A New Synchronization Mechanism called "Forcing Expression" and its Implementation

KIYOSHI SEGAWA*

A new synchronization mechanism—forcing expression—is proposed. Forcing expressions describe the behavior of critical sections by denoting the allowable combinations of processes and data concerning the sections. and cooperation problems can be treated equally be forcing expressions. Forcing expressions are high level synchronization mechanisms, but can be implemented by simpler synchronization primitives such as binary semaphore, cause/await and enq/deq operations. The definition and the implementation details of forcing expressions are described.

1. Introduction

Forcing logic is proposed to describe synchronization properties of critical sections in Process-Data Representation (PDR) [3, 4, 5, 6]—a specification method for parallel processings. Forcing expressions are defined as formulae in forcing logic.

Forcing expressions denote the allowable combinations of processes and data concerning the critical sections. Combinations are specified in terms of "at most . . . of" and/or "at least . . . of".

Forcing expressions can neatly represent *exclusion* and *cooperation* of processes, the key issues of synchronization, even in an equal manner. For example,

$$[A, B, C]: 1 \xrightarrow{CS} \langle D, E \rangle : 2$$

shall be read that at most one of the processes A, B or C can perform their critical section CS as to (at least 2 of) the common data D and E, and therefore specify mutual exclusion among A, B and C.

$$\langle A, B \rangle : 2 \xrightarrow{CS} \langle D \rangle : 1$$

shall be read that all (at least 2) of A and B should perform their critical section CS as to (at least 1 of) the common data D at the same time, and therefore specify cooperation among A and B.

Forcing expressions are truely high-level synchronization mechanisms. Using them, synchronization operations of critical sections can be specified (programmed) independently of others, e.g. contents of the sections. When we have to use low-level primitives such as semaphores, we should spread synchronization operations over the whole programs, leaving the programs unstructured with spaghetti of P's and V's. We could write programs in a more structured manner if we use higher-level constructs such as monitors, but we are not yet free from low-level primitives such as signal/wait operations as well as from other mechanisms, e.g. waiting queues.

The philosophy and the function of PDR are described in [6] and the implementation of forcing expressions is also roughly sketched.

We designed and implemented a parallel programming language called PDRL (Process-Data Representation Language) based on PDR principles, and forcing expressions play the key role as a synchronization mechanism in PDRL [9]. However, it is too complex to implement all functions of forcing expressions. Especially, as explained in section 3.2, the data-sides (the right-hand sides of the arrow) of forcing expressions contain some non-determinisms and these non-determinisms are not easy to be realized. Moreover, the rough sketch in [6], which describes the implementation of the process-sides (the left-hand sides of the arrow) of forcing expressions, is only a guide line and it contains several problems if we concretely implement both cases in which forcing expressions contain "[. . .]" and/or "⟨ . . . ⟩" nested and multiple forcing expressions are specified for one critical section. So, we place some restrictions on forcing expressions, but these restricted forcing expressions used in PDRL are as effective as the original ones.

In this paper, we describe the implementation of the forcing expressions in PDRL by using simpler synchronization primitives: forcing expressions can be translated into combinations of binary semaphores, cause/await [1], and enq/deq operations [2, 7, 10] in a systematic way. We also show that forcing expressions, which are high-level and powerful synchronization mechanisms, are realizable.

2. Definition of Forcing Expressions

Forcing expressions are constructed of the names of the critical sections and the allowable combinations of processes and data specified by forcing operators. In this paper, forcing expressions are defined informally. The formal definition is shown in [5, 6].

2.1 Forcing Operators

Forcing operators generate a family of subsets from the given set as follows:

*Department of mathematics, Waseda University, Tokyo, Japan.

$[x_1, \ldots, x_n]: k$ generates the family of subsets, whose cardinalities are not more than k, of $\{x_1, \ldots, x_n\}$,

$\langle x_1, \ldots, x_n \rangle: k$ generates the family of subsets, whose cardinalities are not less than k, of $\{x_1, \ldots, x_n\}$,

where k is an integer constant and called as the *suffix* of the operator.

