On Proof Rules for Monitors

J. M. Adams and A. P. Black

An inadequacy is pointed out in the original proof rules for monitors and in later extended rules. This inadequacy gives rise to an anomaly in proving the invariant for a monitor simulating a counting semaphore. New proof rules are proposed and used to give a sound proof of the invariant.

Key Words and Phrases: concurrency, monitor, operating system, proof rules, semaphores

CR Categories: 4.32, 4.35, 5.24

Authors' addresses: J. Mack Adams, Computer Science Department, New Mexico State University, Las Cruces, New Mexico 88003; Andrew P. Black, Department of Computer Science, University of Washington, Seattle, Washington 98195.

I. Introduction

The concept of a monitor as a method for ensuring mutual exclusion was introduced by Brinch-Hansen [2]. Hoare [5] developed the idea and introduced condition variables as a method of releasing the mutual exclusion while a process was waiting for a resource. A monitor may contain several condition variables, each associated with a different resource. The operation cond.wait is invoked when a process requiring the resource associated with the condition variable cond finds that it is not available. When another process releases or produces some of the resource it performs the cond.signal operation. Hoare described the operation of wait and signal by associating with each condition a queue of processes currently waiting for it. He then derived proof rules to formalize the semantics.

Howard [6] noted an inadequacy in Hoare's proof rules and proposed extended rules. He demonstrated a monitor which simulated a counting semaphore with a constant of zero. This monitor, which also appears in Turski [7], is given below:

```
monitor semaphore;
var
    na,np,nv : integer;
    cond : condition;

procedure P;
begin
  na := na+1;
  if na > nv then cond.wait;
  np := np+1
end P;

procedure V
begin
  nv := nv+1;
  if na > np then cond.signal;
end V;

begin (initialization)
  na := 0;
  np := 0;
  nv := 0
end semaphore
```

Neither Howard's nor Hoare's rules can be used to prove that this monitor does in fact satisfy the semaphore invariant of Habermann [4], which is accepted as defining the semantics of the semaphore. Howard justified his proof by specific assumptions, but did not generalize beyond this one example. This paper examines the inadequacy of the existing rules and

proposes new rules which can be used to prove the monitor semaphore.

2. Proof Rules

The proof rules given by Hoare [5] with a slightly alterated notation are:

(J) cond.wait (J & B) and (J & B) cond.signal (J)

where J is the monitor invariant and B is an assertion which describes the condition under which a process waiting on cond should be resumed.

The extended rules given by Howard [6] are:

(J & E) cond.wait (J & B) and (J & B) cond.signal (J & E)

where E is an additional assertion which ensures that no B is satisfied unless the corresponding queue is empty.

In Howard's monitor na, np, and nv count attempted P operations, successfull P operations, and V operations respectively. The monitor should satisfy Habermann's invariant for semaphores, which in this case reduces to:

$$np = min(na, nv)$$

We will call this expression J. A proof that J is an invariant of the monitor involves proving that J is true after the initialization and that it is restored by the action of the monitor procedures. Howard's flawed proof is essentially as follows:

```
monitor semaphore;
var
    na,np,nv : integer;
    cond : condition;

procedure P;
begin
    ( np = min(na,nv) )
    na := na+1;
    ( np = min(na-1,nv) )
    if na > nv then ( np = nv  &  np < na )
                    cond.wait
                    ( np < na  &  np = nv-1 );
    ( np = min(na,nv)-1 )
    np := np+1
    ( np = min(na,nv) )
```

```
end P;

procedure V;
begin
    { np = min(na,nv) }
    nv := nv+1;
    { np = min(na,nv-1) }
    if na > np then { na > np  &  np = nv-1 }
                    cond.signal
                    { np = min(na,nv) };
    { np = min(na,nv) }
end V;

begin (initialization)
    na := 0;
    np := 0;
    nv := 0
    { np = min(na,nv) }
end semaphore
```

Both the original and extended proof rules have J & B as the precondition for cond.signal. Clearly, the invariant J does not hold prior to cond.signal, so the proof cannot be justified on the basis of either set of proof rules.

