SYNCHRONIC ASPECTS of DATA TYPES : Construction of a non-algorithmic solution of the Banker's problem.

Piero R. Torrigiani

Gesellschaft für Mathematik und Datenverarbeitung mbH Bonn
Institut für Informationssystemforschung
Schloss Birlinghoven - Postfach 1240
D-5205 St. Augustin 1

Abstract

By means of two simple data structures, namely the "cell" and the "interrupt", increasingly complex structures are defined. The specification of these structures is given in a novel extension of the path notation which allows to study the synchronic characteristics of the types we define. These structures are shown to be sufficient to specify a non-algorithmic solution to dynamic - Banker's like allocation problems.

0. Introduction

The specification of systems involving concurrency is a hard task. Among the available tools, we shall use the object-oriented notation for path expressions presented in [1] : it uses a SIMULA like "class" concept and the syntactic extensions of the path notation introduced by P.E. Lauer and the author in the 1-st part of [2]. Interesting characteristics of this notation appear when they are used for the description of synchronic aspects of data structures. In this paper, by means of composition and modification of two simple structures, we construct a non-algorithmic solution to the problem of dynamically allocating units of a resource. The solution is non-algorithmic in the sense that it is described by a system rather than by an algorithm: no computation is performed in order to decide the order of allocation for concurrent requests, nor any global state of the system drives any decision making. An algorithmic representation is implicitly motivated

by considerations that have to do with a particular implementation. This, invariably obscures the concurrency which is inherent to the problem and which might be present in the solution. Hence we feel that it is important to obtain as a first step, a representation which is in the form of an abstract, concurrent system. For example, in the case of the Banker's problem a solution in the form of an algorithm, imposes a global synchronization or the requests for resources, which originate from essentially concurrent processes. This restriction arises from the nature of the scheme used for describing the solution. In our representation, this restriction will be absent.
In the next sections the notation is shortly introduced and explained through those examples which will be then used in the last section to construct the solution to the Banker's problem.

1. The notation

Programs in our notation consist of class definitions, class instantiations, paths and processes. As we are interested in the synchronic aspects of the structure defined by the classes, their definition will be only in terms of path expressions [3]; the paths defining a structure will state the possible orders of operations that an instantiated class provides to its environment i.e. to the processes and to the other system components. Some operations defined into a class may be possibly used only for internal synchronisation, hence they should not be known to the environment; this is obtained by means of a special declaration, called operation part, in the class where all exportable operations of the class (separated by commas) are listed and specified.
Informally we can say that paths (and eventually processes) are grouped together into a class to which a name is given; a use of this name in the body of a program generates an instance of that class which is also thereby given a name; the generation of an instance of a class consists in replacing the instantiation or call of that class by a copy of the paths (and eventually processes) which constitute it, after properly prefixing the instance-name to the occurrences of operation names in these paths (and processes). The generation of instances is assumed to take place at compilation time; after compilation programs will appear in the usual path notation. The semantic of path programs is given in terms of transition nets in [4] and we will show the nets that our programs produce. (In the figures showing the nets, the double lines belong to the processes, the dotted lines include instances of classes and exportable operations. When unnecessary the internal paths of instances are not shown). The grammar defining our notation can be found

in the appendix A2 in [1] .

2. Examples

Some examples should serve as introduction to the notation. In fig. 1 we see that the class specifies a very simple data structure consisting of a single "cell" where two operations are defined as exportable namely "deposit" and "remove"; the path which associates these operations (fig. 2) states that every "remove" on any instance of that cell must follow a previous "deposit" on the same instance of that cell, and that the first operation, again on any instance must be a "deposit". Mutual exclusion between processes accessing (both for reading and writing) is thus achieved together with the fact that no information will be lost. Various extensions of this structure can be found in [2] where path programms are given which define various buffers and buffer usage. Other applications of our notation are studied in [5,6,7] .
In the previous example nothing has been said about a fair use of the cell; fairness is by itself an interesting problem and the notation we use for our class definition has been proved useful in this sense [1,2] .
We have to mention that our approach is more similar to ideas concerning the use of arbiters than to those where fairness is obtained by the use of no arbiters [8,9] i.e. we are going to have no restriction on the relative speeds of a limited non-fixed number of "parallel" processes and we are going to have an arbitration hypothesis to be used in conflict resolution; (cf. [1] 3-rd part):
"If two or more operations in a path are mutually excluded those which are shared by processes have priority (of activation) over the others".
This hypothesis doesn't solve the starvation of the previous example as it doesn't distinguish between processes in a conflict; in the next section this problem is shortly described.

3. Sequencing through data definition.

Data structures can be defined in such a way that their operations provide the means for sequencing processes on other data. Figure 3 shows the structure needed for the "fair" sequentialization of an operation on a cell.
For those who are already familiar with the macro path notation, the that example should be quite clear. In any case, we wish to recall how the arbitration hypothesis is needed for the correct behaviour of this scheme; i.e. in order to guarantee that, for $1 \leq j \leq k$, the conflict between "INT(j).skip" and "INT(j).accepted"will be always solved in favour of the last after "INT(j).request" took place. (see fig. 3,4).
It is important to notice that now we have the means to distinguish

```
class cell;
  operation deposit, remove endop;
  path deposit; remove end;
endclass;
...
cell X;
...
process...;X.deposit;... end;
process...;X.deposit;... end;
process...;X.remove;... end;
```

Fig. 1

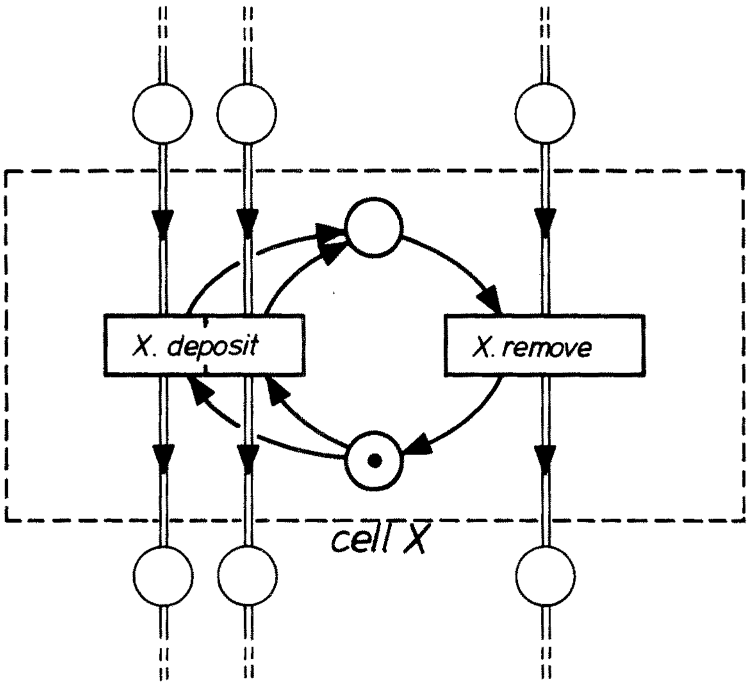

Fig. 2

with the help of indices the processes themselves.
Other discipline of sequentialization of operations, e.g. priority sequentialization, have been considered extensively in [1] ; here we need only to consider the following path, which, when substituted in the sequencer would yield a simple priority scheme:

```
path (INT(1).skip
     ;(INT(i).skip  i  ), (INT(i).accepted;INT(i).served)  2,k,1 ),
                  (INT(1).accepted;INT(1).served) end;
```

```
class interrupt;
   operation skip, request, accepted, served endop;
   path skip, (request;accepted;served) end;
endclass;
...
class sequencer (k: integer);
   array interrupt INT (k);
   cell X;
   operation deposit (i: integer) = INT(i).request;INT(i).accepted;
                                     X.deposit;INT(i).served,
             remove = X.remove endop;
   path [INT(j).skip,
        INT(j).accepted;INT(J).served) ω ; [j] | 1,k,1] end;
endclass;
...
sequencer SEQ(2);
...
process...;SEQ.deposit(1);...end;
process...;SEQ.deposit(2);...end;
```

