Comments on Monitors and Path-Expressions

AKINORI YONEZAWA*

Two programming language constructs, monitors and path-expressions, which facilitate the implementation of *multi*-programming systems are compared and critically reviewed in the light of the programming methodologies which support quality software construction. The background and motivations of the research on the contructs are discussed. Comparison and arguments are made using familiar examples of synchronization primitives and synchronization problems.

1. Introduction

Recently two important proposals have been made on the concepts which support structured multiprogramming. One proposal is the "monitor" concept which was first introduced by P. B. Hansen [2] and later developed by C. A. R. Hoare [11]. The other proposal is the "path-expressions" which was introduced by Campbell and Habermann [3]. This paper focuses on the following two papers:

1) C. A. R. Hoare "Monitors: An operating System Structuring Concept" CACM Vol. 17 No. 10, October 1974, and

2) N. Habermann "Path-expressions" Report, Carnegie-Mellon University, Pittsburgh, Pa., June 1975.

and an attempt will be made to review these concepts, especially with respect to the question of how they contribute to the area of specification and implementation of parallel programs.

Since these two concepts can be viewed, in my opinion, as outgrowths of coupling of two research areas which have been separately studied, in the following two sections such two researches will be briefly surveyed as a background for our discussion.

2. Background I (low level synchronization primitives)

The whole motivation of introducing concepts such as monitors and path-expressions is to enhance the reliability and understandability of parallel programs by providing suitable mechanisms for parallelism and synchronization in high level programming languages. Before these two concepts were introduced in the literature, various primitive mechanisms for the synchronization of concurrently running processes had been proposed. The most well known among them is the P/V operation on semaphores proposed by E. Dijkstra [6] and used in the implementation of THE operating system [6]. Similar concepts such as wait/signal and other extentions of semaphores were also proposed. Although semaphores and other concepts are powerful tools for efficiently implementing low level details of synchronization problems, programmers always have to deal with too much details. Furthermore, no appropriate language constructs which encourage the well-structured use of P/V operations are provided. Therefore it is often quite difficult to understand or debug or prove the correctness of programs in which P/V operations are scattered. In Appendix I, a solution of the readers/writers problem (write priority) of Courtois, Heymans and Parnas [4] which is implemented using semaphores is given. In this implementation five semaphores are used for different purposes. The complexities of this solution need to be reduced by using more structured high level mechanisms for synchronization.

P. B. Hansen's idea [1] about methods of structuring the implementation of synchronization problems is to associate data shared by concurrent processes explicitly with operations defined on the shared data. He gave language constructs which realized this idea in PASCAL-like languages. For the purpose of comparison with more advanced concepts such as "monitors" and "path expressions", the language constructs are explained below. A shared variable of type T is defined as follows.

var v: **shared** T

The shared variable v can be referred to and modified by concurrent processes only inside the following construct, called a critical region:

region v do ⟨statements⟩

Critical regions referring to the same shared variable cannot be executed by more than one process at a time. When concurrent processes cooperate, they must wait until certain conditions on the shared variable are satisfied by other processes. For this purpose, a synchronization primitive:

await B

is used inside critical regions. If the condition B is not satisfied when a process evaluates await B in a critical region of a shared variable, the process is put to sleep and other concurrent processes are allowed to execute critical regions of the same shared variable. When

*Department of Information Science, Tokyo Institute of Technology.

B is satisfied by a completion of other critical sections by other processes, the dormant process is awakened and the execution of the subsequent statements is resumed.

By these constructs, a shared variable and operations on it are explicitly associated. This idea is closely related to the idea of data abstraction which will be discussed in the next section.

3. Background II (data abstraction)

In the context of structured programming, the importance of "abstraction" (namely, a process which extracts relevant information and supresses irrelevant detailed information in programming activities) has been emphasized [7, 12]. Besides the functional abstraction which has been realized as procedures in conventional programming languages, the concept of data abstraction was introduced by B. Liskov [13]. The idea of data abstraction is that an abstract data type defines a class of abstract objects which is completely characterized by the operations available on objects of that class. In short, an abstract data type is defined by operations which can be applied to it.

This idea was partially realized in the class concept of SIMULA-67[5] designed as a simulation language. Then, to realize this idea throughly, the cluster concept of the structured programming language, CLU [14] was designed so that programmers could easily realize their own data abstraction in the language. Furthermore the semantics of CLU guarantees the integrity of a created object of the abstract data type defined by programmers.

