access control. To verify the versatility of the proposed mechanism it is firstly characterized abstractly in terms of a protection model; then some linguistic features enforcing protection are defined.

2. PROTECTION: A MODEL

The versatility of protection mechanisms can be abstractly characterized in terms of a *protection model*. A protection model sees the *system* as a collection of components. System components may be partitioned in two disjoint subsets, namely, *subjects*, that is processes and *objects*, that is resources. Furthermore, a protection model defines the access rights of each subject to each object. Following Lampson's proposal on 1971, a protection model can be represented in the form of a *protection matrix*. Subjects are associated with rows of the matrix, while objects are associated with columns [Lam 71]. Given a pair subject-object, the corresponding entry of the matrix defines the set, possibly empty, of access rights that the subject has to the object. A *protection domain* defines the set of access rights that one subject has to the objects of the system.

A first aspect of the protection mechanism is related to the enforcement of protection. It means that , at any time, a subject can exercise only those access rights that belong to its domain. In terms of the protection model a subject can accede an object if and only if the matrix entry, corresponding to the pair subject-object, is not empty. Furthermore the subject can exercise on that object only those access rights specified in the matrix entry. This aspect of a protection mechanism is called the *enforcement rule* [Jon 73].

As far as system reliability is concerned it should be convenient to maintain protection domains as small as possible in order to restrict the number of objects that a subject can use to those significant to the performed activity. This concept has been called the "principle of least privilege" [Lin 76].

While a small program operates on a small number of objects, a larger one normally needs to operate on a great number of them. To conciliate this characteristic with the principle of least privilege, a large program must run in many different protection domains and then it must be able to switch protection domains during execution. In this way the protection domain in which a program runs, can be held time to time as small as possible.

Domain switching represents the second aspect of a protection mechanism and it is called the *domain binding rule* [Jon 73].

From the model point of view since a subject may run in more than one protection domain, it cannot be represented by only one row of the protection matrix. Thus, rows coincide with domain and a subject must be specified by a pair process-domain.

Fig. 2.1 shows that each process P running in the protection domain D_i can exercise access rights α and β on the object R_j while object R_k is not accessible.

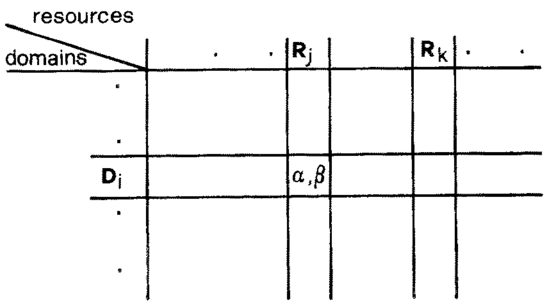

Fig. 2.1

When a process needs to change its current domain to acquire or release a limited number of rights, but one wants to avoid the complete domain switching, the protection mechanism must be able to transfer rights into and out of domains. This is the third and last aspect to be covered by the protection mechanism and it is called the *transfer rights rule* [Jon 73].

In terms of the model we must then allow changes in the protection matrix. This aspect, related to the assignment of access rights to domains, is strongly connected with the problem of resource allocation in the operating system sense.

Let us now characterize objects in relation to the way in which access rights are assigned to domains. If access rights to use an object are allocated to domains during the initialization phase and these rights are no longer modifiable, then the object is statically allocated. Otherwise, if a domain may acquire or release rights to accede an object, this object is dynamically allocated.

If access rights to accede an object belong to only one domain this object is said a dedicated resource. In particular a dedicated resource statically allocated to a domain represents a private resource of that domain. If access rights belong to more than one domain the object is called a *shared resource*.

In order to avoid time dependent errors, concurrent accesses to the same objects must be generally avoided. For this purpose we will suppose that at most one process at a time can run in a protection domain. Furthermore shared resources will require exclusive access and synchronization among processes competing for their use.

3. LANGUAGE ENFORCED PROTECTION

The main goal of this paper is to show how a protection mechanism modeled as described can be embedded in an object oriented language for real time applications. Concurrent Pascal has been chosen as the reference language [Bri 75].

From a linguistic point of view, an object or resource is defined as a uniquely distinguishable structured encoding of information. An *access* to a resource is an algorithm to reference the resource in order to transform it or to extract information encoded within it. The access algorithm depends upon the internal representation of those resources to which the algorithm can be applied. Thus the set of all resources is partitioned into equivalence classes, each called a *type*. The type of a resource determines the accesses that can potentially be applied to that resource.

