ENCLOSURES: AN ACCESS CONTROL MECHANISM WITH APPLICATIONS IN PARALLEL PROGRAMMING AND OTHER AREAS OF SYSTEM PROGRAMMING

K. DELCOUR
Philips-Electrologica B.V., Apeldoorn, The Netherlands

and

A. J. W. DUIJVESTEIN
Technological University Twente, Enschede, The Netherlands

Received 25 March 1976

System programming languages, access control, parallel programming, synchronisation standards, monitor, protected regions, module hierarchies, stepwise refinement, hierarchichal authorisation, software production management, chief programmer team.

1. Introduction

After more than six years of experience with system programming languages in use for the design and implementation of compilers, operating systems, data base languages, utilities at Philips Electrogica, Netherlands, several extensions to the language SPL [1] have been proposed. At the Technological University, Twente, Netherlands, several programming tools, like macro systems, PL1, ALGOL 68 have been investigated for compiler construction. About four years ago the system programming language TAAL [2,3] was defined and implemented.

Although a considerable reduction of errors has been obtained compared with the use of assembly languages, there is still a number of fault sources that could be reduced by further improvements. A classification of errors in an industrial software production activity was reported by Klunder [4]. The experience with SPL showed two important error sources. The first category are wrong interfaces among which the uncontrolled use of global variables is the most serious one. The second main source seems to originate from the pointer variable. To cope with the control of global variables the concept of enclosures was proposed as an extension of SPL [5]. It is described in section 2.

Parallel to these improvements a need came up for providing facilities in both SPL and TAAL for describing synchronisation problems in parallel programming as arising in operating systems. Since there existed a lot of experience with the well known synchronisation primitives introduced by Dijkstra [6], we have tried to formulate a set of standard solutions in resource allocation problems making use of P and V operations. This is discussed in section 3. Furthermore due to the experience in the project Timesharing Operating System for a PDP 11/45 (TOS 45) of the informatics group of the Technological University Twente [7,8] we gained experience with Concurrent Pascal as a description tool. The use of the monitor concept appeared to be a useful tool in TOS 45, especially the version described by Hoare [9]. However, it is our opinion that it is still too risky to introduce the monitor, in particular the synchronisation of the monitor, which is still under discussion, in languages like SPL and TAAL, in view of the rather heavy expected costs involved with implementation. The internal variables of a monitor may only be accessed by its monitor procedures. This protection aspect of monitors is provided by enclosures in a similar way. An example is given in section 4.

The addition of enclosures appears to be a rather easy change of the compiler. Moreover, as will be pointed out in section 5, it is possible to build a monitor, assuming the availability of the enclosure mechanism and of semaphores and their operations.

We expect that a certain freedom in finding standard solutions in parallel programming will be necessary. We also expect that the enclosures and the classical synchronisation primitives are of such a primitive nature that these standard solutions can be implemented with these devices. Besides, enclosures can be used in many more cases where access control is necessary. This is discussed in section 6. The real significance of access control in a language is in its use by a management system to control the programming, for instance as described in [10]. This is briefly discussed in section 7.

We will use TAAL as a description language.

2. Enclosures

Enclosures have been described as an SPL-extension [5]; they can be added to any language with an ALGOL like block structure. In a block structured language the creation of a variable on block entrance implies the accessibility of that variable. The basic idea of enclosures is to separately control the creation and the accessibility. Below is shown how this feature could be added to TAAL. First the following example is given:

```
begin
   enc E1
      open
         int status variable init (0);
         global proc ENTRY (value int i) begin .... end;
         :
         :
      close;
   :
   :
end
```

The "enclosure heading" enc E1, the "enclosure brackets" open and close, and the "global marker" global are added language elements.

When the block is entered, space is allocated for the variables declared in it and their values may be initialised by init (expression). These actions are not influenced by the added language elements, that can be removed from the program without changing the space allocation and initialisation:

```
begin
     int status variable init (0);
     proc ENTRY (value int i) begin .... end;
   :
   :
end
```

An enclosure must lie completely within a block; a block is either a procedure body or begin declarations .... end or begin .... end. For the variables or other entities declared within the "enclosure body" the following three areas are defined:

1) inside the enclosure body (which is also inside a block) they are accessible.
2) outside the block all declarations are unknown; the variables do not exist.
3) inside the block but outside the enclosure body the variables exist. In this area, however, the variables or other declared entities of the enclosure can only be made accessible if they are declared as global.