However, SIMULA-67 and CLU are initially not provided with any synchronization mechanisms and no parallelism is allowed in these languages. But when parallelism is taken into account, a certain abstract data type such as a message buffer needs to be treated more carefully. Depositing and removing a message are operations available on message buffers. But these operations are usually performed by concurrently running processes. To insure the proper use of a message buffer, an appropriate synchronization among those operations has to be made. Therefore, in order to characterize certain abstract data types (i.e. message buffers, stacks etc.), it is not sufficient to give all operations available on them when parallelism is involved. To characterize such data types completely, Hansen's idea, described in the previous section, that all operations on a data object shared by concurrent process should be explicitly associated with that data object, can be easily combined with the idea of data abstraction and a more complete idea of data abstraction which allows operations by concurrent processes becomes possible. The first attempt to realize this idea as language constructs is the monitor concept proposed by P. B. Hansen [2] and C. A. R. Hoare [11].

4. Monitors

A monitor defines a shared data structure and all the operations which can be applied to it. These operations are called monitor procedures. A monitor also defines an initialization of its shared data structure which may consist of more than one data object of various data types. To declare a monitor, Hoare used the following notations which were quite similar to the class declaration of SIMULA-67.

```
{class}* ⟨monitor-name⟩: monitor
  begin...declarations of data local to the monitor...
      procedure ⟨procedure-name⟩(⟨formal-parameters⟩)
        begin...declarations of data local to the ptocedure...
              ⟨procedure-body⟩ end
        ...declarations of other monitor procedures...
        ...initialization of data local to the monitor...
end
```

Fig. 1

Any process can attempt to call monitor procedures at any time. To express a monitor procedure call, the monitor name is always referred to in the following form.

monitor-name.procedure-name (⟨actual-parameters⟩)

The most important feature of a monitor which distinguishes itself from the ordinary class concept is that only one process at a time can execute a monitor procedure and any subsequent executions of monitor procedures defined in the same monitor must be held up until the previous execution of the monitor procedure has been completed.

A new type of variables called "conditions variables" can be declared local to a monitor. When a process encounters the statement:

a-condvariable.wait

in a monitor procedure, it must temporarily suspend its execution and wait in a queue associated with the a-condvariable without preventing other processes from executing monitor procedures. When a process executes the statement:

a-condvariable.signal

in a monitor procedure, a process waiting in the first place in the queue associated with a-condvariable will be reactivated. It is often necessary in writing monitor procedures to know whether or not the queue associated with a-condvariable is empty. To do so, the following boolean expression can be used.

a-condvariable.queue

This expression evaluates "true" if the queue is not empty.

In the following example of a message buffer, all the features of monitors explained so far (except the

*When more than one identical monitor is necessary, the monitor is defined as a class.

```
message-buffer: monitor
  begin
      buffer: message;
      count: 0 . . . 1;
      empty, full: condition;
      proc deposit (m: message)
          begin
              if count=1 then full.wait;
              buffer:=m;
              count:=1;
              empty.signal
          end
      proc remove (result m: message)
          begin
              if count=0 then empty.wait;
              m:=buffer;
          count:=0;
          full.signal;
          end
      count:=0;
  end
```

Fig. 2

last one) are used. The variable buffer is of message type. An actual deposit (i.e. buffer:=m in the procedure deposit) must be performed after the previous message is removed and an actual removal of message (i.e. m:= buffer in the procedure remove) must be performed after a new message is deposited. This synchronization is made by the wait/signal operations on two condition variables, empty and full.

So far queues associated with condition variables have been assumed to observe the first-in first-out (FIFO) discipline. To allow more flexible scheduling of waiting processes, "scheduled wait" is introduced. When a process is put in a queue associated with a condition variable, a certain number p which indicates the priority is assigned to the process as a parameter of the wait operation:

a-condvariable.wait(p).

When this a-condvariable is signalled, a process with the lowest number among processes in the associated queue is awakened and resumes its execution of the subsequent statements.

The second attempt to extend the idea of data abstraction so that concurrent processes are allowed to operate on shared data is the "path expressions" which will be presented in the next section.

5. Path Expressions

As remarked before, in defining a data type it is not sufficient only to give all available operations on that data type if parallelism is involved. The whole idea of "path expressions" is to provide in the definition of a data type a separate specification of how the operations (procedures) on the data type should be used.

