Proving Monitor Proof Rules

Leif S. Nielsen and Andrew P. Black

Department of Computer Science
University of Washington
Seattle, Washington 98195

Technical Report 85-08-01
August 6, 1985

Abstract

Various authors have proposed differing sets of Axiomatic Proof Rules for Monitors. Which of these sets of rules are correct? What are the relationships between them? To answer these questions we need a description of the monitor that is both formal enough to permit the derivation of the proof rules and yet intuitive enough to correspond clearly with our informal understanding of monitors.

DMC is a new descriptive tool for concurrent modules that meets these goals. The central idea is the separation of Data Manipulation and Control. A monitor can be translated into a DMC module in which the semantics of *signal* and *wait* is given by the Control description. From the use of DMC we derive a new set of proof rules that is strictly more powerful than any that has been previously published. We are also able to clarify the role of the various state predicates in these rules.

Nielsen is supported in part by an IBM Graduate Fellowship.
Black is supported in part by the National Science Foundation under Grant No. MCS-8004111.

1 Introduction

The monitor is an important concurrent programming language concept [3]. It combines the idea of an abstract data type in a sequential language with the idea of a critical section in a concurrent language. A monitor encapsulates a set of related data variables and permits them to be accessed only through procedure calls. Additionally, only one process is permitted to execute inside the monitor at any one time (mutual exclusion). That process maintains exclusive access to the monitor until it reaches a control point where the monitor variables are in a consistent state.

A *monitor invariant* is a consistency assertion on the monitor's variables. If each procedure execution is allowed to proceed from entry to exit without giving up the exclusive access to the monitor variables, we may formulate a simple set of proof rules for maintaining the monitor invariant. The initialization must establish the invariant; each procedure execution may assume the invariant to be true at the entry point and must reestablish the invariant at the exit point. If the monitor invariant is J, and the state predicate *init* describes the state after the initialization, we can formulate these proof rules as follows:

$$init \Rightarrow J$$
$$\{J\}\ procedure\ body\ \{J\}$$

for any monitor procedure with body *procedure body*.

Using a dynamic resource allocator as an example, Hoare [3] points out that a procedure execution may have to wait for the resource to become available. Thus, there is a need for an operation to *wait* for a resource, and a *signal* operation to wake up a waiting process. Hoare introduces the *condition variable* as a synchronization mechanism. If *cond* is a condition variable associated with a particular resource, then a process may relinquish its exclusive access to the monitor by executing a *cond.wait*. When another procedure execution has established that the resource is available, it executes a *cond.signal*. If one or more processes are waiting on *cond* at this time, the signaling process will be suspended and one of the waiting processes will be allowed to continue. If no process is waiting, then the signaler may proceed. Suspended signalers and processes waiting to enter the monitor are *ready*. When a process executes a *cond.wait* or exits from a procedure, one of the ready processes may continue.

We shall use R for the *resource condition* associated with a condition variable *cond*. R is an assertion on the monitor's variables which describes the condition under which the resource associated with *cond* is available. Using this notation, Hoare's proof rules for *signal* and *wait* are:

$\{J\}$ *cond.wait* $\{J \wedge R\}$
$\{J \wedge R\}$ *cond.signal* $\{J\}$

The invariant J must be true before a *wait* operation since the next ready process to execute may be a new caller, which needs the invariant. The ready process could also be a suspended signaler; thus, the postcondition for *signal* is also the invariant. The resource condition R should clearly be true when control is transferred from a signaler to a waiting process. However, although the rules require that the invariant should be true before a *signal* that will transfer control, correct monitors can be written in which this is not the case. Similarly, the rules unnecessarily require that the resource condition should be true when there is no process waiting at the time of the *signal*. While simple and intuitive, the proof rules in [3] may often be too weak to prove a monitor invariant. An example is the monitor implementation of a semaphore [1, 4].

Two modifications have been made to these rules. Howard [4] introduces the convention that there should be *no unnecessary waiting*, i.e. that there should be no process waiting on a condition variable when the resource condition is true and no other process is executing. As an example, a procedure execution should not exit with R true if there are waiting processes. Proof rules are given for enforcing this property (see section 4). These rules avoid the above problems by requiring that there must be a process waiting on a condition variable before it can be signaled.

Adams and Black [1] address the problem of reflecting the semantics of condition variables more closely. The main idea is to introduce a signaling condition B which will be assumed as the postcondition for *cond.wait*. In the precondition of *signal*, B is assumed when there are waiting processes, while the invariant is assumed when there are no waiting processes (see section 4).

Both of the modified proof rule sets refer to the number of processes waiting on *cond*. If the assertions in the proof rules are allowed to refer to this control information, it is necessary to be careful about the pre- and postconditions for *cond.wait*. Howard [5] shows how this can be done.