In the given example, accesses to int status variable and proc ENTRY are allowed within the enclosures body. Within the block and outside the enclosure body "remote access" can be obtained to proc ENTRY by a parallel block or enclosure that specifies enclosure E1 in its "environment modifier". An environment-modifier is a list of one or more enclosure identifiers between paranthe-ses, inserted in the heading of an enclosure or block, as follows:

```
enc (E1) E2 open .... close                or
proc (E1) P1 (value int) begin .... end.
```

Enclosures, and blocks that are procedure bodies have already a heading that can accomodate such an environment modifier. A further addition to the language enables to give other blocks a heading as well:

```
block B1 begin .... end,
```

so that also such blocks can have an environment modifier;

```
block (E1) B1 begin .... end.
```

(Another function of the enclosure or the block with a heading is that it can be a "module" for separate compilation. During compilation, information derived from declarations in previously compiled modules is retrieved from a special file by the compiler.

This method of nested modularisation and hierarchical development of programs is briefly discussed in section 7.)

Within the body, the globals of the enclosures specified in the environment modifier given in the heading can be "remotely accessed" by "qualified identifiers", for instance

E1.ENTRY

By this rule, identifiers of **global** declarations, that are only unique within their enclosure, are made unique within the embracing block or enclosure.

The example can now be extended to show remote access.

```
begin
   enc E1
      open
         int status variable init (0);
         global proc ENTRY (value int i) begin .... end;
            .
            .
            .
      close;
   block (E1) B1
      begin
            .
            .
            .
         E1.ENTRY (1);
            .
            .
            .
      end
end
```

Like within a block, the identifiers that are known within an enclosure are the identifiers declared inside it – with or without **global** – and further any other identifiers of its "environment". If the normal rules for block structured languages are followed, the environment of a block or enclosure consists of all identifiers known within its lexically immediately embracing block or enclosure. Now, if "box" stands for "enclosure or block with a heading", then a box can modify this "lexical environment". In the given example the lexical environment of **block** B1 is the set of identifiers declared in the outer block consisting of the identifiers E1 and B1. By the environment modifier this set is extended with the qualified identifiers of all **global** declarations of enclosure E1. By other forms of the environment modifier, using the symbol **none**, the lexical environment can be suppressed or replaced, instead of extended. If only **none** is written an empty environment is the result. If **none but** is written in front of an explicit list, then the environment consists of the identifiers made accessible by that list; all other identifiers of the lexical environment are made unknown inside the box.

For instance, if in the given example the heading of B1 is replaced by

block (none) B1

then within the body of B1 only its local declarations are known. If the heading is replaced by

block (none but E1) B1

then also the identifiers E1 and E1.ENTRY are known inside the body, but any extra declarations that would be inserted in the outer block, for instance between the first **begin** and **enc** E1, would then be inaccessible by the body of B1.

A box that is immediately surrounded by an enclosure can be declared **global**. In the given example this is shown by **global proc** ENTRY. Procedure identifiers can thus be remotely accessed. This is useful for remote activation of procedures. All box identifiers that are declared **global** may be remotely accessed by environment modifiers. In that case, a special rule for remote access is valid, namely that if a known box contains boxes declared global then these boxes are also known.