Hoare's informal semantics for the cond.signal operation make it clear that if there are no processes waiting on the queue of a particular condition, then signaling that condition is effectively a null operation; the signaling process simply continues. In this case the only way of achieving J as a postcondition of signal is to require it as a precondition. If there is a process waiting on the queue (a situation characterized by Hoare's predicate cond.queue) then control must pass to that process. Since this process is waiting for B to become true, B must be a precondition of cond.signal in this case. However, the monitor invariant J is not a precondition; the signalling process is not resumed until the waiting process has completed the monitor procedure, and thus restored J.

Thus we see that Hoare's precondition was too strict in two senses. It required J in all cases rather than just when the queue was empty, and it required B in all cases rather than just when there was a process waiting on the queue. A summary of the permissable preconditions of cond.signal and their semantics follows:

B & cond.queue & J	resume the waiting process
B & cond.queue & not J	resume the waiting process
B & not cond.queue & J	null operation

not B & not cond.queue & J null operation

The remaining four combinations are not permissable. The weakest prcondition for cond.signal is the disjunction of the above, which simplifies to:

(B & cond.queue) or (J & not cond.queue)

Note that since J is no longer a precondition of signal when a process is resumed, it cannot be assumed as a postcondition of wait. Thus we obtain the following new proof rules:

{ (B & cond.queue) or (J & not cond.queue) } cond.signal { J }

{ J } cond.wait { B }

The advantage of our rules is the weaker precondition of the rule for signal. This allows the rules to be applied to more programs: in particular they can be applied to the semaphore monitor.

In the procedure V of the semaphore, cond.queue is represented by na > np: clearly, if na=np all the processes which called the P procedure have completed it, so none are waiting. As in Howard's proof, we let B be np < na & np = nv-1. In the then part of the conditional in V, we have cond.queue & B (see Howard's proof given above). Thus the new proof rule for cond.signal applies and we may assume J as the postcondition. This completes the proof for the V procedure, since the rest of Howard's proof is sound.

In the P procedure, J is implied by na > nv and the postcondition of na := na+1, and thus we have established the precondition of cond.wait. The rest of the proof is straightforward.

3. Simplifying the semaphore monitor

Our asymmetric proof rules help to illustrate that signal has two possible actions, whereas wait has only one. However, in the above semaphore monitor, signal only performs one of these actions. Since B & cond.queue always holds before cond.signal in procedure V, cond.signal never acts as a null operation but always causes resumption of a waiting process.

Interestingly, the V procedure will still perform the same function if the test na > np which guards cond.signal is omitted. The test of cond.queue which takes place as part of the signal operation is exactly equivalent. The revised procedure becomes:

```
procedure V ( J ≡ np = min(na,nv) );
begin
    ( np = min(na,nv) )
    nv := nv+1;
    ( Q ≡ np = min(na,nv-1) )
    cond.signal;
    ( J ≡ np = min(na,nv) )
end V;
```

Note that the precondition of cond.signal is satisfied since

Q & cond.queue ≡ Q & na>np => na>np & np=nv-1 ≡ B & cond.queue

and

Q & not cond.queue ≡ Q & na ≤ np => np=na & nv-1 ≥ na => J

In the revised procedure V, the variable np is now redundant and may be removed. It is what Clint [3] calls a "mythical variable", although "assertional variable" would be more suggestive.

4. The Role of B

In Section 2 we repeated the conventional view that B is an assertion which describes the condition under which a process waiting on cond should be resumed [5], [6], [7]. This is indeed true in the sense that a process released from a wait will find B satisfied. However, the role of B has changed a little from that envisioned by the above referenced authors.