Howard [5] also gives proof rules for a number of alternative implementations of condition variables. We consider the simplest possible implementation in which the *signal* results in an immediate transfer of control to a waiting process, and the suspended signaler goes back to the pool of ready processes. This is the "Signal and Wait" semantics in [5].

This paper clarifies the relationship between these sets of rules using a new descriptive tool, DMC. This is the basis of a general specification technique for concurrent modules; in this presentation, however, we restrict ourselves to the description of the semantics of a monitor. This monitor description is formal, yet intuitive enough to establish a clear correspondence between it and our informal understanding of monitors. Using DMC, we derive a new set of proof rules that is strictly stronger than any of those previously suggested. The monitor description in DMC provides a simple medium for experimenting with different definitions of monitors and with different ideas for monitor proof rules.

2 DMC: A model of concurrent modules

DMC is a concrete descriptive tool for concurrent modules currently under development at the University of Washington [7]. The central ideas are the separation of **Data Manipulation** and **Control** (giving the acronym DMC), and the use of familiar underlying concepts such as modules, procedures, and control points. This presentation does not attempt to cover all the features of DMC; we restrict our attention to those features that are necessary to describe monitors. Specifically, we will consider modules which receive and execute procedure calls but do not initiate calls to other modules.

A system in DMC consists of a fixed number of named modules; each module description consists of a *Data Manipulation* description and a *Control* description.

The **Data Manipulation** description declares some initialized data variables and some exported procedures that may manipulate these variables. Each procedure has a number of control points, including an entry point and an exit point, and may also have some local variables. The body of the procedure defines atomic *data transitions* that occur as the procedure execution moves from one control point to the next.

The following DMC procedure definition is used in section 3. It is the P operation of a semaphore implementation translated from a monitor into DMC. The variable s is a global data variable.

```
procedure P
entry:
    if $s = 0$ then $cw$:
    $s := s - 1$
exit:
```

We represent the control points by labels using an Algol-like notation. The internal control point cw represents the *cond.wait* in the monitor implementation. In general, a DMC control point is used to represent a point in the execution where interaction with other procedure executions or external communication may take place. In our examples, the data transitions are programmed in an imperative style using variables and assignments. In principle, the data transitions are defined by a function which takes as input the state of the module's variables, a control point,

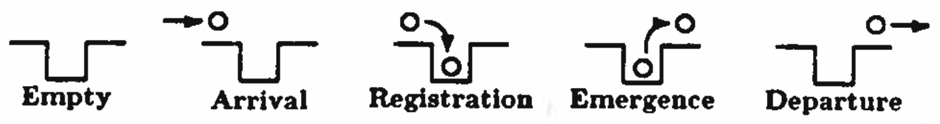


Figure 1: A control point and its use

and the state of the local variables of a procedure execution (at this control point); the data transition function returns a new global data state, a new control point, and a new local data state for the procedure execution. In particular, the choice of the new control point may depend on the global and local data state: local control is dynamic. In procedure P the value of s decides whether the procedure execution moves from *entry* to *cw* or *exit*.

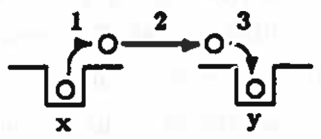
In our model the control points are represented by buckets (⊔). The execution of a procedure is represented by the movement of a *pebble* (∘) from bucket to bucket; the local variables of the procedure are written on the pebble. Each bucket is equipped with two shelves. The left-hand shelf (¬) is called the *register shelf*; it is occupied by arriving pebbles before they are *registered* (moved down into the bucket). The right-hand shelf (⌈) is called the *emerge shelf*; it is occupied by a pebble that has *emerged* from the control point. A control point and its use is illustrated in figure 1.

The **Control** description defines *enabling conditions* for control points, declares some control variables, and associates atomic transitions on the state of these variables with registration and emergence. These transitions are called *control transitions* and are programmed in an Algol-like notation. Some control variables are pre-declared: the *control point counter* $|x|$ indicates the number of pebbles in the bucket for control point x, i.e. registered pebbles.

The enabling condition b for control point x is denoted by $b \rightarrow x$, where b is a boolean expression referring to control variables. *Selection* of a pebble from a bucket may take place only when the enabling condition for the corresponding control point is satisfied. When more than one pebble can be selected, the choice is made non-deterministically. Following the selection, the pebble will emerge from the control point. Figure 5 shows the complete Control description of a DMC module.

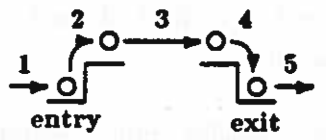
The movement of a pebble from one bucket to another is shown in figure 2; we refer to this multiple transition as the *triple-transition*.

The triple-transition is indivisible; once a pebble has emerged from x, no other pebble is moved until this pebble is registered at the next control point y. The emergence and registration transitions always include the updating of the control point counters; naturally, these counters may not be updated explicitly.



