1 Introduction

One of the most influential papers of Computer Science to appear during the late 1960's undoubtedly was Tony's "An Axiomatic Basis for Computer Programming" [1], introducing a special purpose logic, now called "Hoare logic", for reasoning about the behaviour of imperative programs. That paper and its many followers, notably [2], have induced a change of attitude to the art of programming, in focusing on verifiability as an essential aspect of program quality, and on program development as a mathematical activity.
A concept of *monitors* important for the design of interacting concurrent processes took form a few years later, based on ideas from various sources: critical regions, [5], shared variables and mechanisms for the sequencing of critical regions, [3], [6], and object classes, [7]. The concept received its final, convincing form in [4], where a monitor is an object in the role of a shared variable, whose operators are critical regions with respect to the object, an where the internal sequencing is by *wait* and *signal* operations, which are primitives comparable to (semi-) coroutine calls. Rules are given for the reasoning about partial correctness with respect to a local invariant. It is fitting in the present context to return to the application of Hoare logic to monitors in order to look at some useful extensions compared to the logic of [4].

2 Monitors redefined

The main purpose of our redefinition is to impose a LIFO discipline on signallers, so that when a signalled process gives up the critical region, it is given to the last signaller waiting. This enables stronger state assertions at the time when a signaller regains control. We provide access to the length of process queues (using the convenient ad-hoc notation $\#c$ for *condition* variable c), not only a test for empty/non-empty. We also drop the notion of priority *condition* queues in favour of more explicit sequencing. Notice that the access to queue lengths does not enable access to individual processes in a multi-process queue, which implies that one can leave the queuing discipline undefined if desired (possibly to be defined at higher levels of abstraction for the purpose of system tuning).
We formulate the concept of monitor as such as a class to be used as a "prefix", in the style of Simula, to user defined monitor classes, thereby making them subclasses of the class *monitor*. We also extend the prefixing mechanism analogously to procedures, providing a "procedure prefix" *mon* as one of the components of the monitor concept, to be applied to user defined monitor procedures for turning their bodies into critical regions. The class concept as used below is supposed to give rise to named objects, i.e. (shared) program variables. The definition is given in terms of binary semaphores. N stands for a sufficiently large natural number.

class *monitor* {*prefix to monitor classes*} ==
begin var s: *Nat* = 0 {*no. of signallers*}, m: **array** 0..N **of** *semaphore*(0)
{$m[0]$ is an exclusion semaphore, $m[1..N]$ contain the signaller stack};

```
    invar m[0] ≤ 1 ∧ m[1..N] = 0↑N ∧ (m[0] = 1 ⇒ s = 0)
          ∧ m[1..s] have queues of length 1 ∧ m[s+1..N] have empty queues;
    proc mon {"prefix" to monitor procedures} ==
          begin P(m[0]); {grabs critical region}
                inner; {represents a user defined monitor procedure body}
                V(m[s]) {frees critical region or gives it to last signaller}
          end;
    class condition {gives rise to named waiting queues} ==
    begin var k: Nat = 0 {no. of waiting processes}
               w: semaphore(0) {the process queue};
          invar w = 0 ∧ w has k processes waiting;
          proc wait ==
                 begin k := k+1; V(m[s]); P(w) {k ≠ 0} k := k−1 end;
          proc signal == if k ≠ 0 then s := s+1;
                      var t := s; V(w) {t ≠ 0} P(m[t]) {s ≠ 0} s := s−1 fi;
          func #^: Nat == k;
    end;
    V(m[0]) {frees critical region initially}
end
```

It is assumed that *mon* and *condition*, as well as the operators *wait*, *signal* and #^ of the latter (where the ^ indicates an argument position), are the only declared items accessible within subclasses of *monitor*, and that no items local to user subclasses, other than *mon* procedures, are accessible from the outside, except possibly in state assertions. Thereby the *monitor* class is protected from outside interference.

Although monitor objects are shared, the above coding, together with the access rules, ensure that the internal activity is essentially sequential. Thus, at any time all *mon* procedure activations, except at most one, are in (or directly in front of) a P-operation on a semaphore with the value 0. (The exceptional one may stand in front of a P-operation on a semaphore equal to 1, thus temporarily violating the invariant specified for one of the m or w semaphores.) It follows that Hoare logic applies to the contents of monitors, when augmented with special rules for *wait* and *signal* operations.

The postcondition $s \neq 0$ of the operation $P(m[t])$ of *wait* deserves a comment. By the precondition, the process in question has queued up on an m-semaphore with positive index, and the exit must be from the same queue. Inspection shows that all V-operations on m-semaphores have the form $V(m[s])$, so s must be positive at the time of the exit.

In the following sections we show and exemplify reasoning and programming strategies. We assume throughout that references to non-local variables do not occur within monitor objects. Although LIFO signalling may be assumed for the purpose of reasoning, it makes no difference to the logic if a more efficient implementation is chosen for a *signal* immediately prior to *mon* procedure **end**, as in [4].