Example 1.

$$[x, y, z]: 2 = \{\phi, \{x\}, \{y\}, \{z\}, \{x, y\}, \{y, z\}, \{z, x\}\}.$$

Example 2.

$$\langle p, q, r \rangle: 2 = \{\{p, q\}, \{q, r\}, \{r, p\}, \{p, q, r\}\}.$$

Forcing operators can be used within other forcing operators. If nested forcing operators are used, we apply the inner forcing operator first.

Example 3.

$$[\langle r_1, r_2 \rangle: 1, w]: 1 = [\{r_1\}, \{r_2\}, \{r_1, r_2\}, w]: 1 = \{\phi, \{r_1\}, \{r_2\}, \{r_1, r_2\}, \{w\}\}.$$

2.2 Forcing Expressions

A *forcing expression* is defined as the following formula:

$$A \xrightarrow{P} B \tag{1}$$

where A and B are sets specified by forcing operators, and P is the name of an operation. Forcing expression (1) means that an element of A does the operation P to an element of B. If A is the family of sets of processes, B is the family of sets of data, and P is the name of a critical section, then (1) means that a set of processes enters the critical section P and uses a set of data within P.

The left-hand sides of forcing expressions may be called *process sides* and the right-hand sides may be called *data sides.*

Simple forcing expressions are forcing expressions in which forcing operators are not nested.

Let us consider the following forcing expressions containing forcing operators on process sides:

$$[x_1, \ldots, x_n]: k \xrightarrow{P} B, \tag{2}$$

$$\langle x_1, \ldots, x_n \rangle: k \xrightarrow{P} B. \tag{3}$$

From the definition of $[\ldots]: k$, the process side of (2) is the set

$$\{X | X \subset \{x_1, \ldots, x_n\} \ \& \ \overline{X} \leq k\} \tag{4}$$

where $\overline{X}$ denotes the cardinality of X. Then, the meaning of the forcing expression (2) turns out to mean that an element of (4) does the operations P, and this may be said as "*at most* k processes out of $\{x_1, \ldots, x_n\}$ can do the operation P (to an element of B)". By the same way, the meaning of the forcing expression (3) turns out to mean that "*at least* k processes out of $\{x_1, \ldots, x_n\}$ should do the operation P". So, the forcing operator $[\ldots]$ may be called "*at most operator*" and $\langle \ldots \rangle$ may be called "*at least operator*".

Note that forcing expressions might specify that only ϕ does the operation if "at least operators" whose suffixes are 0 or "at most operators" appear on the process sides. Such a case—ϕ does the operation—can be regarded meaningless, so it is ignored in the following discussions.

We can also define the meanings of forcing expressions containing forcing operators on data sides.

In this paper, we assume that the name of a process or a datam does not appear more than once in one forcing expression.

2.3 Examples

Some specifications for typical synchronization problems are shown using forcing expressions.

Example 1. Readers' and writer's problem.

Let R_1 and R_2 represent two readers, W represent the writer, FILE represent the shared file to be accessed, and ACCESS be the name of the critical section. Then, the specification of this problem can be written as follows:

$$[[R_1, R_2]: 2, W]: 1 \xrightarrow{ACCESS} \langle FILE \rangle: 1.$$

The concrete description of the critical section "ACCESS" is out of the scope of this paper, but it might look like the following:

```
process R1:
  /* program of process R1 */
    :
  ACCESS:
    begin
      /* description of critical section */
    end;
      :
end.
```

Process R_2 and W also contain the critical section "ACCESS" in the same manner.

Example 2. Dining philosophers' problem.

