# On the Static Access-Control Mechanism in Concurrent Pascal

**Richard Kieburtz**
Department of Computer Sciences, SUNY at Stony Brook, NY, 11794, USA
**Abraham Silberschatz**
Department of Computer Sciences, The University of Texas, Austin, Texas, 78712, USA

**In Concurrent Pascal, an explicit hierarchy of access rights to abstract variables is stated in the program text and checked by the compiler. This declarative control of static access rights provides a considerable degree of protection against unauthorized use of an abstract variable by an errant program component. However, there are cases in which the mechanism of Concurrent Pascal falls short of enforcing the principle that each program component should have within its name space only those rights of access to variables that it requires. This paper outlines an extension to the static access control mechanism of Concurrent Pascal that can enforce the need-to-know principle.**

## INTRODUCTION

A large software system is commonly configured by linking together a number of previously compiled modules drawn from a library. To specify a system configuration is to tell what instances of modules need to be created and initialized, and what are their linkages to other modules. It is not common practice to specify a system configuration in the same notation used for programming its components.

The programming language Concurrent Pascal was not designed to serve as a systems configuration language,[1] but as we shall see, it comes close to being suitable for the purpose. In this paper we are concerned with its facilities for instantiating and initializing program modules, and with the means it provides to specify linkages. We are also concerned with a protection issue: how to limit the set of system resources that an individual module might access to the smallest set necessary to carry out its task. This form of protection is important because a system that imposes such a restriction upon its components is better able to restrict the scope of damage that may be caused by a malfunctioning component, than is a system without access restrictions.

One of the most noteworthy aspects of Concurrent Pascal is that it supports modularity in the construction of programs. A system programmed in this language is constructed from three kinds of modules called *processes*, *monitors* and *classes*. The inter-module service calls of such a system can be modelled by a directed graph, called an *access-graph*, whose nodes are labelled with the names of system modules. An arc from *A* to *B* indicates that module *A* can call upon module *B* for service. In such a case we say that module *A* has an *access-right* to module *B*. An access-graph G has the properties that: (a) G is static. Once a system has been configured, no new modules nor access-rights can be created or destroyed. (b) G is acyclic. The modules of a system form a hierarchy. These two properties are always true of a system written in Concurrent Pascal; the language structure guarantees them.

Obviously, a module requires access rights to any other module whose procedures it may need to call in carrying out its task. The specification of need for access is not a decision ordinarily left up to the programmer of an individual module, however. It is a system designer's decision. From the point of view of a system designer, it is equally important that the access-rights of each module are restricted as that it is furnished access to those resources it requires. By restricting the access-rights of system modules, it can be assured that any damage which might be done by an errant module will be limited in its scope. This is an essential characteristic of a reliable system. The principle that a module should be allowed to address only those resources which it is said to need has been referred to as the need-to-know principle.[2]

This paper investigates these ideas as they are incorporated in the programming language Concurrent Pascal. We show that the need-to-know principle cannot always be adhered to in writing operating systems in Concurrent Pascal, and we propose a simple modification that will allow this principle to be observed. This proposal is then compared with two other schemes designed to meet similar goals.

## STATIC ACCESS CONTROL

Let us start by briefly reviewing how access-rights are manifested in Concurrent Pascal. A program component (module P) can gain a permanent access-right to an instance *R* of type *T* in either of two ways: (a) The instance *R* can be declared in the local name space of P; or (b) a permanent parameter of type *T* can be declared in the heading of P. During initialization of P, the permanent parameter name is bound to a designated instance of type *T*. The second of these methods allows static binding of access to a globally defined module. This method for manifesting access-rights displays considerable ingenuity and deserves further explanation.

In Concurrent Pascal, each program component that represents an abstract data type (that is, a *class* or a

CCC-0010-4620/82/0025-0017 $02.50

*monitor*) contains a sequential code segment to accomplish initialization of an instance of the type. No operator can be invoked on a module until the module has been initialized. Initialization is not implicit, but must be called for explicitly by executing an *init* statement upon the module, causing its initialization code to be executed.