This special rule is introduced to avoid writing extra boxes that are not useful for protection and would only make the program more complex.

```
begin
   enc E1
      open
         .
         .
         .
      global enc E2
         open
            .
            .
            .
         global proc P (value int k) begin .... end;
            .
            .
            .
         close
```

```
    close
      :
  block (E1.E2) B2
    begin
      :
      E1.E2.P(100);
    end
end
```

In this example the enclosure E1.E2 may be refered to directly in the environment modifier of B2. If the general rule for remote access were followed then an extra box, say B1, would be necessary around B2 to make E1's global enc E2 known to B2, as follows:

```
:
block (E1) B1
  begin
    block (E1.E2) B2
      begin
        :
        E1.E2.P(100);
        :
      end
  end
:
```

In an environment modifier it is allowed to refer to the identifier of an embracing box. This adds a set of qualified identifiers to the environment namely of all declarations – with or without global – written immediately inside the referenced box.

```
block A1
  begin int I;
      :
      block (A1) A2
        begin int I;
              :
              I
              :
              A1.I
        end
end
```

In this example, the reference to I within A2 denotes the local I of A2 whereas A1.I stands for the I declared immediately inside A1.

According to the definitions given so far, the identifiers added by the environment modifier are prefixed by their box identifiers These prefixes ensure that all identifiers added to the environment of a certain box are unique. In some programming methods, however, naming conventions might be followed in a group of boxes, that imply already the uniqueness of identifiers added to the environment of a certain box. The prefixes are not needed in that box in this case; they would only be cumbersome. By giving the symbol **implicit** after box identifiers refered to in an environment modifier, those box identifiers can be omitted as prefixes in the body of the box to which the environment modifier belongs. For instance, the block B2 in a previous example could be rewritten as follows:

```
:
block (E1.E2 implicit) B2
  begin
    :
    P(100);
    :
  end
:
```

Because **implicit** is written after E1.E2 in the heading of B2, the prefix E1.E2 is suppressed within the body of B2 for the globals of enclosure E2, so that P(100) must be written instead of E1.E2.P(100).

If a list of more than one box identifier – with or without **implicit** – is written in the environment modifier, each of them adds a set of identifiers to the environment. References from the body are resolved as if each set of identifiers were declared in a separate box inserted around the body and within the lexically surrounding box, the order of nesting from outside to inside being given by the order of the list from left to right. Consider the following program:

```
block B    begin
              :
           enc E1 open .... close;
              :
           enc E2 open .... close;
              :
           enc (E1 implicit, E2 implicit) F
              open
                 :
              close;
              :
           end
```

Say a reference *x* occurs in the body of *F*, then this *x* identifies a declaration of *x*, if any, within that body. If no *x* is declared in *F*, then the reference is to a declaration of *x* with **global**, if any, within enclosure E2. If that does also not exist, the reference is resolved by an *x* declared with **global** within enclosure E1 and if no such declaration exists, the binding is done with a declaration of *x* in the outer block B.

3. Standardisation in parallel programming

Especially in areas where larger programs are constructed it is a necessity to restrict oneself to standard solutions of which the correctness can be verified. Particularly in parallel programming there is a need for such solutions.

We make a distinction between mutex semaphores and private semaphores. We shall therefore introduce two new types. The first is **mutex semaphore** *x* **init** (integer value) and the second is **semaphore** *x* **init** (integer value). Although the initial value of a mutex semaphore is always 1 and that of the other (private) semaphore (used for blocking purposes) is always 0, we use the **init** facility of TAAL where variables can be initialised upon declaration with **init** (expression of suitable type). On both the mutex semaphore and the other semaphore we define two standard operations namely P(x) and V(x) with the well known effect [6]. A critical section is surrounded by a pair P (mutex semaphore) and V (mutex semaphore). We prefer that the compiler checks the pairing of P and V operations with respect to mutex semaphores.

Rather than programming the critical section in the following way

```
P(mutex);
if access to resource not allowed
then update status variable for waiting on resource; V(mutex);
     P(private semaphore)
else update status variable for using resource; V(mutex)
fi
```

where a proper check on P(mutex), V(mutex) is difficult we adopt the following solution.

```
P(mutex);
if access to resource not allowed
then update status variable for waiting on resource;
else update status variable for using resource; V(private sema-
     phore)
fi;
V(mutex);
P(private semaphore);
```

There is, however, more that should be checked. We would like to be sure that updating the status variables is only done inside a critical section and not at arbitrary places in the program. This requirement will be discussed later.