Fig. 3

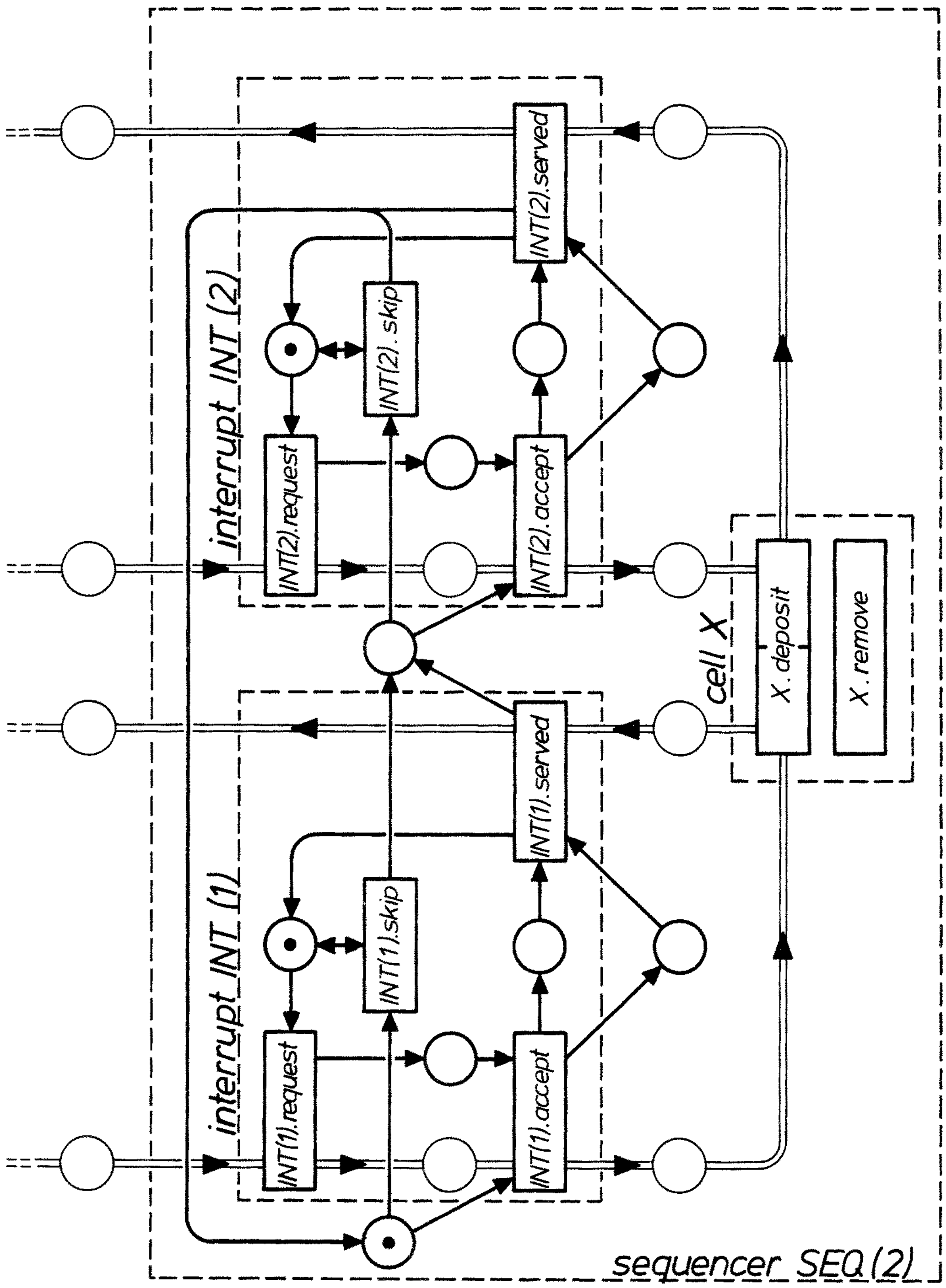
sequencer SEQ(2)
interrupt INT(2)
interrupt INT(1)
INT(2).served
INT(2). skip
INT(2).request
INT(2).accept
INT(1).served
INT(1).skip
INT(1).request
INT(1).accept
cell X
X.deposit
X.remove

Fig. 4

4. The pipeline

An interesting way of connecting cells into a buffer is a scheme where the cells are organised in such a way that every successive deposit must be preceded by the remove of the preceding cell; this fact implies that more processes may be concurrently depositing and removing without interferring with each other if they access, both for depositing and for removing, the cell of the buffer in the same order, and each of them in alternation, and by doing so the order with which they activate the first deposit is preserved up to the last remove. The data structure with this a synchronic characteristic has been called a pipeline, in fig. 5-6 we show how it will be used to define a "fair" queue. The path replicator in the definition of "queue" specifies the interconnections of the cells "W" to create a pipeline, while the path synchronize the "sequencer" with the pipeline. All the computation denoted by "comp" will be performed by the processes in a mutally exclusive manner; the order of "comp" will be the same order of activation of "SEQ.deposit" (any of them), hence, as the sequencer guarantees that all processes will eventually succeed in activating one of such operations, no process will starve. Incidentally one can note how this scheme resembles a monitor with a queue associated with the "wait" operation, and observe that the sequencing mechanism is here explicitly specified and that it is independent of the "duration" of the protected computation denoted by "comp".

```
class queue (k: integer);
  array (cell) W (k);
  sequencer SEQ (k);
  operation enqueue (i: integer ) = SEQ.deposit(i); SEQ.remove;
             [W(i).deposit;W(i).remove ∂ ; [i] |1,k-1,1] ; W(k).deposit
              dequeue = W(k).remove endop;
  [path W(i).remove;W(i+1).deposit end; [i] | 1,k-1,1]
  path SEQ.remove;W(1).deposit end;
endclass;
...
queue Q (2);
...
process...;Q.enqueue(1);"comp";Q.dequeue;...end;
process...;Q.enqueue(2);"comp";Q.dequeue;...end;
```