1. Emergence (control transition, includes $|x| := |x| - 1$)
2. Data transition
3. Registration (control transition, includes $|y| := |y| + 1$)

Figure 2: Moving a pebble from control point x to control point y



1. Registration of call
2. Emergence of call
3. Data transition
4. Exit registration
5. Return transition

Figure 3: A simple procedure execution

The Data Manipulation description will refer only to global data variables and the local variables of a procedure; the Control description will refer only to control variables. This restriction is the main way in which we separate Data Manipulation from Control.

A module computation takes place as follows. Initially, all buckets and shelves are empty. A call of an exported procedure is represented by the registration of a pebble at the procedure's entry point. The entry point has no register shelf (a *destination* control point). The arguments of the call are written on the pebble. Following the registration of the pebble at the exit point, the pebble returns to the caller; the results are written on the pebble. The exit point has no emerge shelf (a *source* control point).[1] The exit registration increments the exit counter, but the return transition does not decrement this counter. The exit counter thus acts as a history variable recording the number of completed procedure executions.

Figure 3 represents the execution of a simple procedure with no internal control points.

[1]At the caller side, the call is represented by two control points: a call point (source control point) and a return point (destination control point).

Because the triple-transition is indivisible, there will be at most one pebble on a shelf. A state in which there is a pebble on a shelf is termed *private*; states in which there is no such pebble are *public*. An external observer may think of the module computation as a sequence of public states. The public state is changed by call registrations and internal transitions (triple-transitions). The module programmer is also concerned with the intermediate private states.

Call registration may occur in any public state. The call registration transition may have programmer defined actions.

For reasoning about the module (e.g. for giving a module specification), we introduce state predicates to describe the four states in the triple-transition. The symbols $\lrcorner$, $\ulcorner$ etc. can be imagined to be part of the bucket for the control point, with the pebble on the horizontal line.

$\lrcorner x$:	control point x is enabled (public state)
$\ulcorner x$:	a pebble on the emerge shelf of x (private state)
$\urcorner y$:	a pebble on the register shelf of y (private state)
$\llcorner y$:	a pebble has just been registered at y (public state)

These predicates are not part of the DMC algorithmic notation, e.g. they cannot be used in enabling conditions or in the tests of if/while statements. $\lrcorner x$ implies that there are pebbles at x and that x is enabled, but does not imply that one of these pebbles will be selected. $\llcorner y$ is true only until a new pebble is selected or a call is registered. Registration of a call of p has postcondition $\llcorner p.entry$. $\llcorner p.exit$ remains true until a new pebble is selected or a call is registered. The return transition will not affect the value of this predicate.

Other predicates can be derived from these four state predicates. For example, the predicate *public* is true in all public states. The predicate *init* is true in the initial public state where all data and control variables have been initialized and where no call has been registered.

In section 3 we describe how a monitor can be translated into a DMC module. In section 4 this DMC module is used to derive a new set of proof rules. The above state predicates are important for this derivation.

3 Monitor semantics using DMC

The essential properties of a monitor as defined by Hoare [3] are the encapsulation of the monitor's global data variables, the mutual exclusion of procedure executions, and the use of condition variables. Figure 4 shows a monitor implementation of a semaphore.

```
monitor sem

    var
        s : integer := 0
        cond : condition

    procedure P
    begin
        if s = 0 then cond.wait
        s := s − 1
    end

    procedure V
    begin
        s := s + 1
        cond.signal
    end

end sem
```

Figure 4: Monitor implementation of a semaphore

We now describe how a monitor can be translated into a DMC module. For simplicity, we shall model a monitor with a single condition variable and two exported procedures (a generalization requires some extra notation). Procedure W does a single *cond.wait*, and procedure S does a single *cond.signal*. The *cond.wait* is represented by a control point cw, and the *cond.signal* is represented by control point cs. The monitor template is shown in figure 5.

The monitor code is used to fill out the Data Manipulation description. The control points are represented as labels. The mutual exclusion of procedure executions is achieved by grouping monitor statements into a single data transition which is part of an indivisible triple-transition.