When a number of parallel processes compete for a set of common resources we adopt the following known technique.

```
parbegin
              :
process i:   do
                ENTRY;
                use resource;
                EXIT;
                other work
             od;
              :
parend
```

In the program section ENTRY, the status variables are investigated and it is determined whether the resource can be given to process i. This program section has the following structure.

```
ENTRY:  P(mutex);                    ¢ ENTRY of process i ¢
        if resource may be used by process (i)
        then update status variable for resource in use by
             process (i); V(private semaphore (i))
        else update status variable for resource being
             waited for by process (i)
        fi;
        V(mutex);
        P(private semaphore (i));
```

In the program section EXIT the status variable of the process that releases the resource is updated. Furthermore EXIT has the duty to inspect the status variables of the other processes in order to start waiting processes.

The structure of the program section EXIT is as follows:

```
EXIT:  P(mutex);                    ¢ EXIT process i ¢
       update status variable for resource not in use by
       process (i);
       while there are waiting processes that may now use
             the resource one of them being process (j)
       do update status variable for resource in use by pro-
          cess (j); V(private semaphore (j))
       od;
       V(mutex);
```

Rather than to write the full text of ENTRY and EXIT in each of the processes we prefer to call procedures ENTRY and EXIT. These procedure calls may be considered as services of a secretary. An activity of the secretary is to be considered as an extension of the process that is making use of the services of the secretary. It is therefore not a separate process. The status variables must be accessible by different secretary procedures and their values should be retained for new procedure calls. Furthermore it would be nice if the status variables could be accessed from the secretary procedures only. We can achieve this with the enclosure mechanism.

4. Use of enclosures in parallel programming

We will now give the structure of a set of parallel processes competing for a resource using the enclosure mechanism.

```
begin
   enc secretary
      open
         type status variable init (...);
         :
         mutex semaphore m unit (1);
         semaphore prisem (n) init (0); ¢ n has been declared
                  outside this block ¢
         global proc ENTRY (value int i)
            begin
               P(mutex);
               if resource may be used by process (i)
               then update status variable for resource in
                     use by process (i); V(private semaphore (i)
               else update status variable for resource being
                     waited for by process (i)
               fi;
               V(mutex);
               P(private semaphore (i));
            end;
         global proc EXIT (value int i)
            begin int j;
               P(mutex);
               update status variable for resource not in use
                     by process (i);
               while there are waiting processes that may
                     now use the resource one of them being
                     process (j)
               do update status variable for resource in use
                  by process (j); V(private semaphore (j))
               od;
               V(mutex);
            end
      close;
         parbegin
            :
            block (secretary) process i
               begin
                  do
                     secretary.ENTRY (i);
                     use resource;
                     secretary.EXIT (i);
                     remainder of process
                  od
               end;
            :
         parend
end
```

5. Implementation of the monitor of Hoare

The monitor provides two essential facilities. First there are data only accessible by a set of monitor procedures. The monitor procedures are taking care of the communication with the outside world. It is the only possibility to enter the monitor. Secondly a very specific synchronisation is provided. Hoare [9] has shown that his monitor synchronisation can be implemented by semaphores. We show that enclosures are sufficient to implement the data protection of the monitor.

To implement the synchronisation of his monitor Hoare [9] introduces a **mutex semaphore** (say m) initialised with value 1 that takes care of the mutual exclusion of the monitor. For each condition: cond he introduces a **semaphore** condsem initialised with value 0. Next he introduces a **semaphore** urgent, initially with value 0 and finally a variable urgentcount with initial value 0. Each exit from the monitor procedure is as follows:

```
if urgentcount > 0 then V (urgent) else V (m) fi
```

The operation cond.wait is implemented by

```
condcount plus 1;
if urgentcount > 0 then V (urgent); V (m) fi;
P (condsem);
condcount minus 1;
```

Finally the operation cond.signal is implemented by

```
urgentcount plus 1;
if condcount > 0 then V (condsem); P (urgent) fi;
urgentcount minus 1;
```