Let PH_i $(i=1, \ldots, 5)$ represent the philosopher i, F_j $(j=1, \ldots, 5)$ represent the fork j, and EAT be the name of the critical section (in which philosophers eat spaghetti). Then, the specification of this problem can be written as follows:

$$[PH_1]: 1 \xrightarrow{EAT} \langle F_1, F_2 \rangle: 2$$

$$[PH_2]: 1 \xrightarrow{EAT} \langle F_2, F_3 \rangle: 2$$

$$[PH_3]: 1 \xrightarrow{EAT} \langle F_3, F_4 \rangle: 2$$

$$[PH_4]: 1 \xrightarrow{EAT} \langle F_4, F_5 \rangle: 2$$

$$[PH_5]: 1 \xrightarrow{EAT} \langle F_5, F_1 \rangle : 2$$

$$[PH_1, PH_2]: 1 \xrightarrow{EAT} \langle F_2 \rangle : 1$$

$$[PH_2, PH_3]: 1 \xrightarrow{EAT} \langle F_3 \rangle : 1$$

$$[PH_3, PH_4]: 1 \xrightarrow{EAT} \langle F_4 \rangle : 1$$

$$[PH_4, PH_5]: 1 \xrightarrow{EAT} \langle F_5 \rangle : 1$$

$$[PH_5, PH_1]: 1 \xrightarrow{EAT} \langle F_1 \rangle : 1.$$

3. Implementation

Forcing expressions can be implemented by simple synchronization primitives—binary semaphore, cause/await [1] and enq/deq operations [2, 7, 10].

Forcing expressions are translated into *initialization parts*, *prologues* and *epilogues*. At first, the translator for forcing expressions generates initialization parts for all of the forcing expressions and places them in the initialization process, which is used to initialize the whole program. Then the translator processes each program for each process. If it finds a critical section, it generates a prologue and an epilogue based on the forcing expressions corresponding to the section. The prologue and the epilogue serve as the *front guard* and the *rear guard* of the critical section respectively.

We show one particular implementation for process sides of forcing expressions. Implementations for data sides can be omitted under some reasonable restrictions. We explain this reason in the last section.

3.1 Implementation of Process Sides of Forcing Expressions

3.1.1 Translations of Simple Forcing Expressions

The process side of the simple forcing expression

$$[A, \ldots]: k \xrightarrow{CS}$$

is translated as shown in Fig. 1 (A is the process with which the translator deals).

P/V operations are ordinary binary semaphore operations which operate on "b_sem" and "wait_queue", and CAUSE/AWAIT operations are variants of Brinch Hansen's cause/await operations [1] which operate on "wait_queue" and "event_queue". He separates "event-queue" from the semaphore, but we extend the semaphore to include "event_queue". If a process executes "AWAIT" operation in the prologue surrounded by P/V, the process will release the ownership of the prologue (i.e. execute V operation), block itself and enter into the event_queue. When the other process executes "CAUSE" operation for the same semaphore, all of the blocked processes will be moved from the event-queue to the wait-queue and wait for V operation. When the process resumes its execution, it will resume from the (last executed) P operation (the first line of the prologue).

If only one "at most operator" (without nestings) is used, a counting semaphore is enough to implement it. We implement "at most operators" by the rather little bit complex way shown in Fig. 1 because we use nested forcing operators and multiple forcing operators. These things are described in the following sections.

```
   s: the counter for the forcing operator [. . .]
sem: the extended binary semaphore for the critical section CS,
     whose type is
     type extended_binary_semaphore=
       record
         b_sem       : binary semaphore;
         wait_queue  : queue;
         event_queue : queue
       end
initialization part:
     s:=k;
     sem. b_sem:=1;
     sem. wait_queue:=empty;
     sem.event_queue:=empty;
prologue:
     P(sem);
       if s>0 then s:=s-1
       else AWAIT (sem)
       fi;
     V (sem);
critical section: CS;
epilogue:
     P (sem);
       s:=s+1;
       CAUSE (sem);
     V (sem);
```

Fig. 1 Translation of at most operator.

For "at least operator ⟨. . .⟩", the type *inverse semaphore* is prepared (Fig. 2).

The process side of the simple forcing expression

$$\langle A, \ldots \rangle : k \xrightarrow{CS}$$

is translated as shown in Fig. 3.