Fig. 5

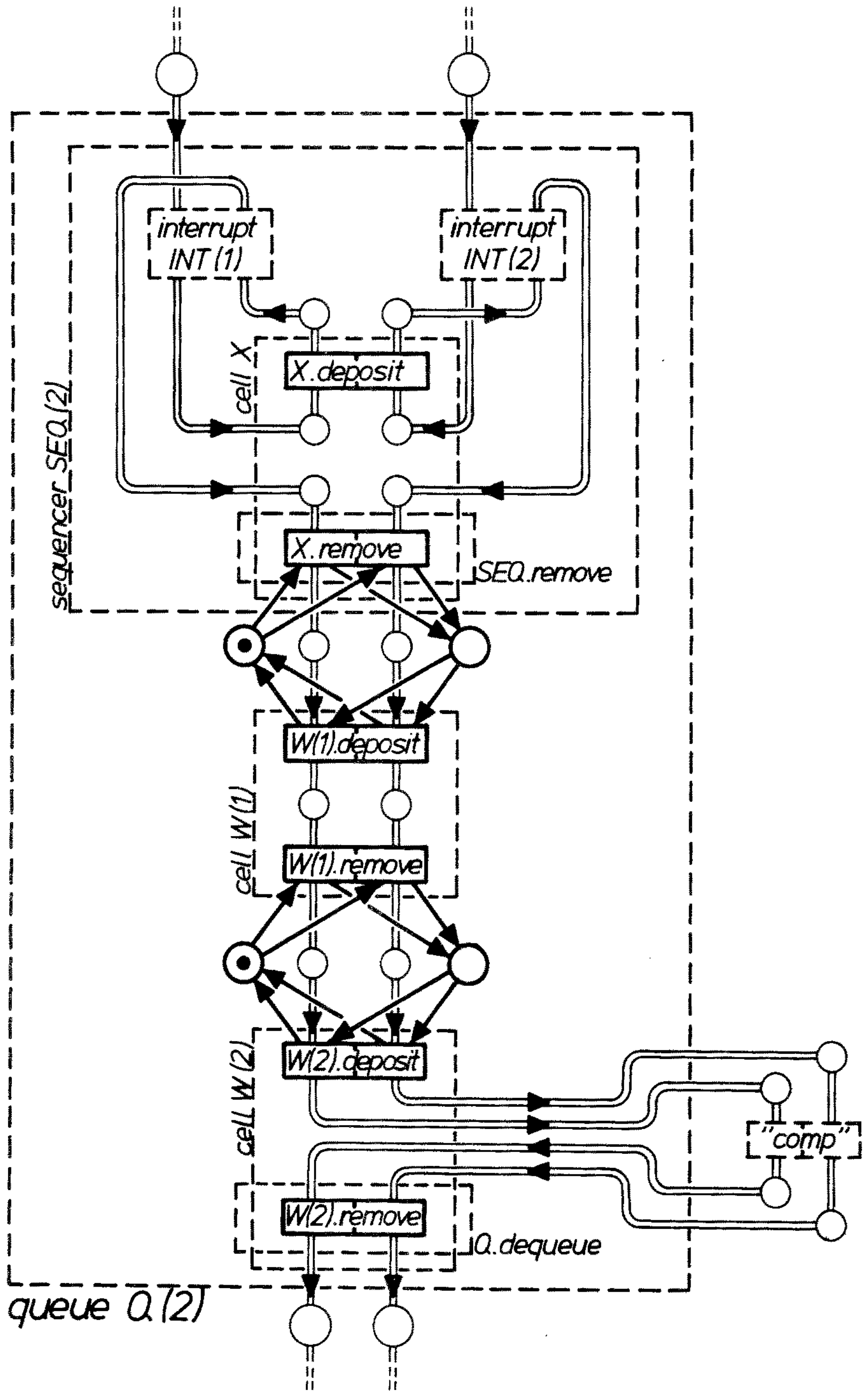
interrupt INT(1)
interrupt INT(2)
cell X
X.deposit
X.remove
SEQ.remove
sequencer SEQ(2)
W(1).deposit
cell W(1)
W(1).remove
W(2).deposit
cell W(2)
"comp"
W(2).remove
Q.dequeue
queue Q(2)

Fig. 6

5. The stack

Another data structure which is commonly used is the stack; as we did in the case of the queue we will not concentrate at all on the type of recorded data that the stack will contain and we will not hence define a stack of integers or reals or so; our main goal will be the definition of the synchronic aspects of any stack. The path in the stack definition states all possible sequences of pushes and pops that can happen to an empty stack of length k; i.e. (see also fig. 7,8) every push can be followed by a pop or by another push if the previous one was not the k-th one, and every pop can be followed by a push or another pop if the previous one was not the first one, and that initially the first push is the only possibility. In that path the replicator is used in its general form; again for the sake of simplicity we just present what that path corresponds to when the class is instantiated with k equals, e.g., three:

```
path Z(1).deposit; (Z(2).deposit; (Z(3).deposit;Z(3).remove)  ;
      Z(2).remove)  ; Z(1).remove end;
```

6. Counters

Both the pipeline and the stack can be used to define counters. We shall distinguish between two types of counters depending on the usage. The first type of a counter will keep track of the number of times some process has prevented some other process from accessing some common resource. The second type of a counter will show the number of units of a common resource that are available. We will use the pipeline scheme in order to define the first type of counter. (fig. 9,10). The stack provides the structure for the other type of counter. The cells "R" provide the operations of reading and resetting. (fig. 11,12).

```
class stack (k: integer);
  array (cell) Z(k);
  operation push = [Z(i).deposit ω , [i] | 1,k,1],
            pop  = [Z(i).remove  ω , [i] | 1,k,1] endop;
  path Z(1).deposit;
       [Z(i).deposit; [i] Z(i).remove)* ; | 2,k,1]
                   Z(1).remove end;
endclass;
```

Fig. 7

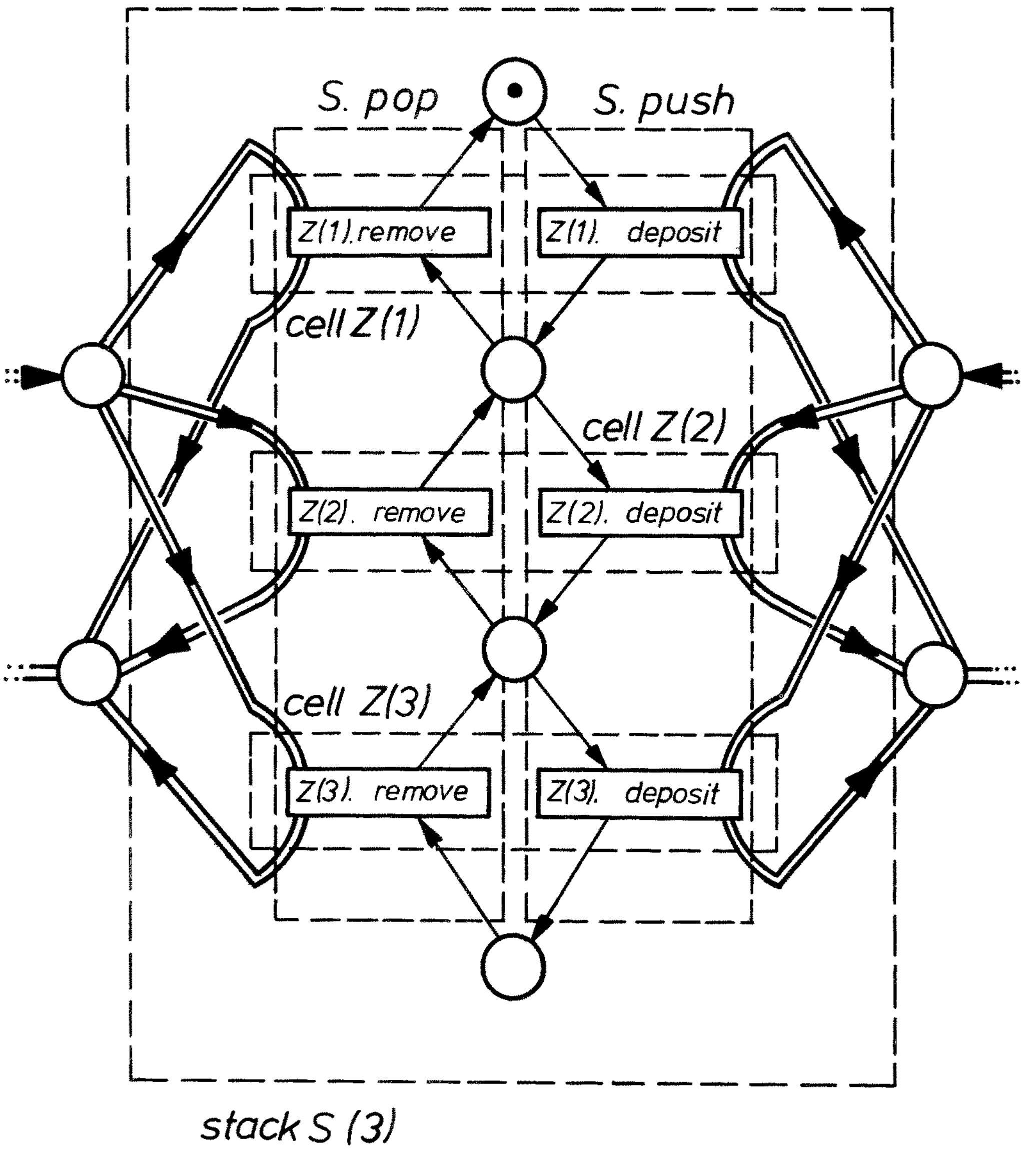
S. pop
S. push
Z(1).remove
Z(1). deposit
cell Z(1)
cell Z(2)
Z(2). remove
Z(2). deposit
cell Z(3)
Z(3). remove
Z(3). deposit
stack S (3)