In an *init* statement, actual parameters are furnished, corresponding in type to the permanent parameters of the program component type being initialized. The permanent parameters are bound to local names of the corresponding actual parameters, and the binding persists throughout the lifetime of the initialized program segment. An actual parameter mentioned in an *init* statement must itself have been previously initialized.

To provide for initialization of global modules (i.e. monitors and processes), all global declarations are included in the textual structure of an enclosing process declaration. This all-enveloping process, called the *initial process* of a Concurrent Pascal program, has no name nor any parameters. Typically, its code body consists only of a sequence of *init* statements, although in principle, it might also give initial values to globally defined variables of simple types. Since all global processes and module instances are declared within the initial process, all are initialized there as well. The only *init* statement which is implicitly executed when control is given to a Concurrent Pascal program is that of the initial process.

Let us illustrate how access rights are obtained in Concurrent Pascal by an example. Consider the system depicted by the access graph of Fig. 1. In this system,

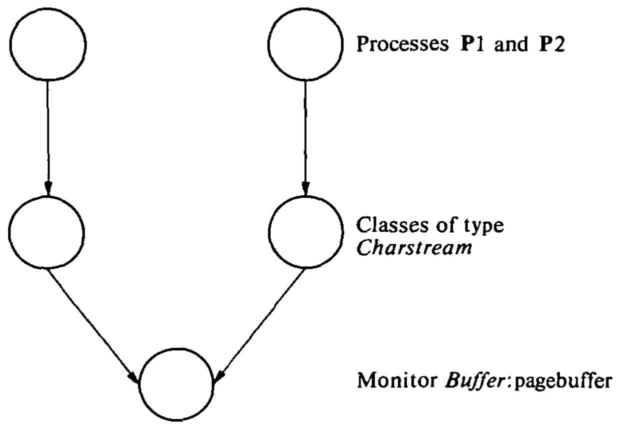

**Figure 1.** An access graph.

processes P1 and P2 communicate with one another via a shared page buffer, which is encapsulated as a monitor to ensure exclusivity of access by each process. However, a process is not given a raw buffer to use for communication; an interface to the buffer is provided by the class type *Charstream*. A program skeleton corresponding to this system is

```
type Pagebuffer = monitor;
        begin . . . end;
    Charstream = class (Buffer: Pagebuffer);
        begin . . . end;
    P = process (Buffer: Pagebuffer):
      var Messagestream: Charstream;
      begin
        init Messagestream (Buffer);
          :
        end;
var  Buffer: Pagebuffer;
     P1, P2: P;
begin (*initial process code*)
     init Buffer;
     init P1 (Buffer);
     init P2 (Buffer)
end.
```

The example above, which is taken from the Solo operating system,[3] is typical of the way that Concurrent Pascal programs are structured. The problem here is that the skeleton program violates the need-to-know principle, in that processes P1 and P2 hold an access right to the monitor, *Buffer*, although they have no actual need to use that monitor directly.

If one were to try to remedy this situation, using Concurrent Pascal as it has been defined, the skeleton program might be rewritten as follows:

```
type Pagebuffer = monitor;
        begin . . . end;
     Charstream = monitor (Buffer: Pagebuffer);
        begin . . . end;
     P = process (Messagestream: Charstream);
        begin . . . end;
var  Buffer: Pagebuffer;
     Instream, Outstream: Charstream
     P1, P2: P;
begin (* initial process code*)
     init Buffer;
     init Instream (Buffer), Outstream (Buffer);
     init P1 (Instream), P2 (Outstream)
end.
```

Here the local variable *Messagestream* that was declared within process type P in the original version has been eliminated. Instead, there has been substituted a pair of globally declared instances of type *Charstream*, each of which has been given a static access right to the monitor *Buffer*. Note that type *Charstream* is now a monitor type, rather than a class type as it had been previously. This is because class instances cannot be bound as permanent parameters to a monitor or a process in Concurrent Pascal.