3.1.2 Translations of Nested Forcing Expressions

For nested forcing operators, we must notice that the processes contained in the inner forcing operator are regarded as one process for the outer forcing operator. For example, if the forcing expression is

$$\langle A, B, [C, D, E]: 2 \rangle : 3 \xrightarrow{CS},$$

the processes A, B and one of the elements of [C, D, E]: 2 are needed to execute the critical section CS (from the discussion in section 2.2, ϕ is not considered as an element of [C, D, E]: 2). On the other hand, the processes B, C and D, for example, can not execute CS because two processes C and D belong to the same element of the outer operator.

```
type inv_semaphore=
  record
    counter    : integer;
    init       : integer;
    wait_queue : queue
  end
```

Fig. 2 Type inverse semaphore.

```
s: the inverse semaphore for the forcing operator <...>
sem: the extended binary semaphore for the critical section CS
     (it may be an ordinal binary semaphore in this case)
initialization part:
     s. counter:=k;
     s. init:=k;
     s. wait_queue:=empty;
     sem. b_sem:=1;
     sem. wait_queue:=empty;
     sem. event_queue:=empty;
     ENQ (s. wait_queue, EXCLUSIVE);
prologue:
     P (sem);
       s. counter:=s. counter-1;
       if s. counter=0 then
         DEQ (s. wait_queue, EXCLUSIVE)
       fi;
     V (sem);
     ENQ (s. wait_queue, SHARED);
critical section: CS;
epilogue:
     P (sem);
       DEQ (s. wait_queue, SHARED);
       s. counter:=s. counter+1;
       if s. counter=s. init then
         ENQ (s. wait_queue, EXCLUSIVE)
       fi;
     V (sem);
```

Fig. 3 Translation of at least operator.

"ENQ" is an enqueue operation and "DEQ" is an dequeue operation. ENQ/DEQ and programs (prologue/epilogue) are explained briefly in the appendix.

To implement such a situation, we must distinguish whether or not a process is the first/last one to enter/exit the critical section among processes belonging to the same (inner) forcing operator. In the above example, suppose that the process C attempts to execute CS. The condition specified by the inner forcing operator is checked, that is, the prologue for the inner forcing operator is executed. Only when the process C is the first one to enter CS among C, D and E (i.e. the processes D and E are not executing CS), is the prologue for the outer operator executed. On the other hand, suppose that the process C attempts to exit CS. The epilogue for the outer operator is executed only when the process C is the last one to exit CS. Of course, the epilogue for the inner operator is always executed.

So, the additional counter of the inner forcing operator is prepared and used in the prologue and epilogue for the outer forcing operator.

To generate the prologue for nested forcing operators, we generate the prologue for each of the forcing operators from inside to outside.

This sequence of prologues has a number of P's, V's and ENQ's (the last operation in the prologue for at least operator). The sequences of "V; P;" are not necessary, so they can be removed. Here, recall that it is the P operation from which the process executed AWAIT operation will resume its execution. So, before the AWAIT operation will be executed, all counters decremented/incremented after the last P operation must be reset to the former values.

The epilogue for nested forcing operators is generated in the same way. If there are more than one CAUSE operations, all but one are removed. Thus, the epilogue has only one pair of P and V, and (at most) one CAUSE operation.

In the prologue, the part from one P operation to the next V operation and the following ENQ operation (if it exists) is *one prologue unit*, in which there are, in general, a number of prologues for at most operators and after those there is *one prologue unit*, at least operator. Then, a prologue unit corresponds to the set of some at most operators and one at least operator. A typical example of such a set is

$$\langle[\ldots[\mathrm{A},\ldots]: k_1\ldots]: k_n\rangle: k_{n+1}. \qquad (5)$$

A set of forcing operators like (5) is one of the general nested forcing operators. So, we use the translation for (5) to show how to use additional counters and how to reset counters when AWAIT operation is executed. The translation of (5) is shown in Fig. 4.

3.1.3 Translations of Multiple Forcing Expressions

Multiple forcing expressions can be specified for one critical section such as Dining Philosophers' Problem in section 2.3, and a process can enter the critical section if all conditions specified by forcing expressions are satisfied.

We generate all prologues and epilogues for all forcing expressions as before.