We introduce an enclosure (with name monitor) in which all status variables, semaphores and monitor procedures are declared. Only the monitor procedures are declared with **global**. The implementation of a monitor with enclosures is then as follows:

```
begin
   enc monitor
      open
         type status variable init (...);
            :
         mutex semaphore m init (1);
         semaphore condsem init (0);
         int condcount init (0);
            :
         semaphore urgent init (0);
         int urgent count (0);
         global proc monitorprocedure begin .... end;
            :
            :
         close;
         parbegin
            :
         block (monitor) process i
            begin
               do
                  :
                  monitor.monitorprocedure;
                  :
               od;
            end;
            :
         parend
end
```

6. Significance of enclosures

The "secretary", that was described in section 4 as a possible way of handling process synchronisation, is an example of a situation that occurs frequently in programming, where the following requirements have to be met simultaneously:

1. delimiting the area in the program where references to a group of declarations may occur.
2. allowing references to a selected subset (consisting of more than one element!) of that group from outside that area.

One meets this situation for instance, if one tries to describe, in a common block structured language, the instructions of a programmable machine, by a set of procedures declared in a block around the actual "program". The procedures implementing the instructions have to access common internal state variables or internal procedures of the programmable machine. Therefore, in the description these internal machine entities must be global to the instruction procedures, but then they are also accessible from the "program", which is not the case in the real machine that one is trying to describe.

```
programmable machine:

   begin
      int internal state;
      proc internal proc begin .... end;
      int accumulator;
         :
```

```
        :
      int MEMORY (8191);
      proc LOAD (int op) begin ... end;
      proc ADD (int op) begin ... end;
        :
program: begin
           LOAD (MEMORY(100));
           ADD (MEMORY(101));
             :
      end
   end
```

The array MEMORY, and the procedures LOAD and ADD belong to the "hardware-software interface" of the described machine. It is supposed that "internal state" and "internal proc", which do not belong to the hardware-software interface, have to be accessed from a number of "instruction-procedures", for instance LOAD and ADD. If, in the described machine, the "accumulator" is never an explicit operand of an instruction, it should, as such, also be inaccessible from "program". The same is true for the "instruction pointer", which is not shown in the example. The simple device needed in this situation, is a pair of brackets, say openarea and closearea, around all declarations of the block "programmable machine", and a way of marking the declarations that are accessible from outside those brackets. For this purpose the symbol global will be used. In this stage, the symbols openarea and closearea are used rather than open and close, to avoid confusion with the full enclosure-concept, of which the semantics are slightly different. The justification of these differences is discussed in the sequel.

```
programmable machine:
   begin
   openarea int internal state;
            proc internal proc begin .... end;
              :
            int accumulator;
   global   int MEMORY (8191);
   global   proc LOAD (int op) begin ... end;
   global   proc ADD (int op) begin .... end;
              :
      closearea;
program: begin
           LOAD (memory (100));
           ADD (memory (101));
             :
         end
end
```

The semantics of openarea, closearea and global deals only with the accessibility of names declared in a certain area of a program text. We would not want the dynamic creation and the life-time of variables etc., to be influenced by this device. The implementation of such a device as an added feature to an existing block-structured language is a rather simple task. The compiler has to accept some additional symbols that are interpreted only during the resolution of names but that have no effect on the object program. When tackling a large system programming task, often a first subdivision is made into partial development tasks, each consisting of providing a set of procedures on some common subject matter. It is likely that the development of such a set of procedures will give rise to the definition of common auxiliary entities that are used by that set of procedures only. These auxilary entities may be variables or procedures. In such a situation, the language feature introduced above can be applied. The program part resulting from the partial development task is bracketed between openarea and closearea. Only the set of procedures that had to be provided originally are marked global, so that only they can be referenced by the rest of the program. Trying to find analogy with the given example one can see that such a program part is a virtual "programmable machine" containing its auxiliary entities as "internal machine entities" and a specific set of procedures as "instructions" to be used by the rest of the program.

For a very complicated system program, it is useful to start out by carefully designing a structure of the program. For instance, an Operating System may be structured as a hierarchy of virtual machines. The description of a programmable machine in the example was used to illustrate a first useful addition to block structured languages. If one tries to generalise this example to a hierarchy of machines, it appears that something more is needed.