Any description of a resource type is completely specified by the definitions of applicable accesses. Knowledge of the internal representation of resources need not be available outside the access algorithms [Jon 73].

Then each resource type may be conceived as an *abstract data type* [Lis 75].

Following Concurrent Pascal notation, a dedicated resource type is defined as a *class:*

```
type dedicated-resource-type = class (.,.,.,.);
     begin ....... end;
```

Each instance of a class represents a dedicated resource to which only the accesses specified by *entry* procedures can be applied.

Analogously, a shared resource type may be defined as a *monitor:*

```
type shared-resource-type = monitor (.,.,.,.);
     begin ....... end;
```

In this case the entry procedures are mutually exclusive, so that at most one process can be active in a monitor, at any time. Furthermore, queue variables are allowed as part of the shared data base of a monitor in order to permit the specification of the coordination among processes competing for this resource [Bri 75, Hoa 74].

Note that instances of abstract data types correspond to columns of the protection matrix.

Scope rules of the language establish that local objects of abstract data types are accessible only by the procedures of those types. Then each instance of an abstract data type identifies a particular domain represented by a row in the protection matrix.

Each protection domain can be conceived as a capability list, where capability is a token that identifies an object and a set of access rights; possession of the token confers those access rights for that object. From the model point of view a capability is represented by an entry of the protection matrix.

The syntax of a capability declaration is:

$$\mathit{var}\ \mathrm{v}\colon T\ \{\alpha,\beta,\ldots\}$$

The semantics of such a declaration specifies that a resource of type T is accessible through the capability v; α, β,.. are the only procedural accesses available to refer or alter that resource through v. They correspond to a subset of procedure entries of type T.

The concept of capability does not correspond to the traditional concept of variable as an object containing a value that can be modified by means of assignments. A capability can be conceived rather as a pointer that can be bound to a particular object containing the value [Jon 76].

Rules to bind capabilities to objects are different from assignment rules involving traditional variables. In fact, the first ones correspond to assignment rules involving variables of type pointer. A capability, bound to an object, can be conceived as an access path for the object. Both, the type of the object and the type of the capability must be the same, in order to guarantee a correct binding.

For simplicity we will use the notation resource or object x of type X instead of resource or object referred by the capability x of type X.

A process can refer and use a resource if and only if it executes in a domain owning an access right to that resource, that is a capability bound to the resource.

As stated in the previous paragraph assignment rules of access rights to domains may be either static or dynamic, and then static and dynamic binding of capabilities to resources must be provided.

Nevertheless while resources can be dynamically allocated they are created statically and will exist forever after the system initialization; that is no dynamic resource creation is considered in this paper.

First of all the static assignment of access rights to domains, that is the static capability binding, is considered. There are two diffe rent ways to statically bind a capability to a resource:the first one corresponds to the allocation of a dedicated resource, while the second one allows resource sharing.

A capability y of type Y is created in any domain containing the declaration:

var y: Y $\{\alpha,\beta,\ldots\}$

that is, in any instance of an abstract data type containing the previous declaration.

A resource of type Y, that is an instance of an abstract data type Y, is created during the initialization of y:

init y .

The init statement allocates space for private variables of the resource,initializes them and binds the capability y to the resource [Bri 75].

Thus, given a domain associated with an instance x of type X, in order to statically allocate to x the access rights α, β,... to a private resource of type Y, where Y must be a class, it is sufficient to declare the capability:

var y:Y $\{\alpha,\beta,..\}$

into type X. During the initialization of x its local variables, included the variable y, are allocated in the name space of x and are initialized. That implies the initialization of y, that is the creation of a resource of type Y and the binding of y to this resource.

The static allocation of shared resources involves the definition of a resource for which more than one, statically declared, access path exists.

The problem now is to allow a set of domains associated with instances $x_1,x_2,\ldots,x_n$ of types $X_1,X_2,\ldots,X_n$ respectively, to share a resource of type Y, where Y must be a monitor. For this purpose the capability:

var y:Y $\{\alpha,\beta,..\}$

is declared in the main program. As stated above, during the initialization of y a resource of type Y is created and y is bound to it.