The semantics of *signal* and *wait* is given by the Control description. The condition variable is not one of the Data Manipulation variables in the template; instead a control variable *transfer* is used to remember a *signal*. According to our semantics, a *signal* on an empty condition variable may still give up control to a new call or another suspended signaler. Suspended signalers could be given priority over new calls by disabling the entry points when there are suspended signalers; this can be accomplished simply by putting $|cs| = 0$ in the enabling conditions for the entry points. If we wanted a *signal* on an empty condition variable to be a no-op, we could introduce the ability to enable particular procedure executions rather than control points; i.e. by putting a mark on the procedure execution during the registration at

```
module MonitorTemplate

Data Manipulation

    var
        ...

    procedure W
    entry: ..
    cw: .. { cw represents cond.wait }
    exit:

    procedure S
    entry: ..
    cs: .. { cs represents cond.signal }
    exit:

Control

    var
        transfer : boolean := false

    register cs:
        if |cw| > 0 then transfer := true

    emerge cw:
        transfer := false

    enable
        ¬transfer → W.entry
        ¬transfer → S.entry
        ¬transfer → cs
        transfer → cw

end MonitorTemplate
```

Figure 5: Monitor template in DMC

cs, and use this mark in the enabling condition.[2] Alternatively, *cond.signal* could be translated into: **if** $|cw| > 0$ **then** *cs*:. However, this tranlation would obscure the clean mapping of programmer provided code to Data Manipulation description and monitor implementation code to Control description, and it would also introduce a control variable in the Data Manipulation description.

If the monitor in figure 4 were translated to DMC, the Data Manipulation description would look like this:

```
var
    $s : integer := 0$

procedure P
entry:
    if $s = 0$ then cw:
    $s := s - 1$
exit:

procedure V
entry:
    $s := s + 1$
    cs:
exit:
```

The monitor template of figure 5 is used in the next section to derive a new set of proof rules. The most important property of the monitor template is that *transfer* is true only when control is being transferred from a signaler to a waiting process.

4 Monitor proof rules using DMC

The usual verification technique for a monitor uses an *invariant*; this invariant is established initially, and is preserved by each monitor procedure execution. The tricky part of the verification involves *signal* and *wait*. Since the *signal* may include a transfer of control, the precondition of the *signal* must imply the postcondition of the *wait*. On the other hand, a *signal* on an empty condition variable may be a no-op; in our monitor template in figure 5 we allow any ready process to continue. Therefore, when no process is waiting on the condition variable the precondition of the *signal* must imply both the postcondition of *signal* and the precondition of each procedure.

In the following we shall assume a monitor invariant I and a signaling condition B which is the postcondition for *wait*. B implies that the resource (corresponding

[2] In the general version of DMC, the enabling condition may refer to the local variables of the procedure execution; the enabling condition then becomes a condition for selecting a particular procedure execution at a given control point.

to the condition variable) is available. So, using R for the resource condition, we have $B \Rightarrow R$. Some proof rules also require $B \Rightarrow I$; without this requirement, B should be strong enough to reestablish I again before procedure exit. It is the monitor programmer's responsibility to come up with meaningful Is and Bs. We (the formulators of the proof rules) will make some restrictions on these predicates.

Using a set of monitor proof rules, the monitor programmer may assume certain preconditions for the data transitions and must then establish the corresponding postconditions. As an example, in programming the data transition from an entry point to a *signal*, the programmer may assume the precondition for the procedure and must then establish the precondition for *signal*, which is actually the postcondition for the data transition. In terms of our model, the proof rules are presented to the programmer as the private pre- and postconditions for the data transitions; that is, they impose restrictions on the two intermediate states in the triple-transition in figure 2. The initialization is the common responsibility of the monitor programmer and the implementation of the monitor.

We shall use the monitor template in figure 5 to derive proof rules. We shall first state a simple set of rules describing all public states. The state predicates *init* and *public* are described in section 2. The rules are:

Pub_0:	$init \Rightarrow I$
Pub_1:	$public \wedge \neg transfer \Rightarrow I$
Pub_2:	$transfer \Rightarrow B$

Comments on the Pub-rules: The predicate *init* is true in the initial public state where all variables have been initialized; from $init \Rightarrow public \wedge \neg transfer$ it follows that Pub_0 can be derived from Pub_1. *transfer* becomes true during a *cs* registration which results in a public state. Two state transitions may occur when *transfer* is true: a call registration may occur in any public state, and a *cw* emergence has *transfer* as enabling condition. The first maintains the public state, while a *cw* emergence makes *transfer* false and results in a private state. Therefore $transfer \Rightarrow public$, and Pub_2 is a public state predicate.

The Pub-rules express the essential ideas of I and B. We shall now translate these rules into a set of predicates describing all private states. This will lead us one step closer to the normal formulation of monitor proof rules.

Let us assume that I and B refer to the data variables and to the counter $|cw|$. We will allow the predicates to refer neither to any other counters, nor to the special DMC predicates describing private and public states, because those predicates do not make sense to the monitor programmer. In other words, the monitor programmer is allowed to reason about the number of waiting processes at *cond.wait* ($|cw|$), but not about any other control information.