Fig. 8

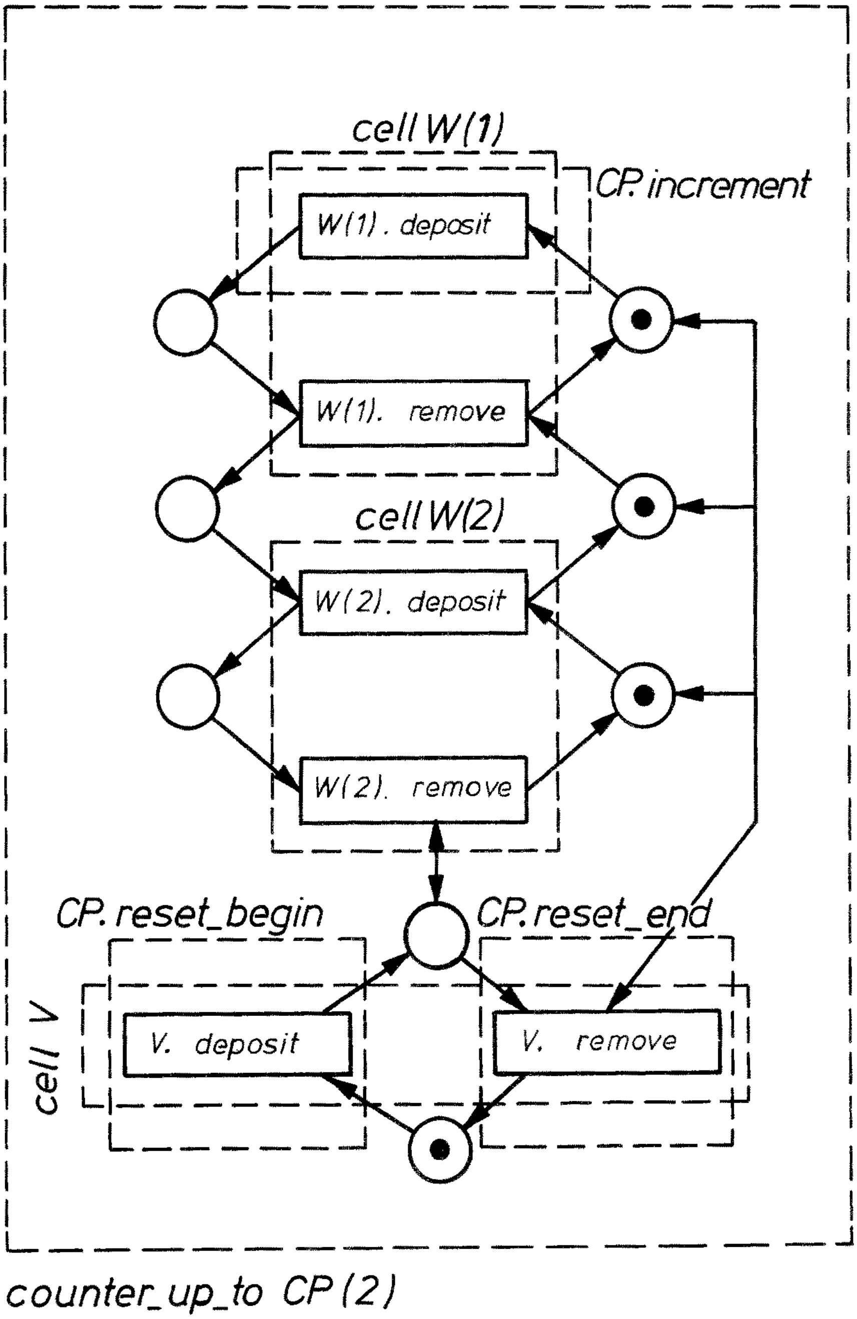
cell W(1)
CP. increment
W(1) . deposit
W(1). remove
cell W(2)
W(2). deposit
W(2). remove
CP. reset_begin
CP. reset_end
cell V
V. deposit
V. remove
counter_up_to CP(2)

Fig. 9

```
class counter_up_to (k: integer);
  array (cell) W (k);
  cell V;
  operation reset_begin = V.deposit,
            reset_end   = V.remove
            increment   = W(1).deposit endop;
 [path W(i).deposit;reset_end* ;W(i).remove end ω; $\boxed{i}$ | 1,k,1];
 [path W(i).remove;reset;reset_end* ;W(i+1).deposit end ω; $\boxed{i}$|1,k-1,1];
path V.deposit;W(k).remove * ;V.remove end;
endclass;
```

Fig. 10

```
class readable_counter (k: integer):
  array (cell)  W(k);
  array (cell)  R(k-1);
  operation increment = [ W(i).deposit  ω , $\boxed{i}$ | 1,k,1],
            decrement = [ W(i).remove   ω , $\boxed{i}$ | 1,k,1],
            read(1: integer) = R(1+1).deposit endop;
  path (R(k+1).deposit;R(k+1).remove),
 [(W(i).remove;((R(i).deposit;R(i).remove)* $\boxed{i}$ ) * ;W(i).deposit)
     ω, | k,1,-1] end;
endclass;
```

Fig. 11

7. A more complex example

With the apparatus we have we are ready to construct a system for the dynamic allocation of a resource to concurrent processes. It is worth remembering that the current programming languages (e.g. Concurrent Pascal) do not give the programmer tools for constructing solutions to this kind of problems, in spite of the extensions that have been recently proposed.[11,12] . A nice example of this kind of a problem is well-known Banker's problem which is due to Dijkstra. The solutions that have been proposed so far [13,14,15] are basically sequential solutions to a problem which inherently involves concurrency. Briefly stated, these solutions analyze the "present" states of a fixed number of concurrent processes to determine if there is a "sequence in which" these