Suppose p, q and r are procedures associated with a certain data type. The path expression:

path p; q **end**

specifies that p must be always performed before q. If both q and r should not be performed at a time (i.e. mutual exclusion), the path expression:

path $q+r$ **end**

specifies this requirement. The symbol + means exclusive selection. The pair of key words **path** . . . **end** represents Kleene's star operation. Also * can be used as the star operation in path expressions. For example, the following path expression is legitimate.

path p; $(q; r)^*$ **end**

The class of path expressions introduced so far can be viewed as the whole class of the regular expression constructed with two connecting operations; and +.

The following code is a path expression version of message buffers.

```
type message-buffer;
  begin
      var buffer=message;
      path deposit; remove end;
      let m=ref message in
    op message-buffer.deposit (m);
              begin buffer←m end;
    op message-buffer.remove (m);
              begin m←buffer end
  end
```

Fig. 3

Besides expressions for simple ordering of execution of operations, logical conditions on which operations can be performed are specified by "conditional elements" in a path expressions. A conditional element is given by the following form:

[⟨cond.1⟩: ⟨elem.1⟩, . . . , ⟨cond.n⟩: ⟨elem.n⟩, {elem.n+1}]

The left-most element which satisfies its conditions (if none of conditions are satisfied, then elem.n+1) represents the element of the above conditional form. There are basicly two restrictions on the condtions. First, conditions must be boolean and operands of the conditions must be either constant or data objects local to the type definition in which the path expression is defined. Second, all operations which change the operands of conditions must occur in the path expression in which the conditions appear. A path expression which defines stacks is a simple example of the conditional path expressions.

path [length=0: push, length=max: pop, push+pop] **end**

The above path expression specifies that if the length of a stack is zero, only the push operation can be performed and if the stack is already of maximum length, only the pop operation can be performed and otherwise push and pop operations are mutually exclusive.

There are three more constructs needed to be ex-

plained for the subsequent discussion. The first one is the specification of simple priority between two operations, say p and q which are mutually exclusive. Suppose the executions of both p and q have been already requested and are suspended until some preceding operation has been completed. The expression:

$p>q$

indicates that the execution of p will be started, when the preceding operation is completed. The second one is the multiple path expression. It is allowed to define more than one path expression in a type definition. For example, a multiple path expression:

path p; r **end**
path q; r **end**

specifies the order of the execution of p and r, and q and r, but does not specify the order between p and q. Therefore the above multiple expression can be used to express potential parallelism between p and q. The third one is the connected path expression. It is generally assumed that only one procedure among others which appear in a single path expression can be performed at a time. From this assumption, the following path expression made by connecting the above two path expressions:

path p; r & q; r **end**

imposes an additional constraint that p and q cannot be executed simultaneously (or concurrently) though the order between p and q are unspecified.

6. What is the Difference?

Our discussion of the two synchronization concepts will develop mainly by comparing them.

As long as no parallelism is involved, both concepts are based on the same thesis of "data abstraction" that a data type should be defined by operations available on that data type. As noted before, when parallelism is involved, the synchronization of operations on a data type must be specified in the definition of that data type in an appropriate way. In monitors, the synchronization of operations on shared data is realized (or specified) by two different mechanisms:

1) mutual exclusion among monitor procedures, and
2) wait/signal operations on condition variables inside monitor procedures.

On the other hand, path expressions directly and externally specifies the synchronization of procedures defined in the data type definition. In the definition of the message buffer type in Fig. 3, a very simple path expression:

path deposit; remove **end**

was sufficient for the specification of synchronization. However, the same type definition by a monitor in Fig. 2 needs to introduce two conditional variables and programmers have to take care of when and on what conditions the wait/signal operations must be performed.

Since path expressions require programmers only to gives external specification of how operations on shared data should be used, it seems to encourage programmers to write more well-structured programs than monitors do. But the following definition of semaphores shows that path expressions could be complicated even for a simple data type. The procedures prefixed by **proc** cannot be called by users directly. Users are allowed to call V-op and P-op only.

```
type semaphore;
  begin
    var s=integer (1);
    path [s>0: V-op+(P-request; P-granted),
          s=0: V-op+P-request,
          s<0: (V-op; P-granted)+P-request] end;
    op semaphore.V-op;
            begin s:=s+1 end
    op semaphore.P-op;
            begin P-request; P-granted end
      proc P-request;
              begin s:=s-1 end
      proc P-granted;
              begin ⟨null statement⟩ end
  end
```

Fig. 4

The complication of this definition results from the decomposition of the P operation into two internal procedures P-request and P-granted (Note that P-op does not appear in the path expression.). Such decomposition is necessary in order to suspend the completion of the P-operation if the counter s is negative after decreasing s.