However, the substitution of a monitor for a class cannot be considered to be a satisfactory solution, since this would burden a system with the needless run-time overhead of monitor entry at each call upon one of its procedures. When a monitor procedure is called, mutual exclusion of access (by customer processes) is enforced by a call upon the system kernel. Upon return from a monitor procedure, the kernel must again be invoked. Thus even though no instance of the type *Charstream* is shared among multiple processes, a substantial loss in efficiency would be implied by the standard monitor implementation.

An equally serious problem with this solution is that it interferes with a reasonable modularization of the system. This is because declaration and initialization of *Instream* and *Outstream* are pushed into the definition of the initial process. The solution given above is artificial, and would never be used in practice. In the following section, we propose another alternative.

## ENVIRONMENT SPECIFICATION

A possible way to satisfy the need-to-know principle is to separate the definition of rights of access from the declaration of a shared instance of an abstract type. The rights of access may then be limited to those program components that have actual need to use a module, while allowing the module name to have a global declaration which affords it the necessary scope. The separation needed is between a type definition and the declaration of instances of that type. Instead of placing type definitions in a global context and allowing instance declarations to be made locally, we wish to restrict type definitions to a local context, but allow global instances to be declared. In order to accomplish this, we shall allow qualified type names to be exported from the context of their definition, but with a restriction of the set of rights (i.e. procedure names applicable to the type) that is exported with the type name.

We can illustrate this proposal by means of a few minor extensions to the notation of Concurrent Pascal. Let us introduce an *environment* declaration, in order to define a context in which types can be declared. The contents of an environment are not to be made known outside of its local context by any implicit exportation rule; however, any names that are to be made generally available will be explicitly designated in an *exports* list. The items exported will, in general, be qualified types. A qualified type specifies the name of an abstract data type[4] (called the *base* type) and a set of operators applicable to it. The qualifying set of operators must be a subset of the operators actually defined by base type. Only those operators that appear explicitly in the qualifying set are allowed to be invoked upon a variable declared as an instance of the qualified type.

We shall allow nesting of environment declarations, and type names may be passed outward as far as necessary by including them in the exports lists of surrounding environments. Within a given environment an abstract type definition, or module, will import (by default) the names of all types known or defined in the environment. We now have a nested syntactic structure in which static qualification can be added to a type by degrees, as the type is made known in successively wider environments.

A consequence of this scheme is that the static access rights held for use of a typed variable by a program module which imports it from a surrounding environment may exceed the rights of access that are available where the variable is declared. This kind of rights restriction is foreign to conventional block-structured languages, but corresponds closely to the notion of amplification[5,6] that has been developed in connection with capability-protected operating systems.

The concept is easily illustrated by an example. We return to the system whose access graph is given by Fig. 1. In this version of the skeleton program, the declarations of types *Pagebuffer* and *Charstream* are encapsulated within an environment definition from which only qualified types are exported:

```
initial process:
  E: environment
    exports Pagebuffer{ }, Charstream {Read, Write};
    type Pagebuffer = monitor;
      begin . . . end;
    Charstream = class (Buffer: Pagebuffer);
      procedure entry Read ( ); . . .
      procedure entry Write ( ); . . .
      begin . . . end;
    end E;
  type P = process (Buffer: Pagebuffer);
        var Messagestream: Charstream;
          begin
            init Messagestream (Buffer); . . .
          end
        var Protectedbuffer: Pagebuffer
          P1, P2: P;
          begin
            init Protectedbuffer
            init P1 (Protectedbuffer),
            init P2 (Protectedbuffer),
          end.
```

Note that in the context of the initial process, one can declare instances of the type *Charstream* and *Pagebuffer*, since these have been exported from the environment *E* in which they are defined. The exportation of *Charstream* gives it a qualified type with access rights to the procedures Read and Write. However, the exportation of *Pagebuffer* gives it a qualified type with no access rights whatsoever, except for the implied right of initialization. (It would make no sense in Concurrent Pascal to allow the declaration of a variable that could not be initialized, for the initialization statement is constrained to occur in the same context as does the declaration.) Subsequently, *Protectedbuffer* is bound as the actual parameter in the initialization of P1 and P2. Access rights to *Protectedbuffer* are held only by the instances of class type *Charstream* which occur within the two processes P1 and P2. This version of the program preserves the need-to-know principle.