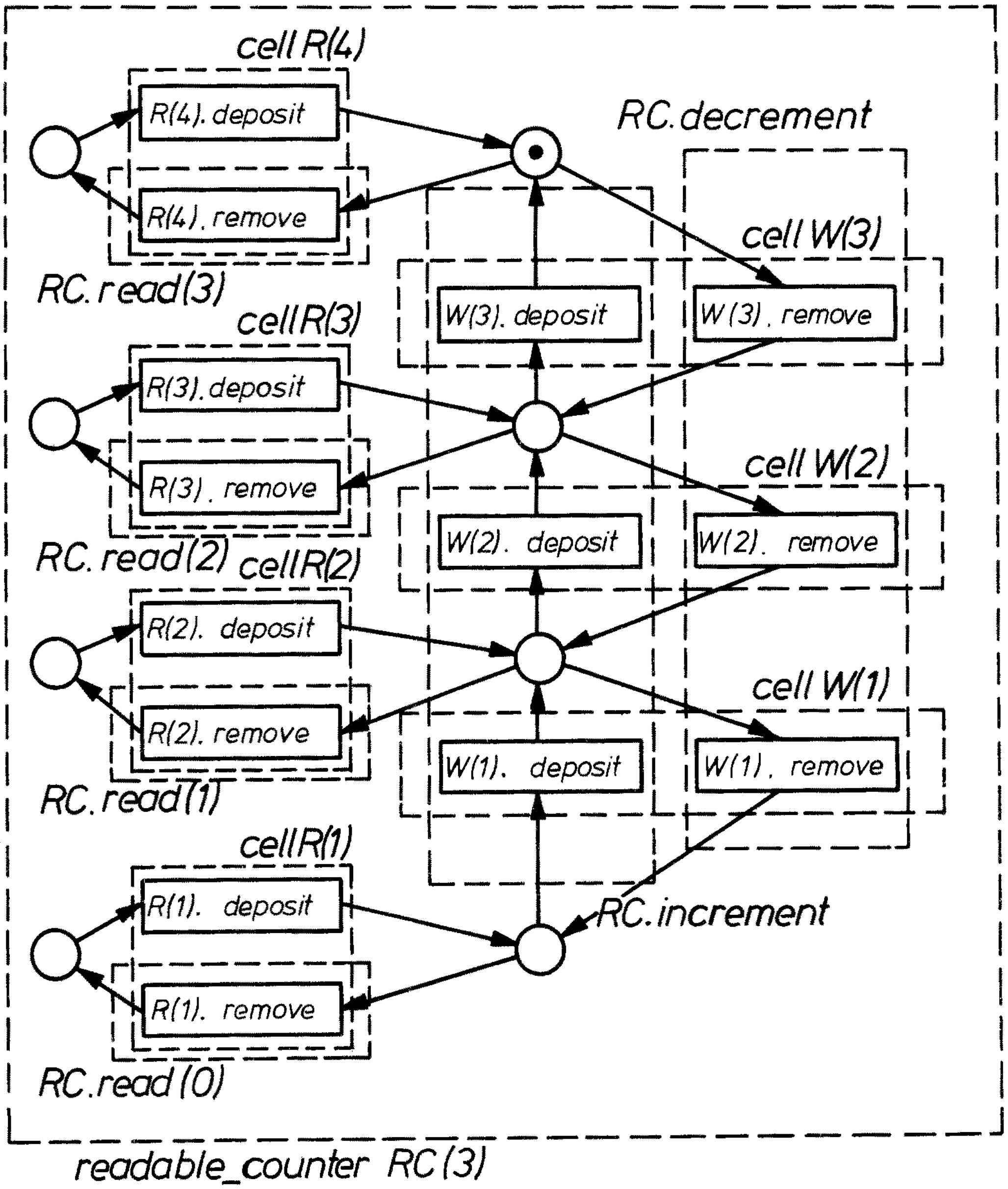
cell R(4)
R(4). deposit
R(4). remove
RC. decrement
RC. read(3)
cell R(3)
R(3). deposit
R(3). remove
cell W(3)
W(3). deposit
W(3). remove
RC. read(2)
cell R(2)
R(2). deposit
R(2). remove
cell W(2)
W(2). deposit
W(2). remove
RC. read(1)
cell W(1)
W(1). deposit
W(1). remove
cell R(1)
R(1). deposit
R(1). remove
RC. increment
RC. read (0)
readable_counter RC (3)

Fig. 12

"processes can be completed, one at a time if necessary".([15] p.124). Consequently, a state in the Banker's problem is defined to be "safe if it is possible for the Banker to enable all his present customers to complete their transactions within a finite time." ([15] p.43). This means that in these solutions what is guaranteed is that the present customers will complete their transactions. We shall now develop a solution to the Banker's problem which will permit an arbitrary and varying number of customers to concurrently enter and leave the "problem space". Our solution will also guarantee freedom from starvation. We shall build stepwise a system which will be by construction always in a safe state. (We will not allow any process to ask for loans bigger than the capital of the bank). In order to do so let us make first some simple observations: a) we have to avoid deadlock, b) we have to avoid starvation. It is true that b) implies a), but the converse, in general, is not true. Deadlock is consequence of "wrong" choices, and it can be solved by forcing the banker to make the "right" ones; but because of the unknown rate of requests of loans the right choice could indefinitely favour some processes over others violating b). Let us first observe the program shown in figure 13. It shows a bank and its customers; the state of the vault of the bank is represented by the readable counter C. Access to the vault (both for getting and giving back units, i. e. C.decrement and C.increment respectively) is protected by the interrupts RQ (ReQuest) and RL (ReLease), which may be viewed as the "cash-counters" of the bank. The Banker's behaviour is shown in figure 14.a. He applies a deadlock-free sequentialization of the operations on the vault. In simple words, one could imagine our banker sitting on the token in figure 14.a and moving from window to window in the bank serving the customers; from this figure we can see easily that he tries first his best to collect money into the vault, and only when he cannot do it any longer, looks at how much he has got and accordingly moves to the "correct" counter and accomplishes the requests of his customers. The Banker gives a single unit (enables C.decrement through the activation of RQ(j).accepted for some j) to exactly one of those who asks for an amount less than or equal to his current cash creating a new situation where for sure a customer (possibly the same one) will get units guaranteeing that at least one customer will get his last unit and will give the loan back. In fact in that program the long path is a modification of the path of the sequencer; here first of all there are two arrays of interrupts to be visited: the "RL" ones used for incrementing the counter, and the "RQ" ones used for decrement it (cf. the operation part); secondly we observe that the main cycle of the path is the one on the "RL" interrupts, which are visited in a priority manner

```
class bank (k: integer);
  array (interrupt) RQ,RL (k);
  readable_counter A (k);
     operation get_loan(i: integer) =
         RQ(i).request; RQ(i).accepted;C.decrement;RQ(i).served,
              give_back (i:integer) =
         RL(i).request;RL(i).accepted; [C.increment ∂ ; [z] |1,i,1] ;
         RL(i).served
endop;

path  [(RL(I).skip [; [ (C.read (1-1)
      [ ;(RQ(j).skip, (RQ(j).accepted;RQ(j).served))
         [j] | 1-1,1,-1] ) ∂ ,  [1] | k+1,1,-1]  [z]  k+1-i,k,-1]  [i]  ),
        (RL(i) . accepted;RL(i).served)) | k,1,-1]  end;
endclass;
...
bank B (3);
...
process...;B.get_loan(3);B.get_loan(2);B.get_loan(1);...
       ...;B.give_back(3);...end;
process...;B.get_loan(2);B.get_loan(1);...
       ...;B.give_back(2);...end;
process...;B.get_loan(1);...;B.give_back(1);...end;
```

Fig. 13

(cf. section 3); when all "RL.skip"'s are activated in a row all "read" will be enabled and depending on the contents of the counter the "QR" interrupts are visited one after the other starting from the i-th one if the counter's content was i. See also figure 14.a,14.b. Due to the fact that the processes are concurrent with one another it may happen (in the "worst" case) that only the process asking for a loan of one unit goes on making the others waiting forever; in fact the other two processes conflict with each other on the request of the secon unit, and the three of them conflict on the third. Again in simple words we could say that at the counter it is true that our Banker sees only one customer at a time and it is true that he (the Banker) is not serving a "wrong" customer, but only from his point of view: i.e. he is sure he will always have enough money to proceed in his task, but the confusion at the cash-counters may be such that the Banker is actually serving a "wrong" customer from a customers' point of view; e.g. at the