We now present some rules describing the private states, and then prove that these Pri-rules can be derived from the Pub-rules. We shall use the private state predicates presented at the end of section 2. Using the two control transitions in figure 5, and remembering the update of a control point counter in each control transition (figure 2), we get:

	{ Data Manipulation }	{ Control }	
Pri_0:	$init \Rightarrow I$		shared responsibility
Pri_1:		$\ulcorner entry \Rightarrow I$	for each procedure
Pri_2:	$\urcorner exit \Rightarrow I$		for each procedure
Pri_3:	$\urcorner cw \Rightarrow I^*$		
Pri_4:		$\ulcorner cw \Rightarrow B^*$	
Pri_5:	$\urcorner cs \Rightarrow (B \wedge \lvert cw\rvert > 0) \vee (I \wedge \lvert cw\rvert = 0)$		
Pri_6:		$\ulcorner cs \Rightarrow I$	

where X^* is obtained by replacing all occurrences of $|cw|$ in X by $|cw| + 1$.

Given the semantics of figure 5 we will prove that the Pri-rules can be derived from the Pub-rules. The main observation is that *transfer* becomes true during a *cs* registration and remains true until the next emergence which must be a *cw* emergence.

- Pri_0 is the same as Pub_0; Pri_0 can be inferred from Pri_1.

- Pri_1, Pri_2, and Pri_6 follow from Pub_1 since I must be true in the public state preceding an *entry* or *cs* emergence, and I must be true in the public state following an *exit* registration.

- Pri_3 follows from $\llcorner cw \Rightarrow public \wedge \neg transfer \Rightarrow I$, and the usual axiom for assignment $\{I^*\}\ |cw| := |cw| + 1\ \{I\}$. Pri_4 similarly follows from $\lrcorner cw \Rightarrow transfer \Rightarrow B$, and $\{B\}\ |cw| := |cw| - 1\ \{B^*\}$.

- Pri_5 follows from $\urcorner cs \Rightarrow \neg public \Rightarrow \neg transfer$ which together with the *cs* registration gives:

$$\llcorner cs \Rightarrow (transfer \wedge |cw| > 0) \vee (\neg transfer \wedge |cw| = 0)$$

  We then use Pub_1 and Pub_2 to introduce I and B instead of *transfer*, and finally notice that the *cs* registration does not change I, B, and $|cw|$.

We have now proved that the Pri-rules follow from the Pub-rules. In general it is true that if we have a set of predicates describing all the public states, we can derive a complete set of private state predicates, and vice versa. In the above transformation from Pub-rules to Pri-rules we have not lost any information. We could derive the Pub-rules from the Pri-rules using similar arguments. The only underlying assumption is that a call registration will not make I or B false; this follows from the restriction that I and B do not refer to any entry counter.

A more traditional formulation of the private proof rules is:

M_{10}: $\mathit{init} \Rightarrow I$ (from Pri_0)
M_{11}: $\{I\}\ \mathit{procedure\ body}\ \{I\}$ (by combining Pri_1 and Pri_2)

M_{12}: $\{I^*\}\ \mathit{cond.wait}\ \{B^*\}$ (by combining Pri_3 and Pri_4)
M_{13}: $\{(B \wedge K) \vee (I \wedge \neg K)\}\ \mathit{cond.signal}\ \{I\}$ (by combining Pri_5 and Pri_6)

The predicate K in M_{13} is equivalent to $|cw| > 0$. The counter $|cw|$ corresponds to the number of waiting processes (at *cond.wait*). In existing monitor languages, the programmer is normally provided with a predicate which indicates that there are waiting processes (e.g. $K \equiv \mathit{cond.queue}$).

We can now discuss some of the monitor proof rules that have been suggested in the past. Adams and Black [1] introduce the signaling condition B and the disjunctive precondition for *signal*. They do not use the asterisk to formalize the use of the number of waiting processes. Their rules are the M_1-rules leaving out the asterisk in M_{12}.

Howard [4, 5] introduces two extra requirements on the programmer. First, he requires that *signal* is only done when there are waiting processes. A predicate Q is used to ensure that there are waiting processes and that the resource is available. Second, he requires that no process should be left waiting at a *cond.wait* if the resource is available and the monitor is idle (that is, no process is executing). The consistency of the monitor variables are described by the invariant J. E is introduced to give the extra "no unnecessary waiting" property. Our interpretation of Howard's rules is:

M_{20}: $\mathit{init} \Rightarrow J \wedge E$
M_{21}: $\{J \wedge E\}\ \mathit{procedure\ body}\ \{J \wedge E\}$

M_{22}: $\{(J \wedge E)^*\}\ \mathit{cond.wait}\ \{(J \wedge Q)^*\}$
M_{23}: $\{J \wedge Q\}\ \mathit{cond.signal}\ \{J \wedge E\}$

M_{24}: $J \wedge E \Rightarrow \neg R \vee \neg K$
M_{25}: $J \wedge Q \Rightarrow K$
M_{26}: $J \wedge Q \Rightarrow R$

The asterisk was presented in [5]; however, in [4] it was not used, which may have caused some confusion. For example, if the postcondition of *cond.wait* is $J \wedge Q$, then by M_{25}, K holds, i.e. there are always waiting processes.