In [5] Hoare disucsses a bounded buffer monitor and gives an implementation using two condition variables, NotEmpty and NotFull, which indicate that the buffer is in the appropriate state. Clearly, before appending to the buffer it may be necessary to wait for the NotFull signal; before removing a message it may be necessary to wait for the NotEmpty signal. Figure 1 shows an implementation of such a buffer with a proof that the invariant is maintained. The implementation uses abstract sequences: UnitSequence(x) is a new sequence containing x as its only element, and + denotes sequence concatenation. A and R appear only in assignments to themselves, and need not be implemented: they are assertional variables like np.

The assertions R1 and R2 formally describe the conditions associated by Hoare with NotFull and NotEmpty. But the proof cannot be completed using, say, R1 for B1, the B associated with NotFull. In order to obtain (J & Buffer.length < N) at line 17, B1 must be J & R1, not just R1. Similarly, B2 must be J & R2.

With our proof rules the only information available to a process which has just been released from a wait is the truth of the appropriate B. So B must be strengthened to include not just the "resource condition" R but also any other information necessary to complete the proof. In the bounded buffer example Hoare's rules already supplied this information: by demanding J & R as a precondition of signal they were able to supply it as a postcondition of wait. In general, our rules can be used in any proof which relies on Hoare's rules simply by strengthening B so that it implies J.

The semaphore example cannot be proved using Hoare's rules because the J & R precondition they demand is too strong. R is nv > np, the condition that the number of calls on V exceed the number of completed calls on P. The proof in section 2 uses B = (np < na & np = nv - 1): no weaker B will suffice to establish J at the conclusion of the P procedure.

Thus we see that the choice of B is quite critical, like the choice of a loop invariant. The resource invariant R is usually easy to identify from the problem definition. Certainly B must imply R, but its exact composition may be difficult to determine.

5. Howard's Extended Proof Rules

Howard's motivation for proposing his extended rules [6] was that Hoare's rules allow a process which could legally continue to be left waiting. He recognized two situations when this could occur:

1) A procedure which makes available a resource for which a process is waiting might not signal the appropriate condition.

2) A process may execute a cond.wait even though the resource associated with cond is available.

As Howard states, the first situation can be prevented by the introduction of an additional predicate E (to be true at monitor exit) which allows a resource to be available only if there are no processes waiting for it. In the notation introduced above, E is the conjunction of (not R or not cond.queue) for each condition in the monitor. Howard required that E be true whenever the invariant J must be true, except that after a wait and before a signal (when Hoare's rules required J & B) Howard did not require E in addition. The reason should be obvious. However, our proof rules for wait and signal have weaker conditions in those places, so E may be incorporated into J without difficulty. We do not need a revised set of rules to ensure that necessary conditions are signaled. All that is

necessary is J => E.

The second situation is not prohibited by Howard's precondition J & E. If the resource is available and the queue is empty then E is true, and provided that J is true the precondition for wait is satisfied. Executing a cond.wait in these circumstances would place a process on the queue and release the mutual exclusion. The monitor is thus exited with cond.queue & R, i.e. not E. This can be remedied by including not R in the precondition for wait, giving the rule:

(J & not R) cond.wait (B)

In the semaphore example, R is nv > np. The precondition of cond.wait is np = nv & np < na. This clearly implies not R as well as J. Indeed, we should be worried were this not the case. Habermann's invariant was designed to preclude unnecessary waiting. In a semaphore satisfying this invariant cond.wait can only be executed when absolutely necessary. Thus we should expect to find that R is false before cond.wait.

In the bounded buffer example, J & not R1 is true on line 14 after the then: the if statement ensures this. Similarly, J & not R2 is true on line 31 after the then. Thus the proof of the bounded buffer can be justified with our revised rule.

If not R1 were not required as a precondition of NotFull.wait, the test Buffer. length = N could be omitted entirely. The proof would be valid but any use of the buffer would lead to deadlock because the append procedure would wait unnecessarily.