One can think of extending the given example to the description of a microprogrammed computer. An extra outer block is added that contains procedures representing the micro instructions, which are used by the procedures of the middle block. The latter procedures represent, as before, the programmer instructions used in the inner block, the "program".

The extended description is given below. Identifiers have, however, been changed to more general ones, so that the example can also stand for a program structured as a hierarchy of virtual machines. There are three machines: "bottom machine" in the outer block, "middle machine" in the moddle block, and "top machine" in the inner block. **Openarea, closearea** and **global** are applied in a similar way as in the previous example.

```
bottom machine:
begin
   openarea
      int internal variable of bottom;
         :
      global proc instruction 1 of bottom begin .... end;
         :
   closearea;
middle machine:
begin
   openarea
      int internal variable of moddle;
         :
      global proc instruction 1 of middle begin .... end;
      begin
         :
       instruction 1 of bottom;
         :
      end;
         :
   closearea;
top machine:
begin
   int internal variable of top;
   instruction 1 of middle;
      :
   end;
end
```

When this example is considered as a description of a microprogrammed computer, the "instructions of bottom" are micro instructions. It is supposed that they are used in the "middle machine", describing the microprogram, but from the "top machine", describing a program on the computer, they should be inaccessible, as in reality. In the example, however, every declaration that has been made **global** in the bottom machine, is accessible throughout that block, including all inner blocks. So the description is not adequate. A similar situation can occur in a system program structured as a hierarchy of virtual machines. In an Operating System, for instance, a virtual machine may provide an abstraction from some specific features of the computer on which that virtual machine is implemented, so that its users need not know those features and can be programmed independent of them. It would even be dangerous if those specific features of the computer could still be accessed by the users of that virtual machine directly. So the restriction that must be imposed on the hierarchy in the example to make it a realistic exercise is that the top machine may only use the middle machine, the middle machine may only use the bottom machine and the bottom machine may not use either of the other two.

The enclosure concept as defined in section 2 enables to impose such a restriction. It enables to specify such a relation among the virtual machines, that only the necessary references are allowed. Part of the use of a nested program structure may be replaced by this explicit reference relation. However, a certain amount of nesting levels may still be useful, especially in very large programs, to represent different levels of authorisation in a management system as discussed in section 7.

The basic difference with the brackets **openarea** and **closearea** is, that the surrounded area – the "enclosure" – now bears a name, so that the enclosure itself can be treated as a declaration. This makes it possible, firstly, to mark enclosures as **global**, which is important for nested enclosures, and secondly, to specify an enclosure in an "environment modifier" in the heading of a "box". Boxes were defined in section 2 as enclosures or blocks with a heading. The latter can be constructed with the symbol **block**. They may also be procedure-declarations. An enclosure identifier may be mentioned in the environment modifier of a box that lies outside the enclosure but within the scope of its identifier. This causes the **global** declarations of the enclosure to be accessible inside the box by identifiers qualified with the enclosure identifier. By using the symbol **implicit** in the environment modifier, the qualification of the names can be suppressed, and by using **none** or **none but**, names known outside the box can be made inaccessible inside it.

It is this last feature that can be used to adapt the example such that it represents precisely the required access possibilities among the three virtual machines. First the example is rewritten with the same nested structure as before. The two areas surrounded by the openarea; closearea brackets are replaced by enclosures with the names BOTTOM and MIDDLE. In the original version the body of the middle block had access to the globals declared in the outer block. With enclosures this access has to be granted explicitly in the heading. Similarly the globals of MIDDLE must explicitly be made accessible by the inner block. Now, by writing this as **none but** MIDDLE, the globals of the outer block are made inaccessible by the inner block.

```
block bottom machine
  begin
    enc BOTTOM
      open
        int internal variable of bottom;
        :
        global proc instruction 1 of bottom begin .... end;
        :
      close;
    block (BOTTOM) middle machine
      begin
        enc MIDDLE
          open
            int internal variable of middle;
            :
            global proc instruction 1 of middle
              begin
              :
              BOTTOM.instruction 1 of bottom;
              :
              end;
            :
          close;
        block (none but MIDDLE) top machine
          begin
            int internal variable of top;
            :
            MIDDLE.instruction 1 of middle;
            :
          end
      end
  end
```