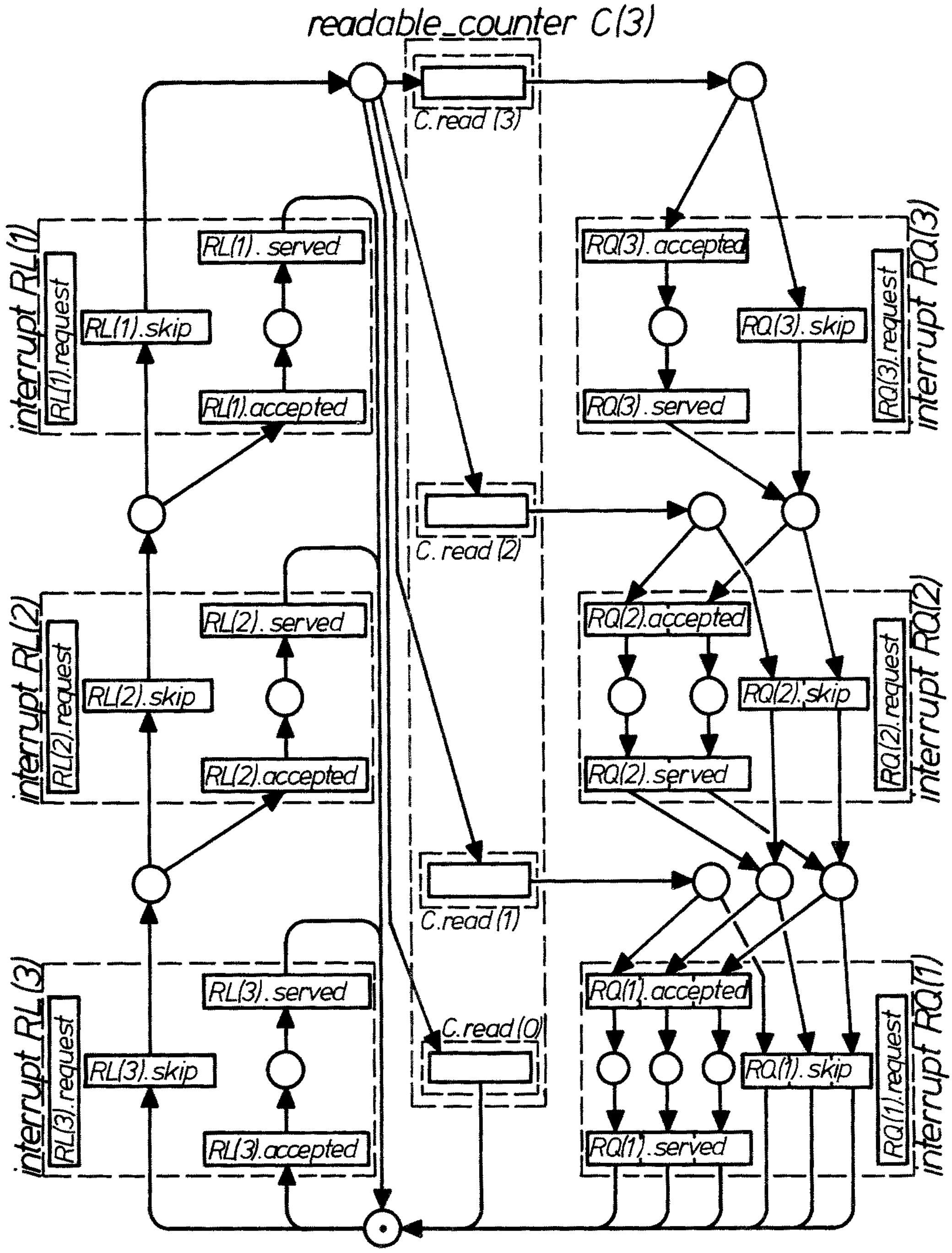
readable_counter C(3)
C.read (3)
C. read (2)
C.read (1)
C.read (0)
interrupt RL(1)
RL(1).request
RL(1). served
RL(1).skip
RL(1).accepted
interrupt RL(2)
RL(2).request
RL(2). served
RL(2).skip
RL(2).accepted
interrupt RL(3)
RL(3).request
RL(3). served
RL(3).skip
RL(3).accepted
interrupt RQ(3)
RQ(3).request
RQ(3). accepted
RQ(3). skip
RQ(3) . served
interrupt RQ(2)
RQ(2).request
RQ(2).accepted
RQ(2).skip
RQ(2). served
interrupt RQ(1)
RQ(1).request
RQ(1).accepted
RQ(1). skip
RQ(1).served

Fig. 14.a

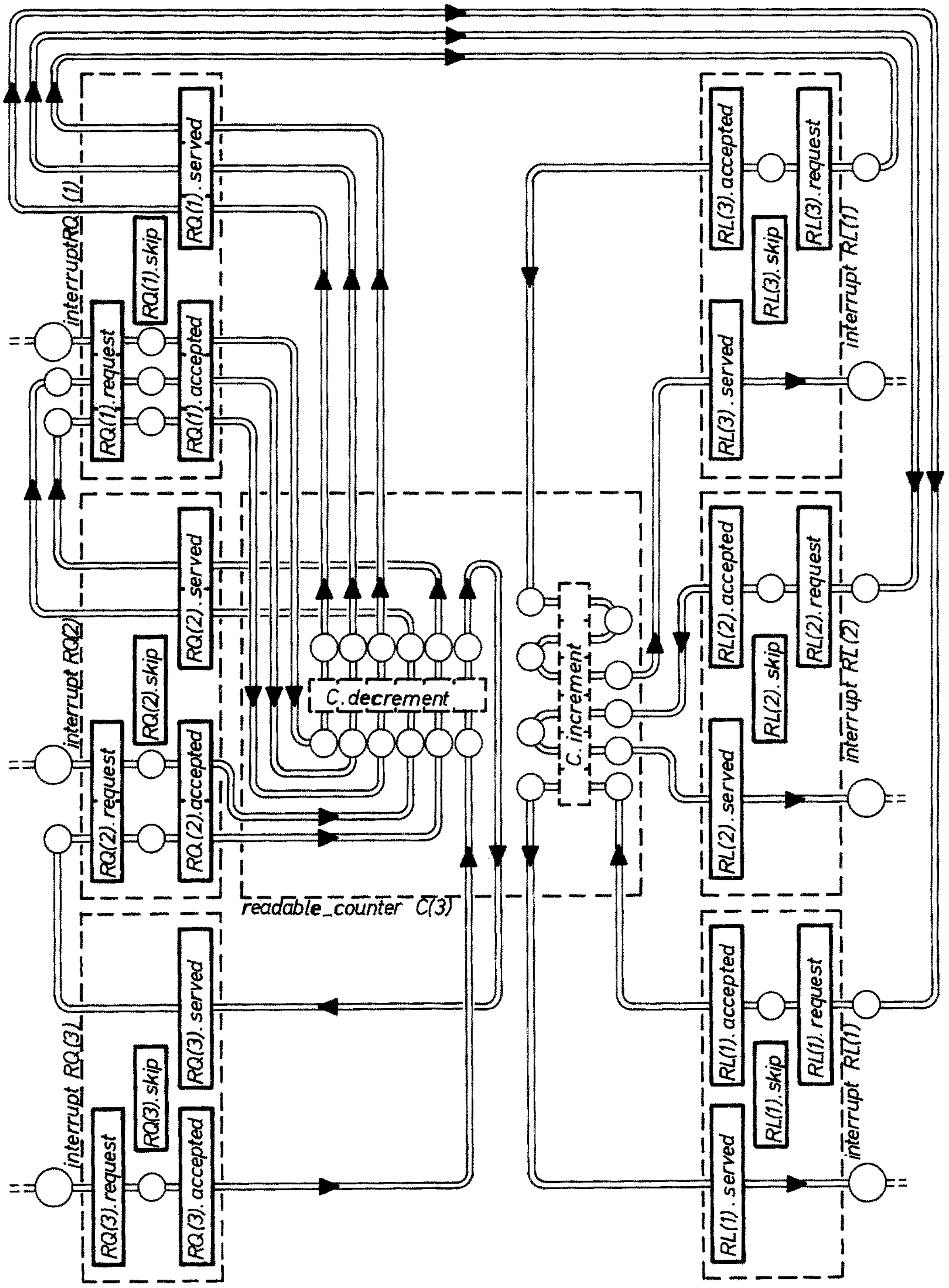
interrupt RQ (1)
RQ(1).request
RQ(1).skip
RQ(1).accepted
RQ(1).served
interrupt RQ(2)
RQ(2).request
RQ(2).skip
RQ(2).accepted
RQ(2).served
interrupt RQ(3)
RQ(3).request
RQ(3).skip
RQ(3).accepted
RQ(3).served
C.decrement
C.increment
readable_counter C(3)
RL(3).accepted
RL(3).request
RL(3).skip
RL(3).served
interrupt RL(1)
RL(2).accepted
RL(2).request
RL(2).skip
RL(2).served
interrupt RL(2)
RL(1).accepted
RL(1).request
RL(1).skip
RL(1).served
interrupt RL(1)