Then, in order to statically assign to each domain $x_i (i = 1...n)$ the access rights $\alpha_{1_i}, \beta_{1_i}$ ... to the resource y, a formal parameter of type Y with access rights $\alpha_{1_i}, \beta_{1_i}$ ... is associated to the system type X_i, for instance:

$$\textit{type}\ \ X_i = ...(.....,z : Y\ \{\alpha_{1_i},\beta_{1_i},...\}...)$$

and y is declared as the actual parameter during the initialization of x_i:

$$\textit{init}\ x_i\ (..., y, ...).$$

In this way, a capability $z : Y\ \{\alpha_{1_i},\beta_{1_i},...\}$ is allocated in the name space of x_i, and, during the initialization of x_i, z is bound to the resource y.

Note that $\{\alpha_{1_i},\beta_{1_i},...\}$ must be a subset of $\{\alpha, \beta,...\}$ in order to guarantee the correct binding of z to the resource y.

In order to allow dynamic resource allocation, bindings between capabilities and resources must be delayed until execution time. If a resource of type Y is dynamically requested by an instance x of type X it means that x, initially, does not own any access path to accede to the resource. A path must be granted to x only when the resource is needed and then the path must be reclaimed. For this purpose the capability:

$$\textit{var}\ y : Y\ \{\alpha,\beta,..\}\ \textit{empty}$$

is declared in the abstract data type X. In this way the capability y is created in any domain associated with an instance x of type X.

During the initialization of x all its capabilities, identified by the key word *empty*, are not bound to a particular resource. The domain x can acquire the access rights $\alpha,\beta,...$ to use a resource of type Y. For this purpose the capability y must be bound to the resource.

Assignments between capabilities, that is new bindings of capabilities to resources, may appear only in instances of a particular type called *manager* with the following restrictions:

a) Each manager may modify only bindings between capabilities and resources of a particular type.

b) Only an *empty* variable may appear as the left member of an assignment involving capabilities. Then bindings statically declared cannot be modified.

c) The set of access rights associated with the left member of an assignment involving capabilities must be a subset of the access rights associated with the right member [Jon 76].

In this way a manager can both bind an empty capability $y : Y \{\alpha, \beta, \ldots\}$ to a specific resource of type Y - resource allocation - and assign the value *nil* to the *empty* capability - resource reclaim. In other words a manager handles the dynamic allocation of resources of a particular type. This type can be either a class or a monitor.

The syntax of a manager definition is similar to that of a monitor or a class definition [Sil 77, Anc 77a, Anc 77b]. The main difference resides in the form of the heading. In fact, together with the identifier of the defined type, the identifier of the resource type handled by the manager is also required.

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

Let us resume the most important characteristics of the proposed method for dynamic binding of capabilities to resources:

i) A declared *manager* of resources of type Y is able to modify bindings among capabilities and resources of type Y. Then the manager must have access rights to a certain number n of resources of type Y. For this reason n capabilities of type Y are declared into the body of the manager. In other words any instance of a manager represents a domain with access rights to a certain number of resources of type Y. By means of capabilities assignments a manager can transfer its access rights to other domains. Following the previous restriction c) a manager is able to transfer only its own access rights.

ii) Two *entry* procedures, each of them with an *empty* capability of type Y as a formal parameter, are declared into each manager handling resources of type Y. The first procedure allocates one of the resources handled by the manager and therefore in that procedure one of the capabilities of type Y is assigned to the formal parameter. The second procedure deallocates the resource by assigning the value *nil* to the formal parameter.

iii) If the capability:

var y : Y {α,β,...} *empty*

is declared into a system type X, then X has a formal parameter whose type corresponds to a manager of resources of type Y. Each instance x of type X requires the manager to allocate a resource of type Y in order to operate on it. This request is done by calling the allocation procedure of the manager. When the resource is no longer needed, it must be released by calling the deallocation procedure of the manager. The variable y is passed as an actual parameter to both procedures. The set {α,β,...} must be a subset of the access rights associated with the formal parameters of the allocation procedure of the manager.

In order to guarantee completely controlled accesses to resources at compilation time, the programmer must specify for each dynamically allocated resource R all the program regions in which R is referred. In this way the compiler is able to establish, for each access to a resource R, whether the access rights will belong to the domain in which the requesting process is running.