The proposed static access-control mechanism as described thus far is still not adequate to meet the requirements of all situations. There are conceivable circumstances in which it might be necessary to duplicate a type definition in two or more environments in order to assure the type-correctness of a program. To illustrate such a case, consider the spooling system given as an example in Ref. 1. The part of this example that presents a problem is shown in the access graph of Fig. 2, in which the circles labelled with *C* denote instances of a

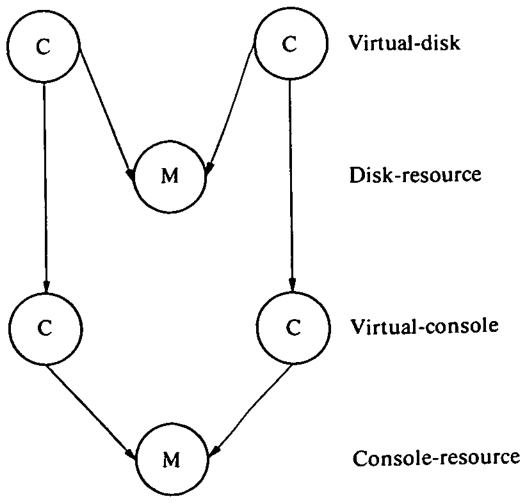

**Figure 2.** An access graph.

class type, and those labelled with *M* denote instances of a monitor type. The part of the system illustrated here provides virtual disk resources which are used by a pair of disk buffer monitors. These in turn serve the input, output, and job processes of a spooling system. These higher level program components are not shown in Fig. 2, nor are they included in the illustrative program texts that follow.

In this example, the variables *Disk_resource* and *Console_resource* are both instances of a common monitor type. However, the criteria for the access-correctness of these two instances are not the same. An instance of the class *Virtual_console* is to be declared locally within the class *Virtual_disk*. Both of the monitor instances, *Console_resource* and *Disk_resource*, are to be declared externally to these class instances and bound as permanent parameters, so that access to the monitors can be shared. But the code body of class *Virtual_disk* is to be allowed access rights to one of these monitors (*Disk_resource*) and not to the other.

To allow the discrimination of access rights, we suggest the use of *alias* names for the resource type, to be introduced in the exports list of an environment declaration. Access rights can then be specified separately, with each name given for the type. The following program illustrates the use of this scheme, and of nested environments, to give an access-correct skeleton of the spooling system. The declarative mechanism is both specific and powerful, yet quite simple. It may require some time to become accustomed to it, as the idea of exporting declarative information (types) outward in a nest of environments has no counterpart in purely block-structured languages. (The programming language Ada allows a module declaration (called a *package*) to export a type,[7] along with a set of procedural operators defined on the type. Ada does not provide a static parameterization capability like that of Concurrent Pascal, however.)

```
initial process:
  E2 environment
      exports Virtual_disk {all}, M2 { }, M1{ }
      E1 environment
         exports Virtual_console {all}
                 M2{ } = M, M1 {all} = M;
         type M = monitor;...
                 Virtual_console = class (Console: M);
                 ...
      end E1;
      type Virtual_disk = class (Disk: M1;
                                 Console: M2);
            var V_console: Virtual console;
            begin
               init V_console (Console);
               ... use V_console, Disk ...
            end;
  end E2;
  var D1,D2;Virtual_disk;
      Disk_resource: M1;
      Console_resource: M2:
  begin
        init Disk resource, Console_resource;
        init D1 (Disk_resource, Console_resource);
        init D2 (Disk_resource, Console_resource);
  end.
```

Here the notation {*all*} means all rights defined upon the base type.