The relation between the M_1-rules and the M_2-rules are contained in: $I \equiv J \wedge E$ and $B \equiv J \wedge Q$. If *signal* is done only when K is true, then the precondition from M_{13} for *cond.signal* reduces to the precondition in M_{23}. M_{24} could also be added to the M_1-rules. However, this requirement has nothing to do with the semantics of the monitor; instead, it expresses a convention for its use. The requirement in M_{25} that *signal* is only done when there are waiting processes is not required by the M_1-rules. At the beginning of this section we mentioned $B \Rightarrow R$ as a reasonable assumption, corresponding to M_{26}.

Finally, we may point out that there is a conceptual difference between the M_1-rules and the M_2-rules. The M_2-rules require that the data invariant J is true when control is transferred from a signaler to a waiting process, while the signaling condition B in the M_1-rules has no direct relation to the invariant I. The interpretation of Howard's rules is discussed further in the exchange of letters [6] and [2].

For comparison we shall repeat the original proof rules from [3]. They refer to the data invariant J and the resource condition R.

M_{30}: $\;$ $init \Rightarrow J$
M_{31}: $\;$ $\{J\}\; procedure\ body\; \{J\}$

M_{32}: $\;$ $\{J\}\; cond.wait\; \{J \wedge R\}$
M_{33}: $\;$ $\{J \wedge R\}\; cond.signal\; \{J\}$

Compared with the M_1-rules, there is no reference to the number of waiting processes, and the signaling condition $J \wedge R$ implies both the postcondition for *wait* and the postcondition for *signal*. If we assume the relations $I \equiv J$ and $B \equiv J \wedge R$, then the precondition for *signal* in M_{13} is still weaker than $J \wedge R$, because R is not required when no process is waiting.

Of the rules we have discussed, clearly the M_1-rules are the most powerful. The rules were derived from the monitor template definition in DMC. Thus, the soundness of these rules follows from their constructive derivation. Given the equivalence with the Pub-rules it seems unlikely that a more powerful set of proof rules can be formulated based on the idea of a monitor invariant and a signaling condition. However, we did make one simplifying assumption in going from the Pub-rules to the Pri-rules. We assumed that I and B are allowed to refer to the counter $|cw|$, but not to any other counters. It would be simple to allow references to any control point counter (*entry*, *cs*, and *exit*), but it is not clear whether or not this would be a useful generalization (the phase synchronizer example below includes some generalizations). Howard [5] presents a number of alternative proof rules for monitors. However, they all reflect changes in the monitor semantics, e.g. that signalers have priority over new procedure executions.

We have set up a framework in which experimentation with the monitor semantics and with different ideas for proof rules can be carried out fairly easily.

4.1 Example: Semaphore

We shall prove that the monitor implementation of a semaphore shown in figure 4 satisfies Habermann's invariant. With the resource condition $R \equiv (s > 0)$, Hoare's rules are strong enough to prove that the data invariant $J \equiv (s \geq 0)$ is maintained.

Habermann's invariant is normally formulated $np = min\{nv, na\}$, where na is the number of attempted P calls, np is the number of completed P calls, and nv is the number of V calls (including suspended signalers). Hoare's rules are not strong enough to prove this invariant. Using our DMC template we may interpret these three entities by the relations: $na = |cw|+|P.exit|$, $np = |P.exit|$, $nv = |cs|+|V.exit|$. We also use $s = nv - np$ which is true in all of the public states where our proof rules require I. We may simplify Habermann's invariant by subtracting np on both sides of the equation to get: $min\{s, |cw|\} = 0$. This will be our I. Our signaling condition B must be strong enough to reestablish I at the end of P. $B \equiv (s = 1)$ is the weakest predicate to guarantee this requirement. The M_1-proof rules can now be used to prove the normal monitor implementation of the semaphore as shown in figure 6.

We note that $I \equiv (J \wedge (\neg R \vee \neg K))$, so I satisfies the no unnecessary waiting property introduced by Howard. We also notice that $\neg(B \Rightarrow I)$.

Our verification has not used any auxiliary variables, except the natural control point counter $|cw|$. We have not assumed that *signal* only be done when there are waiting processes, and we have formalized the use of $|cw|$. These properties illustrate that the M_1-rules are stronger than any of the previously published proof rules mentioned in this paper.