Fig. 14.b

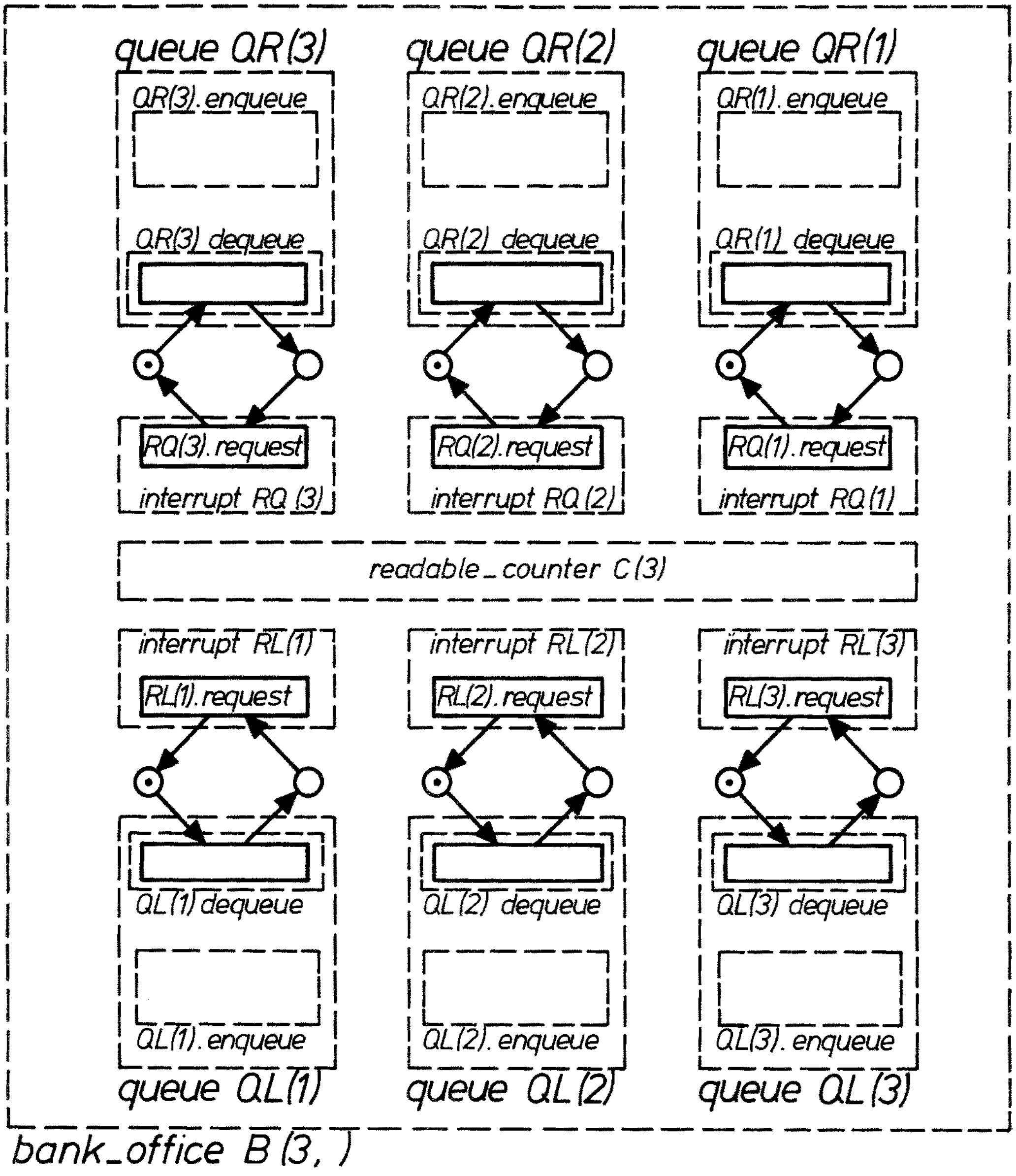
queue QR(3)
QR(3).enqueue
QR(3).dequeue
RQ(3).request
interrupt RQ(3)
queue QR(2)
QR(2).enqueue
QR(2).dequeue
RQ(2).request
interrupt RQ(2)
queue QR(1)
QR(1).enqueue
QR(1).dequeue
RQ(1).request
interrupt RQ(1)
readable_counter C(3)
interrupt RL(1)
RL(1).request
QL(1) dequeue
QL(1).enqueue
queue QL(1)
interrupt RL(2)
RL(2).request
QL(2) dequeue
QL(2).enqueue
queue QL(2)
interrupt RL(3)
RL(3).request
QL(3) dequeue
QL(3).enqueue
queue QL(3)
bank_office B(3,)

Fig. 15

second cash-counter (at the RQ(2).interrupt) there may be the customer who already got his first unit and the one who has not got anything yet and is asking for his first (of two) unit; this last one can be fast enough to always win the conflict at that cash-counter hence letting the other one starve. In order to avoid this conflict we can use "fair" queues on the requests for the second and third unit. By the way if we put long enough queues for all requests (and for all giving back pos-

```
class bank_office(k,m: integer);
  array (interrupt) RQ,RL (k);
  array (queue) QR(m-1),QL(m-1) (k);
  readable_counter C (k);
     operation queue_for_loan_of(i,j:integer)
        [QR(1).enqueue(j);QR(1).dequeue;RQ(1).request;
         RQ(1).accepted;C.decrement;RQ(1).served ∂;  [1] | i,1,-1],
             give_back(i,j:integer) =
         QL(i).enqueue(j);QL(i).dequeue(j); RL(i).request;
         RL(i).accepted; [C.increment∂; z 1,i,1] ; Rl(i).served
        (RL(i),accepted;RL(i).served))| k   ,1,-1]  end;
endop;
path  [(RL(i).skip [ ; [ (C.read(1-1)
       [;(RQ(j).skip, (RQ(j).accepted;RQ(j).served))
        [j] | 1-1,1,-1])∂, [1] | k+1,1,-1]  z  |k+1-i,k,-1]   [i]   ),
   path  QR(i).remove;RQ(i).request end;
   path  QL(i).remove;LQ(i).request end   ;  i  1,k,1  ;
endclass;
...
bank_office B (3,6);
...
process...;B.queue_for_loan_of(3,1);...;B.give_back(3,1)...end;
process...;B.queue_for_loan_of(3,2);...;B.give_back(3,2)...end;
process...;B.queue_for_loan_of(2,3);...;B.give_back(2,3)...end;
process...;B.queue_for_loan_of(2,4);...;B.give_back(2,4)...end;
process...;B.queue_for_loan_of(1,5);...;B.give_back(1,5)...end;
process...;B.queue_for_loan_of(1,6);...;B.give_back(1,6)...end;
```