In the original version, the nested block structure served to prevent "bottom up" references. Now that the reference relation among the boxes can be made explicit, the nested blocks are no longer needed in the example. Without them the following version can be written.

```
begin
  enc BOTTOM
    open
      int internal variable of bottom;
      :
      global proc instruction 1 of bottom begin .... end;
      :
    close;
  enc (BOTTOM implicit) MIDDLE
    open
      int internal variable of middle;
      :
      global proc instruction 1 of middle
        begin
        :
        instruction 1 of bottom;
        :
        end;
      :
    close;
  enc (MIDDLE implicit) TOP
    open
      int internal variable of top;
      :
      instruction 1 of middle;
      :
    close;
end
```

Apparently the example looks simpler now. The symbol **none but** is no longer necessary in the environment modifier of the box TOP, because in this version its "lexical environment" is empty.

The symbol **implicit** has been used to simplify the references to the "instructions" of the different machines. This simplification is possible, because in this example the identifiers of the instructions are supposed to be unique in the outer block and therefore do not need to be qualified by the enclosure-identifiers. Note that the full enclosure feature still affects only the resolution of names by the compiler and therefore its implementation is of the same level of simplicity as for the provisionary version with **openarea** and **closearea** discussed earlier in this section.

7. Hierarchical authorisation

One could bring up the objection that the programmer, who has the task to write the refinement of a box that is a procedure body, a block or an enclosure body, can still change the environment. The further idea behind enclosures is the existence of a management system. Let us assume that we work with chief programmer teams [10]. The chief programmer has other rights than the programmers of his team. He is responsible for designing the structure of the system; that is to say the skeleton of the system consisting of the data structures and program structures that are relevant for the top level of the system. There may be a number of boxes that are still undefined. In that case only box headings are given. It is the task of the programmers of the team to take care of filling in (refinement) the open bodies of the boxes. If we want a protection on changes of the structure designed by the chief programmer, we use a method that is called hierarchical authorisation. The structure or skeleton designed by the chief programmer is precompiled and stored on a file. The refinement result of the programmer is added to the precompiled skeleton. If the programmer tries to change the skeleton the compiler will not accept this. The programmer is only allowed to refine boxes of which only a header is specified. Obviously the access rights will be checked again on this level. Moreover it is possible to check the use of parameters for example input parameters, output parameters, value parameters, reference by value parameters etc. One can generalise to more than one authorisation level by introducing additional compile runs and storing the result. Compilation runs may be protected by a protection key if necessary. With the aid of this mechanism one can better control the design activity than without.

References

[1] SPL-3 Reference Manual. Internal publication, Philips Electrologica.

[2] H. van Berne, J. Schaap-Kruseman, A programming language for software description. First meeting of IFIP WG 2.4 Machine Oriented Higher-Level Languages, La Grande Motte, France (May 7–9, 1974). MOL bulletin issue no 4 to appear.

[3] TAAL, a programming language for software description, 1972. Documentation file, System Programming Group Technological University Twente. ALG.VGP302.

[4] J. Klunder, Experiences with SPL. Machine Oriented High Level Languages, Trondheim 1974. (North-Holland Publishing Company).

[5] K. Delcour, Enclosures as SPL-extension. Internal publication, Philips Electrologica.

[6] E.W. Dijkstra, Cooperating Sequential Processes. In Programming Languages (Ed. F. Genuys) (Academic Press, New York, 1968).

[7] W.A. Vervoort, Concurrent Pascal and the Design of Operating Systems (Dutch), Informatie 17 (1975) 675–683. An English version is available on request.

[8] W.A. Vervoort. Concurrent Pascal and the Design of a Timesharing Operating System. Proc. 1975. DECUS Europe Symp., The Hague.

[9] C.A.R. Hoare. An Operating System Structuring Concept, C.A.C.M. 17 (1974) 549–557.

[10] F.T. Baker. Chief programmer team management of production programming, IBM Systems J. no. 1, (1972) 56–73.