If possible, it is best to execute all prologues/epilogues simultaneously. Note that, in Fig. 5, unit A–1, B and C–1 can be executed simultaneously if AWAIT operations and ENQ operations are executed correctly. (i.e. when AWAIT operation in unit B, for example, is executed, execution of units A–1 and C–1 will be suspended as if AWAIT operation in units A–1 and C–1 are executed. For example, if ENQ operation in unit A–1 is not completed but unit B and C–1 are completed, executions after units B and C–1 will also be suspended. Recall also that an execution of AWAIT operation causes counters to reset.

This is accomplished, if unit A–1, B and C–1 are merged into one unit and special attention is paid to AWAIT operations (and mechanisms to reset counters), as well as gathering parts for at least operators and ENQ operations at the rear of the unit. Units A–2 and C–2 are also merged. All epilogues are merged and unnecessary CAUSE operations are removed. Finally, there are one prologue and one epilogue. Thus, all prologues/epilogues for multiple forcing expressions are supposed to be executed simultaneously.

3.2 Comments on Data Sides of Forcing Expressions

One restriction imposed is that we can only use the forcing operator $\langle d_1, \ldots, d_m\rangle: m$ (the suffix must be same as the number of elements and nesting is not allowed) on the data sides of forcing expressions. With-

```
  s1: the counter for the innermost forcing operator
      :
  sn: the counter for the outermost "at most forcing operator"
s(n+1): the inverse semaphore for the "at least forcing operator"
  ci: the additional counter (i=1, ..., n)
sem: the extended binary semaphore for the critical section
initialization part:
    s1:=k1;
      :
    sn:=kn;
    s(n+1). counter:=k(n+1);
    s(n+1). init:=k(n+1);
    s(n+1). wait_queue:=empty
    ci:=0 (i=1, ..., n);
    sem. b_sem:=1;
    sem. wait_queue:=empty;
    sem. event_queue:=empty;
    ENQ (s(n+1). wait_queue, EXCLUSIVE);
prologue:
    P (sem);
      if s1>0 then s1:=s1-1
      else AWAIT (sem)
      fi;
      if c1=0 then
        if s2>0 then s2:=s2-1
        else
          s1:=s1+1;
          AWAIT (sem);
        fi
      fi;
      c1:=c1+1;
      if c2=0 then
        if s3>0 then s3:=s3-1
        else
          c1:=c1-1;
          if c1=0 then s2:=s2+1 fi;
          s1:=s1+1;
          AWAIT (sem)
        fi
      fi;
      c2:=c2+1;
        :
      if c(n-1)=0 then
        if sn>0 then sn:=sn-1
        else
          c(n-2):=c(n-2)-1;
          if c(n-2)=0 then s(n-1):=s(n-1)+1 fi
            :
          c1:=c1-1;
          if c1=0 then s2:=s2+1 fi;
          s1:=s1+1;
          AWAIT (sem)
        fi
      fi;
      c(n-1):=c(n-1)+1;
      if cn=0 then
        s(n+1). counter:=s(n+1). counter-1;
        if s(n+1). counter=0 then
          DEQ (s(n+1). wait_queue, EXCLUSIVE)
        fi
      fi;
      cn:=cn+1;
    V (sem);
    ENQ (s(n+1). wait_queue, SHARED);
critical section: CS;
epilogue:
    P (sem);
      s1:=s1+1;
      c1:=c1-1;
      if c1=0 then s2:=s2+1 fi;
        :
      c(n-1):=c(n-1)-1;
      if c(n-1)=0 then sn:=sn+1 fi;
      DEQ (s(n+1). wait_queue, SHARED);
      cn:=cn-1;
      if cn=0 then
        s(n+1). counter:=s(n+1). counter+1;
        if s(n+1). counter=s(n+1). init then
          ENQ (s(n+1). wait_queue, EXCLUSIVE)
        fi
      fi;
      CAUSE (sem);
    V (sem);
```

Fig. 4 Translation of nested forcing expression.