4.2 Example: Phase synchronizer

The following example uses both the *wait* counter and the *signal* counter. The M_1-rules are modified to allow reference to the counter $|cs|$. The new proof rule for *signal* is:

M'_{1s}: $\{(B^\circ \wedge K) \vee (I^\circ \wedge \neg K)\}$ *cond.signal* $\{I^\circ\}$

where X° is obtained by replacing all occurrences of $|cs|$ in X by $|cs| + 1$.

```
monitor sem
                    { $I \equiv (min\{s, |cw|\} = 0)$ }
                    { $B \equiv (s = 1)$ }

    var
        $s$ : integer := 0
        cond : condition
                    { $init \Rightarrow (|cw| = 0) \wedge (s = 0)) \Rightarrow$
                        $min\{s, |cw|\} = 0$ }

    procedure P
    begin
                    { $min\{s, |cw|\} = 0$ }
        if $s = 0$ then
                    { $(s = 0) \wedge (min\{s, |cw|\} = 0)$ }
                    { $min\{s, |cw| + 1\} = 0$ }
                    { $I^*$ }
            cond.wait
                    { $B^*$ }
                    { $s = 1$ }
        else
                    { $(s \neq 0) \wedge (min\{s, |cw|\} = 0)$ }
                    { $(s > 0) \wedge (|cw| = 0)$ }
        end if
                    { $(s = 1) \vee ((s > 0) \wedge (|cw| = 0))$ }
        $s := s - 1$
                    { $(s = 0) \vee ((s \geq 0) \wedge (|cw| = 0))$ }
                    { $min\{s, |cw|\} = 0$ }
    end

    procedure V
    begin
                    { $min\{s, |cw|\} = 0$ }
        $s := s + 1$
                    { $min\{s - 1, |cw|\} = 0$ }
                    {$((s = 1) \wedge (|cw| > 0)) \vee ((min\{s, |cw|\} = 0) \wedge (|cw| = 0))$ }
        cond.signal
                    { $min\{s, |cw|\} = 0$ }
    end

end sem
```

Figure 6: Verification of a monitor implementation of a semaphore

The program of n identical processes contains one loop. The processes must stay in phase, that is, they wait at the top of the loop until all processes have completed the current iteration; then they all do another iteration etc.

The synchronization is implemented by a call of the procedure *synchronize* in the monitor *phasesynch*. The implementation uses one variable c to count the number of waiting processes at *cond.wait* ($c = |cw|$ most of the time). The code for the procedure *synchronize* is short:

```
if $c < n - 1$ then
    $c := c + 1$
    cond.wait
else
    $c := 0$
end if
cond.signal
```

It is important that control is transferred immediately from the signaler to the waiting process, otherwise the signaler could complete its loop and join the waiting processes before the current iteration is over. The central property of this implementation is that *signal* is executed when all n processes are either at *signal* or *wait*. This is expressed in the signaling condition $B \equiv ((|cw| + |cs| = n) \wedge (c = 0))$. The invariant is $I \equiv (c = |cw|)$. This invariant does not follow from B, and therefore we need the disjunctive precondition for *signal*. The verification is shown in figure 7.

The subtle point in the verification is the **else** part of the **if** statement. Given $(|cw| \geq n-1)$ we use the fact that there are n processes. With at least $n-1$ processes at *wait* and one process executing the **else** part, all n processes are accounted for. It follows that $(|cw| = n-1) \wedge (|cs| = 0)$. It is interesting to observe that since $c < n$ is always satisfied (could be part of the invariant), we can conclude $|cw| = n - 1$ in the **else** part without knowledge of the number of processes. However, if there may be more than n processes calling the monitor, we can not prove $|cs| = 0$. In fact, the signaling condition will no longer be correct in this case, unless suspended signalers have priority over new incoming calls.

The property we want to prove is that the n processes are in phase. This follows implicitly from B. To express this property explicitly we could introduce *auxiliary* variables to count the number of iterations for each process. If l_i counts the number of iterations for process i, the increment of l_i should happen just before *signal*. The new invariant and signaling condition are restrictions of the two old ones:

```
monitor phasesynch

                    { $I \equiv (c = |cw|)$ }
                    { $B \equiv ((|cw| + |cs| = n) \wedge (c = 0))$ }

  var
     c : integer := 0
     cond : condition
                    { $init \Rightarrow (|cw| = 0) \wedge (c = 0)) \Rightarrow (c = |cw|)$ }

  procedure synchronize
  begin
                    { $(c = |cw|)$ }
     if $c < n - 1$ then
                    { $(c < n - 1) \wedge (c = |cw|)$ }
                    { $(c = |cw|)$ }
        $c := c + 1$
                    { $c - 1 = |cw|$ }
                    { $I^*$ }
        cond.wait
                    { $B^*$ }
                    { $(|cw| + 1 + |cs| = n) \wedge (c = 0)$ }
     else
                    { $(c \geq n - 1) \wedge (c = |cw|)$ }
                    { $|cw| \geq n - 1$ }
        $c := 0$
                    { $|cw| \geq n - 1) \wedge (c = 0)$ }
                    { $|cw| = n - 1) \wedge (|cs| = 0) \wedge (c = 0)$ } { because there are $n$ processes }
                    { $(|cw| + 1 + |cs| = n) \wedge (c = 0)$ }
     end if
                    { $B^\circ$ }
                    { $(B^\circ \wedge (|cw| > 0)) \vee (B^\circ \wedge (|cw| = 0))$ }
                    { $(B^\circ \wedge (|cw| > 0)) \vee ((c = 0) \wedge (|cw| = 0))$ }
                    { $(B^\circ \wedge (|cw| > 0)) \vee (I^\circ \wedge (|cw| = 0))$ }
     cond.signal
                    { $I^\circ$ }
                    { $I$ }
  end

end phasesync
```