Fig. 16

sibilities) we can allow that more than one process can concurrently request the same ammount of units without this starvation. The program is shown in fig. 16. The differences with the previous program consists only in the introduction of the queues QR and QL of the obligation to the processes to use their operations before interrupting the bank, and of the paths which connect the queues to the interrupts. See also figure 15. In that program whenever a process has been accepted the first interrupt it is not yet guaranteed that all its other interrupts will be accepted and served, not because of "unfair" conflict resolution (this is in fact taken care of by the queues) but because the speeds of the processes may be such that (after the initial situation) the counter will never reach again high values; i.e. the Banker will never have again the whole capital and will not be able to guarantee in a finite time high requests. In order to avoid this it is necessary that from time to time the bank office refuse to serve new customers and terminates serving those who have already got part of the loan or which are by the way "inside" the bank office. This must be done with some care. If we simply "lock out" of the bank some new customers, when we open it again no guarantee can be given to all that they can in fact enter it before the new closing time! Actually the closing period is going to be period in which new loans request should be waiting while those which had been started are completed, hence all we need is some more space to let the new coming customers wait and a way to distinguish between new and old customers. This is achieved by substituting the queues "QR" with a more complex mechanism composed by two fair queues, one for the old and one for the new customers, and a special sequencer composed out of two normal sequencers and two interrupts, the first of which is connected to a counter, in such a way that all customers which go through that interrupt will be counted and eventually blocked up to when the reset on that counter is done, the programs in figure 17,18 show this mechanism: the substitution of the queues with the new ones in the bank office produce the programm in figure 19 which completely solves the problem. In fact in that programm one can notice that only the "new" customers increment the counter, (cf. the operation part of figure 19,18 and the paths in figure 17) that is every process will be counted only once i.e. when he is going to obtain the first unit of his loan; as the reset of all counters is done every time the readable counter is found to held is maximum value, (after the activation of C.read(k+1) and before visiting all QR queues: cf. the long path in figure 21) the Banker let new customers wait only when he judges that he served too many people without having had his whole capital back. (This judgement is simply a function of the length of the counters).

```
class multiple_sequencer_and_counter(j,k,m: interger);
  cell M; cell N;
  array sequencer SEQ(k)  (j);
  array counter_up_to B(m) (j-1);
  array interrupt NT(j);
  operation reset_begin = N.deposit,
            reset_end = N.remove,
            deposit(i,j) = SEQ(j).deposit(i); SEQ(j).remove;
                           NT(j).request; NT(j).accepted;
                           M.deposit; NT(j).served,
            remove = M.remove endop;
  path [NT(i).skip, (NT(i).accepted;NT(i).served ∂ ;
       [i] | 1,j,1] end;
  [ path SEQ(i).remove;NT(i).request end ∂ ; [i] |1,j,1] ;
  [ path NT(i).request;B(i).increment end ∂ ; [i] |1,j-1,1];
  [ path N.deposit;P(i).reset_begin;B(i).reset_end;N.remove end ∂;
        [i] |1,j-1,1]
endclass
```

Fig. 17

```
class double_queue (k,m: integer);
  array (cell) F (k-1); queue QOLD (k); queue QNEW  (k);
  multiple_sequencer_and_counter MSEQ(2,k,m);
  operation reset_begin = MSEQ.reset_begin;
            reset_end = MSEQ.reset_end,
            enqueue_and_count(i: integer) = QNEW.enqueue; QNEW.dequeues
            MSEQ.deposit(1,i); MSEQ.remove; [F(1).deposit;F(1).remove ∂;
            [1] |1,K-2,1];  F(k-1).deposit,
            enqueue (i: integer) = QOLD.enqueue; QOLD.dequeue;
                                   MSEQ.deposit(2,i);MSEQ.remove;
                                  [F(1).deposit;F(1).remove ∂;
                       [1] |1,k-2,1] ; F(k-1).deposit,
            dequeue = F(k-1).remove endop;
  path SEQ.remove;F(1).deposit end;
  [ path F(i).remove;F(i+1).deposit end ∂;  [i]|1,k-2,1] ;
endclass;
```

Fig. 18

```
class new_bank_office(k,l,m: integer);
  array (interrupt) RQ,RL (k);
  array (double_queue) QR(l-1,m) (k);
  array (queue) QL(l-1) (k);
  readable_counter C (k);
  cell R;
     operation queue_for_loan_of(i,j:integer) =
          QR(i).enqueue_and_count(j);QR(i).dequeue;RQ(i).request;
          RQ(i).accepted;C.decrement;RQ(i).served;
          [QR(1).enqueue(j);QR(1).remove;RQ(1).request;
          RQ(1).accepted;C.decrement;RQ(1).served ∂; [1]|i-1,1,-1],
               give_back(i,j:integer) =
          QL(i).enqueue(j);QL(i).dequeue(j); RL(i).request;
          RL(i).accepted;[C.increment∂; [2]|1,i,1] ; RL(i).served
endop;
path  [(RL(i).skip
    [;[ (C.read(l-1)[[ ;R.deposit;R.remove[2]|2k+1-l,k+1,1]
       ;(RQ(j).skip, (RQ(j).accepted;RQ(j).served))
       [j]|l-1,1,-1])∂, [1]|k+1,1,-1][2]|k+1-i,k,-1]  [i]  ),
        (RL(i).accepted;RL(i).served))| k-1,1,-1]  end;
 [path QR(i).remove;RQ(i).request end;
  path QL(i).remove;LQ(i).request end;
  path R.deposit;QR(i).reset_begin;QR(i).reset_end;R.remove end
      ∂; [i]|1,k,1];
endclass;
...
bank_office B (3,6);
...
process...;B.queue_for_loan_of(3,1);...;B.give_back(3,1)...end;
process...;B.queue_for_loan_of(3,2);...;B.give_back(3,2)...end;
process...;B.queue_for_loan_of(2,3);...;B.give_back(2,3)...end;
process...;B.queue_for_loan_of(2,4);...;B.give_back(2,4)...end;
process...;B.queue_for_loan_of(1,5);...;B.give_back(1,5)...end;
process...;B.queue_for_loan_of(1,6);...;B.give_back(1,6)...end;
```

Fig. 19

8. Conclusions

Through the simple synchronic characteristics of two structures, through their collection into classes and their connection by paths, new structures have been defined whose synchronic properties define the use of commonly used data types (queues, stacks, buffers) as well as of new types (sequencers, etc.). With this technique a well known allocation problem has been solved, taking into account the specific difficulties it involves: namely the avoidance of deadlock, and the avoidance of the two types of starvation. The first type of starvation overcome by "fair" sequentialization of equivalent conflicting operations, the second by the proper use of counters which could also be adjusted (playing with their length) to optimize the system behaviour accordingly to the expected request-distributions.

References

[1] P.R. Torrigiani, P.E. Lâuer: An object oriented notation for path expressions, in AICA 77, Vol.3, pp.349,371, Pisa, 1977.

[2] P.E. Lauer, P.R. Torrigiani: Towards a system specification language based on paths and processes, Computing Laboratory, University of Newcastle upon Tyne, Technical Report Series, N., 1976.

[3] R. Campbell: Path Expressions: a techniques for specifying process synchronization, Ph.D. Thesis, University of Newcastle upon Tyne, August, 1976.

[4] P.E. Lauer, R. Campbell: Formal semantics for a class of high-level primitives for coordinating concurrent processes, acta informatica 5, 1975, pp. 247,332.

[5] R. Devillers: Non starving solutions for the Dining Philosophers problem, ASM/30, Computing Laboratory, University of Newcastle upon Tyne, 1977.

[6] R. Devillers, P.E. Lauer: Some solutions for the Reader/Writer problem, ASM/31, Computing Laboratory, University of Newcastle upon Tyne, 1977.

[7] P.E. Lauer, M.W. Shields: Abstract specification of resource accessing disciplines: adequacy, starvation, priority and interrupts,

Workshop on Global description methods for synchronization in real-time applications, AFCET Paris, 1977.

[8] K. Lautenbach: Ein kombinatorischer Ansatz zur Beschreibung und Erreichung von Fairness in Scheduling-Problemen, in Applied Computer Science, Hanser-Verlag, München, 1977.

[9] C.A. Petri: Modelling as a communication discipline, in 3rd international Symposium on Modelling and Performance Evaluation of Computer Systems, Bonn, Oct. 1877.

[10] R. Devillers, P.E. Lauer: A general mechanism for the local control of starvation: application to the dining philosophers and to the reader/writer problem, ASM/32, Computing Laboratory, University of Newcastle upon Tyne, 1977.

[11] A. Silberschatz, R.B. Kieburtz, A. Bernstein: Extending Concurrent Pascal to allow dynamic resouree management, in Proceedings of the 2nd international conference on Software Engineering, San Francisco, 1976.

[12] P. Ancillotti, M. Boari, N. Lijtmaer: Dynamic management in a language for real time programming, in AICA 77, Vol.1, pp.335-348, Pisa, Oct. 1977.

[13] E.W. Dijkstra: De Bankiers Algorithme, EWD116, Math. Dep. Technological u., Eindhoven, The Netherlands, 1965.

[14] A.N. Habermann: Prevention of system deadlocks, in CACM 12, N.7, 1969.

[15] P. Brinch Hansen: Operating System Principles, Prentice Hall Series in Automatic Computation; Englewood Cliffs, 1973.