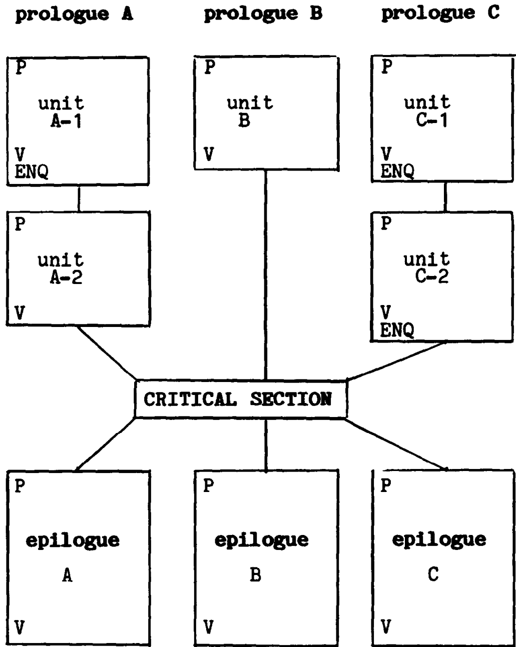

Fig. 5 Ideal execution flow of prologue and epilogue for 3 forcing expressions.

out this restriction, there exists some non-determinisms. For example, if the data side is $\langle d_1, d_2, d_3\rangle$: 2, the process must check which data—d_1 and d_2, or d_1 and d_3, etc.—are available in the critical section at run time. We think such non-determinisms are interesting but not necessary.

Under the restriction, you can regard acquiring all of the data $d_1, \ldots, d_m$ and entering the critical section as the same thing, so it is not necessary to deal with data sides of forcing expressions if process sides are translated properly. The informations in the data sides, however, must be used for checking (e.g. correct accessings to data) by the translator.

4. Discussion and Future Work

Forcing expressions can have several implementations

(interpretations) and only one of them is shown in this paper.

Two forcing operators [[A, B]: 2, C]: 1 and [⟨A, B⟩: 0, C]: 1 generate the same set {ϕ, {A}, {B}, {A, B}, {C}}, but they have different translations. We adopt such an implementation because we want to express the differences among the nuances of two forcing operators [...] and ⟨...⟩. There is, of course, the other implementation by which two forcing operators have the same translations if they generate the same set.

In order to show the correctness of the implementation and the equivalence among several implementations, we must formalize forcing logic more precisely.

We showed it is not necessary to implement data sides in section 3.2, but we would like to implement them and use non-determinisms in programs. The rough sketch of the implementation for data sides is shown in [3]. To implement data sides, however, more considerations are needed about the mechanisms of deciding the available data.

5. Conclusion

Forcing expressions are synchronization mechanisms which have the following characteristics. They are

1. able to treat exclusion problems and cooperation problems in the same manner,
2. high level mechanisms—they specify synchronization operations separately from descriptions of critical sections,
3. practical—they can be implemented.

Forcing expressions are powerful, but not sufficient for describing easily the transitions of states. To supplement it, process expressions [8] are used with forcing expressions in our PDRL implementation on DEC-SYSTEM-20 [9].

Acknowledgments

The author thanks Ken Hirose and Norihisa Doi for discussing with him. He also thanks Hiroyuki Bando and Haruyoshi Noda for helping him to implement forcing expressions. Thanks are also due to Katsuhiko Kakehi for his helpful comments.

References

1. Brinch Hansen, P. Operating System Principles, Prentice-Hall, 1973.
2. DEC. TOPS-20 Monitor Calls Reference Manual, Digital Equipment Corporation, 1980.
3. Doi, N. "At most" and "at least" operators are useful for specifying parallel processings, *Reports of Summer Programming Symposium 1981*, (Jan. 1982), 82–92 (in Japanese).
4. Hirose, K., Saito, N., Doi, N., Segawa, K. et al. Process-Data Representation, *Proc. of 3rd USA-Japan Computer Conference*, (Oct. 1978), 225–230.
5. Hirose, K., Saito, N., Doi, N., Segawa, K. et al. Forcing Logic in Process-Data Representation, Technical Report KIIS-79-01, Institute of Information Science, Keio University, 1979.
6. Hirose, K., Saito, N., Doi, N., Segawa, K. et al. Specification technique for parallel processing: Process-Data Representation, *Proc. of AFIPS 1981 National Computer Conference*, AFIPS Conference Proceeding, Vol. 50, (May 1981), 407–413.