In contrast to the definition above, the following definition of semaphores by a monitor is simple.

```
semaphore: monitor
    begin
        s: integer;
        busy: condition;
      proc V-op;
          begin s:=s+1; if s≦0 then busy. signal end
      proc P-op;
          begin s:=s-1; if s<0 then busy. wait end
      s:=1
    end
```

Fig. 5

7. Readers/Writers Problems (Which is More Structured?)

To point out and discuss various problems in monitors and path expressions, we will look at solutions of the readers/writers problem [4]. In Fig. 6 Hoare's solution [11] is given. A major difficulty with this solution, which originates from the whole concept of monitors, is that there are no specifications of in what order those four

```
class readers-writers-scheduler: monitor
  begin readcount: integer;
          busy: Boolean;
          Oktoread, Oktowrite: condition;
      proc startread;
          begin if busy ∨ Oktowrite.queue then Oktoread.wait;
                readcount:=readcount+1;
                Oktoread.signal
          end
      proc endread;
          begin readcount:=readcount−1;
                if roadcount=0 then Oktowrite.signal
          end
      proc startwrite;
          begin if readcount≠0 ∨ busy then Oktowrite.wait;
                busy:=true
          end
      proc endwrite;
          begin busy:=false;
                if Oktoread.queue then Oktoread.signal
                              else Oktowrite.signal
          end
    readcount:=0;
    busy:=false;
  end
```

Fig. 6

monitor procedures should be used. Suppose that "endread" is inadvertently or deliberately called before "startread" is ever called. Then readcount becomes negative and it will not indicate the number of readers in the data base any more, which completely ruin this solution. This serious shortcoming of the solution (or, more generally, solutions of various problems by monitors) also makes the correctness proof of the solution tremendously difficult. The whole text of programs which might have calls of monitor procedures has to be checked whether or not startread is always called before endread and in the worst case such a check cannot be carried out statically. Thus a plan of finding certain invariant relations (holding among objects local to a data type definition), which is a standard technique for the correctness proof of data representation [10], cannot be carried out. In Appendix II, one way of forcing users to call these monitor procedures in a proper order is proposed. The idea is to define an ordinary class readers-writers-data-base which takes an argument of data-base type and inside this class an instance of the above monitor (in Fig. 6), rws, is declared. And inside procedures READ and WRITE which are defined in the class, those monitor procedures (startread, endread, etc.) are called in a proper order. It seems impossible to realize the proper use of monitor procedures without any help of additional mechanisms such as the class or clusters. In this respect, path expressions allow programmers to write more well-structured programs than monitors do. Let us look at Habermann's solution of a readers/writers problem written in path expressions (Fig. 7). In this solution*, the semantics of the

*Note that the priority assignment among readers and writers in this solution differs from the one in Hoare's solution in Fig. 6.

```
type readers-writers-data-base (db=data-base);
  begin
        var r=integer (0);
        path [r=0: actual-writing, rquit]+rinit end
        path rinit < (write-attempt; actual-writing) end
    op read;
        begin rinit; ⟨...actual reading of db...⟩; rquit end
    op write;
        begin write-attempt; actual-writing end
      proc rinit; begin r:=r+1 end
      proc rquit; begin r:=r−1 end
      proc write-attempt; begin ⟨null statement⟩ end
      proc actual-writing; begin ⟨...actual writing of db...⟩ end
  end
```

Fig. 7

underlying language guarantees that only procedures preceded by **op** can be called by users. Other procedures preceded by **proc** cannot be called directly. So read and write are only procedures which users can call. The code of read guarantees that rinit is always called before rquit. However, as in the case of semaphores in Fig. 4, to suspend a process at a proper point in the writing operation and also to ensure the write request to be granted, the write operation in this implementation needs to be artificially divided into two procedures write-attempt and actual-writing.

8. Priority

In the solution of the readers/writers problem by path expressions (in Fig. 7), to assign priority to writers when both rint and write-attempt have been waiting to be executed, the following path expression was used.

path rinit < (write-attempt; actual-writing) **end**

But in terms of priority, the solution by the path expressions can be considered much more coarse than the monitor solution in Fig. 6. In fact, the monitor solution specifies sophisticated priority:

1) at the end of a writing operation if readers are waiting, a reader is granted and otherwise a writer is granted (if any) and

2) once a reader is granted, all currently waiting readers will be granted while waiting writers still have to remain in the queue.