Figure 7: Verification of a phase synchronizing monitor

$$I' \equiv I \wedge (\forall i : l_i = \min_{k \in 1..n} \{l_k\})$$
$$B' \equiv B \wedge (\forall p_i \in \mathit{cond.signal} : l_i = \min_{k \in 1..n} \{l_k\} + 1) \wedge$$
$$(\forall p_i \in \mathit{cond.wait} : l_i = \min_{k \in 1..n} \{l_k\})$$

The predicate $p_i \in \mathit{cond.signal}$ means that process p_i ($i \in 1..n$) belongs to the set of suspended signalers. In DMC terms this corresponds to saying that the pebble is in the bucket of *cs*. To verify these predicates we need a further generalization of the proof rules. Using the given formulation of the M_1-rules (including M'_{13}), the meaning of X° is now: replace each reference to the set of suspended signalers in X by the same set extended with the executing process. The definitions of $p_i \in \mathit{cond.wait}$ and $X^{\bullet}$ refer to the set of processes waiting on the condition variable. We leave the proof of the new predicates to the reader.

Howard [4] has a similar consideration for condition variables with priority. The set of waiting processes is then a queue.

The generalization from counters to sets of processes is probably the ultimate generalization of the given proof rules.

5 Conclusion

This paper has illustrated how monitor semantics and monitor proof rules can be studied using the DMC notation based on an explicit separation of Data Manipulation and Control. We have demonstrated a method by which a monitor can be translated into a DMC module in which the description of *signal* and *wait* is simple and intuitive. Based on the DMC monitor template we derived proof rules that were strictly stronger than any other proof rules that have been suggested for the same monitor semantics.

The derivation of the proof rules is unique in that we first postulated a simple set of predicates for all public states, and then derived the private state predicates which can be formulated in the usual notation of monitor proof rules. In the translation from public to private proof rules, the only underlying requirement is that the two state predicates (I and B) cannot be made false by a call registration. An alternative approach would have been to start out with a set of existing proof rules and then to translate them into the DMC notation to obtain private proof rules. These private proof rules could then be validated by using the monitor semantics (as given by the monitor template), or they could be translated into a public set of rules which were obviously true. If the proof rules were not consistent with the semantics we would get contradictions either in the validation or in the public state predicates.

The DMC approach is currently being developed by the authors [7]. One of its applications is as a specification language for concurrent modules. It is our hypothesis that the separation of Data Manipulation and Control is useful in intermediate-level specifications. A systematic DMC program development starts with a property specification of a module using a module invariant, pre- and postconditions for the procedures (safety properties), and liveness properties specified using temporal logic. Next, the safety properties are programmed as data transitions and enabling conditions. Finally, the liveness properties are programmed in the Control description. This leaves an algorithmic DMC description which can easily be translated into an existing programming language where Data Manipulation and Control are mixed.

The separation of Data Manipulation and Control and the concrete nature of DMC are useful features both when DMC is used as a specification tool and when, as in this paper, it is used as a descriptive tool.

Acknowledgement

We thank Jørgen Staunstrup and Alan Shaw for comments on drafts of this paper.

References

[1] Adams J.M., Black A.P.: On Proof Rules for Monitors. ACM Operating Systems Review **16**:2, 18-27 (April 1982)

[2] Adams J.M., Black A.P.: Letter with reply to [6]. ACM Operating Systems Review **17**:1, 6-8 (January 1983)

[3] Hoare C.A.R.: Monitors: An Operating System Structuring Concept. Commun. ACM **17**:10, 549-557 (October 1974)

[4] Howard J.H.: Proving Monitors. Commun. ACM **19**:5, 273-279 (May 1976)

[5] Howard J.H.: Signaling in Monitors. Proc. 2nd Int. Conf. on Software Engineering, pp. 47-52, IEEE, 1976

[6] Howard J.H.: Reply to "On Proof Rules for Monitors". ACM Operating Systems Review **16**:4, 8-9 (October 1982)

[7] Nielsen L.S.: Ph.D. dissertation (in preparation). Department of Computer Science, University of Washington, 1985