413.

7. IBM. IBM System/360 Operating System: Supervisor and Data Management Services, IBM Corporation, 1967.
8. Lauer, P. E. and Campbell, R. H. Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes, *Acta Inf.* (1975), 297–332.
9. Segawa, K., Bando, H., Noda, H., and Doi, N. A specification language for parallel processings and its implementation, *WGSE Preprints of IPSJ*, No. 22, (Feb. 1982), 67–72 (in Japanese).
10. Shaw, A. C. The Logical Design of Operating Systems, Prentice-Hall, 1974.

Appendix

ENQ/DEQ are the operations for managing first-in first-out queues. The basic algorithms of ENQ/DEQ operations are shown in Fig. 6. Some error-handlings are omitted here.

Note that the first argument of ENQ/DEQ operations is a record of two queues, but, in Figs. 3, 4, that record is treated as one queue.

The roles of ENQ/DEQ operations are explained for the simple forcing expression

$$\langle A, B, C\rangle : 2 \xrightarrow{CS}.$$

Let $\underline{s}$ be the inverse semaphore and $\underline{sem}$ be the extended binary semaphore.

The initialization process initializes as follows:

s. counter: =2;

```
type proctype=(EXCLUSIVE, SHARED);
    Q=record
        tq: queue of proctype;
        pq: queue of process
      end;
procedure ENQ (var q: Q; var t: proctype);
  begin
    enter t into q. tq;
    if there is more than one elements in q. tq then
      if not (t=SHARED and
                all of the elements in q. tq=SHARED) then
        inactivate current process and
        enter it into q. pq
      fi
    fi
  end;
procedure DEQ (var q: Q; var t: proctype);
  begin
    var p: record
              t: proctype;
              p: process
            end
    remove t from the head of q. tq;
    if q. tq ≠ empty then
      p:=head of q;
      if p. t=EXCLUSIVE then activate p.p
      else if not (t=SHARED) then
              repeat
                activate p. p;
                p:=next element of q
              until p. t=SHARED
            fi
      fi
    fi
  end;
```

Fig. 6 Algorithms of ENQ/DEQ operations.

s. init: =2;
s. wait_queue: =empty;
sem.b_sem: =1;
sem. wait_queue: =empty;
sem. event_queue: =empty;
ENQ (s. wait_queue, EXCLUSIVE);

The initialization process can complete (the last) ENQ operation because s. wait_queue is empty. After completion, there is one element whose type is EXCLUSIVE in s. wait_queue.

If the process A, for example, attempts to enter the critical section CS, it waits till no process is executing prologue (/epilogue), and it will execute prologue. It decrements s. counter, and skips DEQ operation because the value of s. counter is 1. It releases the ownership of prologue and executes ENQ operation. The process A is blocked because there has been the element type EXCLUSIVE in s. wait_queue (in this stage, there are two elements in the queue—one is type EXCLUSIVE and the other is type SHARED).

If the process B, for example, attempts to enter the CS, it executes prologue in the same manner. In this case, it executes DEQ operation in prologue. The element type EXCLUSIVE is taken out from s. wait_queue, so the process A can resume its execution and the process B can also complete ENQ operation. Thus, two processes A and B enter the critical section CS (in this stage, there are two elements type SHARED in the queue).

If the process A, for example, attempts to exit the critical section CS, it waits till no process is executing prologue/epilogue, and it will execute epilogue. It executes DEQ operation and one element type SHARED is taken out from s. wait_queue. It increments s. counter, and skips ENQ operation because the value of s. counter is 1 (the value of s. init is 2).

If the process B attempts to exit the CS, it executes epilogue in the same manner. In this case, the process B executes ENQ operation. After the process B exits epilogue, there is only one element type EXCLUSIVE in s. wait_queue.

Thus, the element type EXCLUSIVE is used as a barrier.

(Received June 16, 1982)