6. Conclusion

Hoare's proof rules for monitors seem not to represent adequately the semantics of the wait and signal operations. The following rules are stronger and appear to capture more of the semantics:

((B & cond.queue) or (J & not cond.queue)) cond.signal (J)

(J) cond.wait (B)

The invariant, J, of the monitor must be true after the monitor is initialized and must be restored by every monitor procedure. The assertion B describes the condition under which a process waiting on cond should be resumed, and cond.queue is true if there is at least one process waiting on cond.

These rules may be used to prove any monitor which can be proved by the rules of Hoare. In addition, they can be used to prove other monitors; the semaphore monitor is an example.

The extended rules of Howard which attempt to prevent unneccessary waiting also seem inadequate. The following extension of our wait rule together with a more stringent invariant appear to accomplish that goal more effectively:

(J & not R) cond.wait (B)

where R affirms that the resource associated with cond is available, B => R, and J => (not R or not cond.queue) for each condition in the monitor.

One detracting feature of the rules proposed in this paper is the informal use of the predicate cond.queue. In Section 2 we stated that within the V procedure na > np = cond.queue. However, this claim cannot be substantiated formally, although it is necessary for the formulation of the "correctness criteria" [3]. Without a correspondence between cond.queue and the variables in J and B our rule is no more powerful than Hoare's.

The rule of assignment does not help us to establish the meaning of cond.queue because a cond.wait in one procedure affects the value of cond.queue in other textually unrelated ones. This seems to be a deficiency in the proof system. Perhaps a more powerful approach such as the concept of "cooperation between proofs" [1] is required.

References

[1] Apt, K.R., Francez, N. and de Roever, W.P. A proof system for communicating sequential processes. Trans. Prog. Lang. Syst. 2, 3 (July 1980), 359-385.

[2] Brinch-Hansen, P. Operating System Principles. Prentice-Hall, Englewood Cliffs, N. J. 1973.

[3] Clint, M. Program proving: coroutines. Acta Inf. 2, 50-63 (1973).

[4] Habermann, A.N. Synchronization of communicating processes. Comm. ACM 15,3 (March 1972) 171-176.

[5] Hoare, C.A.R. Monitors: an operating system structuring concept Comm. ACM 17,10 (Oct. 1974), 549-557.

[6] Howard, J. H. Proving monitors. Comm. ACM 19, 5 (May 1976), 273-279.

[7] Turski, W.M. Computer Programming Methodology. Heyden & Sons, London. 1978.

```
1   monitor bounded buffer(N : PositiveInteger);
       { Invariant J ≡ (A = R + Buffer & Buffer.length ≤ N) }
    var
       A : sequence of message;
5      R : sequence of message;
       Buffer : sequence of message;
       NotFull : condition  { R1 ≡ (Buffer.length < N) };
       NotEmpty : condition  { R2 ≡ (Buffer.length > 0) };

10  procedure append(value x : message);
       { append(x) simulates A := A + UnitSequence(x) }
    begin
       { J }
       if Buffer.length = N then { J & not R1 }
15                              NotFull.wait;
                                { B1 }
       { J & Buffer.length < N }
       A := A + UnitSequence(x);
       { A = R + Buffer + UnitSequence(x) & Buffer.length < N }
20     Buffer := Buffer + UnitSequence(x);
       { J & Buffer.length > 0 }
       { J & B2 }
       NotEmpty.signal
       { J }
25  end append;

    procedure remove(result x : message );
        { remove(x) simulates R := R + UnitSequence(x) }
    begin
30      { J }
        if Buffer.length = 0 then { J & not R2 }
                                  NotEmpty.wait;
                                  { B2 }
        { J & Buffer.length > 0 }
35      x := Buffer.first;
        Buffer := Buffer.rest;
        { A = R + UnitSequence(x) + Buffer & Buffer.length < N }
        R := R + UnitSequence(x);
        { A = R + Buffer & Buffer.length < N }
40      { J & B1 }
        NotFull.signal
        { J }
    end remove;

45  begin {initialization}
        { true }
        A := NullSequence;
        R := NullSequence;
        Buffer := NullSequence
50      { J }
     end bounded buffer
```

Figure 1