To specify such priority by path expressions, some structural changes of the type definition such as decomposition of procedures and complicated conditional path expressions will be needed. It seems that the way monitors specify priority by wait/signal operations is easier for programmers to understand and implement, although they have to deal with wait/signal operations even when sophisticated priority specifications are unnecessary.

To complete our discussion on priority, it should be pointed out that implementations by path expressions have difficulties in expressing the first-in first-out scheduling discipline while queues associated with condition

variables in monitors naturally incorporate the FIFO mechanism.

9. Starvation

Both monitors and path expressions lack no particular mechanisms which might prevent or reduce the possibility of "starvation". Only thing we could argue is the easiness of proofs of no starvation. Since wait/signal operations are usually scattered in monitor procedures while path expressions externally specifies the order of execution of procedures, it could be said that path expressions are easier than monitors to show the starvation freeness. However, as discussed in the previous section, since sophisticated priority specifications would require complicated path expressions, it seems hard to judge which is better with respect to the starvation freeness.

10. Deadlock

Obviously a call of a monitor procedure within monitor procedures which are defined in the same monitor causes "deadlock". This situation is easily detected at compile time. But it is usually quite difficult (theorectically undecidable*, in general cases) at compile time to detect deadlock when wait/signal operations are scattered in monitor procedures or when monitor procedures are not specified as to how they are called in programs. The degree of difficulty in detecting deadlocks of implementations by monitors is almost the same as that of implementations by semaphores.

In the case of path expressions, a simple example of deadlock such as one caused by the multiple path expression:

path $f; p; q$ **end**
path $g; q; p$ **end**

can be easily detected at compile time, but the following example of the multiple path expression:

path $f; g$ **end**
path $p; q$ **end**

can cause deadlock when a process calls g after p and other process attempts to call q after f. It should be noted that this type of deadlock cannot be generally detected at compile time, because it is in general undecidable to determine whether or not particular calling sequences take place.

Both in the monitor and the path expression cases, the detection of deadlock at compile time is difficult except the ones reducible to the obvious situations mentioned above. Though problems of the deadlock detection at compile time tend to face the "undecidability barrier", it would be a fruitful research area to find decidable patterns of program execution by restricting the class of programs.

*The problem can be reduced to the halting problem of monitor procedures because we must check whether or not particular wait/signal operations in the program text are ever executed and such operations could be placed at the end of the monitor procedures.

11. Conclusion

In extending the idea of "data abstraction" to parallel programs, the concepts of monitors and path expressions have been introduced. In monitors, the synchronization mechanism is realized by the built-in property of mutual exclusion between monitor procedures and wait/signal operations on condition variables. Since in the monitor itself no language constructs which directly specify the legitimate order of execution of monitor procedures are provided, it is still subject to misuses of its monitor procedures even if a monitor itself is correctly defined. Therefore the correctness proof of implementations by monitors are often quite difficult to carry out. Furthermore, since the mechanism of wait/signal operations is almost as primitive as that of P/V operations, the detections of deadlock and starvation are also quite difficult. As shown in Appendix II, the idea of using local monitors inside the class or cluster definition will considerably reduce the difficulties with monitors. C. Hewitt's idea of "serializers" [9] can be considered as one realization of this idea in the context of the actor system.

Path expressions externally specify the legitimate order of the executions of procedures declared in a data type definition. Therefore implementations of synchronization problems by path expressiens can be much more well-structured than those by monitors. But on the other hand, because of lack of explicit mechanisms for primitive synchronization (such as wait/signal operations), it is often the case that procedures need to be decomposed into smaller unnatural procedures to attain the same effect of wait/signal operations. And sophisticated priority specifications require quite complicated path expressions even if the built-in expressions $>$ and $<$ are used. Although the path expression itself is not a specification language for synchronization problems, the idea of completely external specifications of the execution order of procedures will greatly encourage the study of specification techniques of parallel programs (For example, see [15] and [16].).

However, it remains to be seen whether or not complicated path expressions in which conditionals, priority expressions and unnatural decompositions appear can be easily written and understood by programmers.

Acknowledgements

The author would like to thank C. Hewitt and B. Liskov who suggested him to write an early version of this paper. Special thanks are due to E. Wada who corrected a mistake in the author's original implementation of semaphores by a monitor. Thanks are also due

to I. Kimura who encouraged the author to write this paper and helped him in receiving the comments of S. Andler.