For this purpose we can use the notation of critical regions introduced by Brinch Hansen [Bri 73] in connection with the mutual exclusion problem. In fact we can assume that references to a dynamically allocated resource R may appear only within structured statements called *allocation regions*. An allocation region is represented by the notation:

region v *do* S;

where v is an *empty* capability. The allocation region associates a statement S with the empty capability v. This notation enables the compiler to check that empty capabilities are used only inside allocation regions and to place a call to the appropriate allocation procedure before the statement S and a call to the deallocation procedure after S. Obviously these procedures must belong to a manager handling resources of the same type of the *empty* capability v.

Let the *empty* capability:

$$\mathit{var}\ \ v : V\ \{\alpha, \beta, \ldots\}\ \mathit{empty}$$

belong to a domain C. If more instances $m_1, m_2, \ldots, m_k$ of manager handling resources of type V are accessible to C, then for each allocation region associated with the capability v, the name m_i of the manager instance,to which requests will be directed, must be passed to the region as a parameter.

$$\mathit{region}\ (m_i)\ v\ \mathit{do}\ S;$$

Since resources may be required in a nested way, nesting of allocation regions must be allowed.

To conclude let us show how the three protection aspects presented in the previous paragraph are implemented by the proposed mechanism.

i) The enforcement rule is implemented by the scope rules of the language.

ii) The domain binding rule is implemented by the calling procedure mechanism of the language; in fact a domain switching takes place every time a procedure entry of an abstract data type is called.

iii) The transfer rights rule is implemented by introducing in the language new features, namely: empty capabilities, managers and allocation regions. In fact the manager allows the dynamic allocation of access rights to those domains in which an empty capability was declared. Furthermore, the allocation region enforces the correct sequence of request use and release of access rights.

CONCLUSION

In this paper we have proposed a general protection mechanism to be embedded in an object oriented language for real time programming.

Some linguistic features have been introduced to allow compile time checking of access rights. Moreover dynamic allocation of access rights to protection domains is permitted.

REFERENCES

[Anc 77a] Ancilotti,P., Boari,M., Lijtmaer,N. - Dynamic resource management in a language for real time programming - *AICA 77* Pisa, Italy, October 1977.

[Anc 77b] Ancilotti,P., Boari,M., Lijtmaer,N. - A mechanism for allocating resources and controlling accesses in languages for real time programming - *Internal Report n.B77-23;* IEI CNR, Pisa, Italy, December 1977.

[Bri 73] Brinch Hansen,P. - *Operating System Principles* - Prentice Hall, 1973.

[Bri 75] Brinch Hansen,P. - The programming language Concurrent Pascal - *IEEE Transac. on Software Engineering*, Vol.SE-1, n. 2, June 1975.

[Hoa 74] Hoare,C.A.R. - Monitors: an operating system structuring concept - *Comm. ACM n. 10*, October 1974.

[Jon 73] Jones,A.K. - Protection in programmed systems - *Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh* - June 73.

[Jon 76] Jones,A.K., Liskov,B.H. - A language extension for controlling access to shared data - *IEEE Transactions on Software Engineering*, vol. SE-2, December 1976.

[Lam 71] Lampson,B.W. - Protection - *Proc.Fifth Annual Princeton Conf. on Information Sciences and Systems*, 1971.

[Lin 76] Linden,T.A. - Operating System structures to support security and reliable software - *ACM Computing Surveys*, Dec. 76.

[Lis 75] Liskov,B., Zilles,S. - Specifications techniques for data abstractions - *IEEE Trans. on Software Engineering*, Vol. 1, n. 1, March 1975.

[Sil 77] Silberschatz,A., Kieburtz,R.B., Bernstein,A. - Extending Concurrent Pascal to allow dynamic resource management. *IEEE Transactions on Software Engineering*, Vol. SE-3, n.3, May 1977.

[Wir 77] Wirth,N. - Modula: a Language for modular multiprogramming - *Software-Practice and Experience*, 1, 1977.

[Wul 74] Wulf,W.A. - Toward a language to support structured programs - *Tech. Report Carnegie-Mellon Univ.,Pittsurgh, Pa.* April 1974.

[Wul 76] Wulf,W.A., London,R.L., Shaw,M. - Abstraction and verification in Alphard: introduction to language and methodology - *Techn. Report, Carnegie-Mellon Univ., Pittsburgh* 1976.