## ALTERNATIVE SCHEMES FOR STATIC ACCESS-CONTROL

Two other schemes that have been proposed for control of static access binding are worthy of note. In Ref. 8 is proposed a notation in which access to a variable or a type is explicitly granted to a module that is to be allowed to refer to it. A *grant* is made as a declarative statement in the module which defines the object to which access is granted, and names both the object and the grantee explicitly. This mechanism seems well suited to a 'flat' hierarchy of modules, as opposed to a deeply nested hierarchy of module definitions. As proposed in Ref. 8, it lacks means of parameterization or for renaming of objects or grantees, which appears as a handicap if the method were applied in the design of large systems.

In the programming language Pascal-Plus[9], a novel form of textual substitution has been implemented to solve the static binding problem. A type declared as an *envelope* (replacing class types of Concurrent Pascal) or as *monitor* may contain an unspecified 'inner statement', denoted by '***'. When an envelope or a monitor type variable is declared within a block *B*, the body of its definition is expanded, with the remainder of the text of *B* substituted for the 'inner statement' of the envelope or monitor definition.

Devotees of macro expansion may find the Pascal-Plus solution preferable to the one we have proposed. We have tried applying the Pascal-Plus scheme to the examples given in the present paper, and have assured ourselves that the scheme can provide static binding consistent with the need-to-know principle. However, we found programming with a textual substitution scheme to be a very demanding exercise when module definitions were nested, as in the case of our second example.

The disadvantage of our proposal is that it plays upon the ability to limit the visibility of names in a way that will be unfamiliar to most programmers. An interactive programming support system would no doubt be of considerable help in following the transitions from one local name space to another, as one reads through a program text. However, the code body of a sequential procedure appears in proper sequence in a program text, which is not always the case with a macro substitution scheme such as that used in Pascal-Plus.

## CONCLUSION

We have proposed a simple modification to Concurrent Pascal to ensure that the *need-to-know* principle can always be adhered to in writing operating systems in that language. We have done so by presenting a few small extensions to the notation of Concurrent Pascal. These extensions allow the programmer to separate the definitions of rights of access from the declaration of a shared instance of abstract type. The rights of access may then be limited to those program components that have actual need to use a module, while allowing the module name to have a global declaration affording it the necessary scope. We have illustrated our concept by presenting a program text corresponding to Brinch Hansen's spooling system.

**Acknowledgment**

This work was supported in part by NSF Grants MCS 79 04817, MCS 77 02463 and in part by ONR Contract N00014-80-K-0987.

## REFERENCES

1. P. Brinch Hansen, The programming language Concurrent Pascal, *Institute of Electrical and Electronic Engineers Transactions on Software Engineering* **1** (No. 2), 199-207 (June 1975).
2. P. J. Denning, Fault tolerant operating systems, *Computer Surveys* **8** (No. 4), 359-389 (December 1976).
3. P. Brinch Hansen, The SOLO operating system, *Software, Practice and Experience* **6** (No. 2), 141-205 (April 1976).
4. A. K. Jones and B. H. Liskov, A language extension for controlling access to data. *Institute of Electrical and Electronic Engineers Transactions on Software Engineering* **2** (No. 4), 277-285 (December 1976).
5. A. K. Jones, Protection in programming systems, PhD. Thesis, Carnegie-Mellon University (1973).
6. E. Cohen and D. Jefferson, Protection in the HYDRA operating system. *Proceedings of the fifth ACM Symposium on Operating Systems Principles,* 141-160 (1975).
7. J. D. Ichbiah, *et al*, Rationale for the design of the ADA programming language. *ACM Sigplan Notices* **14** (No. 6), Part B (June 1979).
8. J. R. McGraw and G. R. Andrews, Access control in parallel programs. *Institute of Electrical and Electronic Engineers Transactions on Software Engineering* **5** (No. 4), 1-9 (June 1979).
9. J. Welsh and D. W. Bustard, Pascal-Plus-another language for multi-programming. *Software Practice and Experience* **9**, 947-957 (1979).

Received May 1980