References

1. BRINCH-HANSEN, P. Structured Multiprogramming. *CACM*, **14**, 10, (October, 1972).
2. BRINCH-HANSEN, P. *Operating System Principles*, Englewood Cliff. N.J., 1973.
3. CAMPBELL, R. H. AND HABERMANN, A. N. The Specification of Processes Synchronization by Path Expressions, *Lecture Notes in Computer Science*, **16**, Springer-Verlag, 1974.
4. COURTOIS, P. J., HEYMANS, S. F. AND PARNAS, D. L. Concurrent Control with 'Readers' and 'Writers', *CACM*, **14**, 10, (October, 1971).
5. DAHL, O. J., et al. Simula 67 Common Base Language. Norwegian Computing Center, Oslo, (May, 1968).
6. DIJKSTRA, E. W. The Structure of the 'THE'-multiprogramming system, *CACM*, **15**, 5, (May, 1968).
7. DIJKSTRA, E. W. Note on Structured Programming, Technische Hogeschool Eindhoven, The Netherland, (1969).
8. HABERMANN, A. N. Path Expressions, Computer Science Department Carnegie-Mellon University, Pittsburgh, Pa., (June, 1975).
9. HEWITT, C. Protection and Synchronization in Actor System, *ACM* SIGCOMM-SIGOPS Interface Workshop on Interprocess Communication. March **24–25**, 1975, Santa Monica, Calif. See also, ATKINSON, R. AND HEWITT, C. Synchronization in Actor Systems, SIGPLAN-SIGACT Symp. on Principles of Prog. Lang., Los Angeles, (January, 1977).
10. HOARE, C. A. R. Proof of Correctness of Data Representations, *Acta Informatica*, **1**, (1972), 271–281.
11. HOARE, C. A. R. Monitors: An Operating System Structuring Concept. *CACM*, **17**, 10, (October, 1974).
12. LISKOV, B. H. A Design Methodology for Reliable Software Systems, *Proc. of the AFIPS.*, **41**, (1972).
13. LISKOV, B. H. AND ZILLES, S. Programming with Abstract Data Types, SIGPLAN Notice, (April, 1974), 50–59.
14. LISKOV, B. H. A Note on CLU, Computation Structure Group Memo 112, Laboratory for Computer Science MIT, (November, 1974).
15. YONEZAWA, A. Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics, (Ph.D. Thesis) Technical Report 191 Laboratory for Computer Science, MIT, (December, 1977).
16. YONEZAWA, A. A Specification Technique for Abstract Data Types with Parallelism, *Proc. of Int. Symp. on Mathematical Studies of Information Processing*, Kyoto, (August, 1978). Also, to appear in *Lecture Notes in Computer Science* (E. K. Blum and T. Takasu eds.), Springer-Verlag, (1979).

Appendix I

A Solution to the Readers/Writers Problem (write priority)

```
integer readcount, writecount; (initial value = 0)
semaphore mutex1, mutex2, mutex3, w, r; (initial
value = 1)
(reader)
  P(mutex3);
    P(r);
      P(mutex1);
      readcount := readcount + 1;
      if readcount := 1 then P(w);
      V(mutex1);
    V(r);
  V(mutex3);
    ...
    ⟨actual reading⟩
    ...
  P(mutex1);
  readcount := readcount - 1;
  if readcount := 0 then V(w);
  V(mutex1);

(writers)
  P(mutex2);
    writecount := writecount + 1;
    if writecount := 1 then P(r);
  V(mutex2);
  P(w);
    ...
    ⟨actual writing⟩
    ...
  V(w);
  P(mutex2);
    writecount := writecount - 1;
    if writecount := 0 then V(r);
  V(mutex2);
```

Appendix II

Implementation of the Scheduled Data Base by the Class Concept

```
class readers-writers-data-base(db: data-base);
    comments this class takes a data-base as an argu-
    ment;
  begin
      rws: readers-writers-scheduler;
    comments an instance of monitor was declared;
    proc READ;
        begin
          rws.startread; comments monitor call;
          ⟨...actual reading of db...⟩
          rws.endread; comments monitor call;
        end
    proc WRITE;
        begin
          rws.startwrite; comments monitor call;
          ⟨...actual writing of db...⟩
          rws.endwrite; comments monitor call;
        end
  end
```

(Received June 12, 1978; revised November 17, 1978)