3 Strategy 1

Here we review shortly the reasoning strategy identified in [4]. The central idea is to postulate an invariant I for the set $\mathcal{D}$ of variables local to the monitor, excluding *condition* variables. The invariant is to hold whenever the monitor object is at rest. It is also assumed that the programmer specifies an "await" condition Bc associated with each *condition* variable c, specifying the expected state of $\mathcal{D}$-variables at the time when a *c.wait* is terminated. The following rules of thought were given in [4]. The syntactic function $\mathcal{V}$ computes the set of variables occurring free in the argument expression.

$$\{I\}\, c.wait\, \{I \wedge Bc\} \quad \text{and} \quad \{I \wedge Bc\}\, c.signal\, \{I\}, \quad \text{where} \quad \mathcal{V}[I, Bc] \subseteq \mathcal{D}.$$

Clearly, if Bc holds prior to all operations *c.signal*, it must hold after any *c.wait*. The requirement I in the precondition of *c.signal* caters for the case that c is empty (the *signal* having no effect).
Technically, the above rules play the role of *assumptions* in a set of rules which involves all *mon* procedures, as well as the initializing class block tail.

$$\frac{\vdash \{P_0\}S_0\{I\}; \quad \{I\}c_i.wait\{I \wedge Bc_i\},\ \{I \wedge Bc_i\}c_i.signal\{I\},\ i{=}1..m \vdash \{P_j \wedge I\}S_j\{I\};\ j{=}1..n}{\vdash \{P_k\}S_k\{I\}} \quad \text{for } k{=}1..n$$

The deduction rules pertain to a *monitor* class containing m *condition* variables $c_1, \ldots, c_m$, n *mon* procedures with the bodies $S_1, \ldots, S_n$, and the initializing block tail S_0. The preconditions $P_0, P_1, \ldots, P_n$ express possible requirements for the correct usage of the monitor.

Example 1: (high level) semaphores

monitor **class** $Sema(s_0\colon Int)\ \{P_0\colon s_0 \geq 0\}$ ==
begin var $s\colon Int = s_0$;
  var $c\colon condition\ \{Bc\colon s{\neq}0\}$;
  invar $I\colon\ s \geq 0$;
 mon **proc** P ==
 $\{I\}$**begin if** $s = 0$ **then** $\{I\}$ *c.wait* $\{I \wedge Bc\}$ **fi**; $s := s{-}1$ **end**$\{I\}$;
 mon **proc** V == $\{I\}$**begin** $s := s{+}1$; $\{I \wedge Bc\}$ *c.signal*$\{I\}$ **end**$\{I\}$;
$\{I\}$ **end**

A partial correctness verification of *Sema* consists in applying Hoare logic to the decorated program, where the inserted state assertions are those indicated in the inference rule. (The predicates P_0 and I might alternatively have been expressed by declaring s_0 and s to be of the subtype *Nat* of natural numbers.) Then, for any given semaphore object, say **var** $X : Sema(k)$, where $k \geq 0$, we may conclude that the invariant $X.s \geq 0$ holds throughout the scope of X.

A binary semaphore would be described by a specialized version of the *Sema* class, in which the predicates P_0 and I are strengthened accordingly, and the procedure V has the (added) precondition $s=0$.

4 Strategy 2

Unfortunately, the class *Sema* would remain partially correct with respect to I even if the *signal* operation of the V procedure were omitted. In complicated cases forgetting a *signal* could be an error easy to make and difficult to locate. The difficulty would arise from the kind of incorrect program behaviour that would be observed: nothing "wrong" would happen, but actions which ought to take place would not. For instance, the system as a whole might stop prematurely without error indications. This indicates that partiality of correctness might be a more dangerous notion in the context of concurrency than for sequential programs.
It turns out, however, that errors of this kind can in fact be combatted within the framework of Hoare logic, provided invariants refer to a sufficient amount of "historic" data, such as sequences of records of past actions, say accumulated in "mythical" auxiliary variables. In order to prove sufficient signalling in a monitor, *condition* queue lengths together with ordinary $\mathcal{D}$-variables usually contain enough historic information. For instance, the monitor invariant of *Sema* strengthened by conjoining $\#c=0 \vee s=0$ asserts that waiting occurs only as long as necessary for the program logic.
In order to see the consequences of having an invariant, as well as await conditions, referring to queue lengths, it is useful to look at the details involved in *wait* operations. (In the present context it may be natural to drop the requirement that the monitor invariant should hold before a *signal*; it may not hold unaltered after the corresponding *wait* anyway):

$$\{I^{\#c}_{\#c+1}\} <\mathsf{enters}\ c> \{I\} <\mathsf{sleeps}> \{\#c\neq 0 \wedge Bc^{\#c}_{\#c-1}\} <\mathsf{leaves}\ c> \{Bc\}$$

Clearly entering and leaving the c-queue have the effect of changing the value of $\#c$, seen as a program variable. It is at the moment of falling asleep that the critical region is relinquished, and that is when the invariant must be required to hold. Furthermore, it is after having completed the entire *wait* operation that the await condition will be expected to hold. Left construction now shows what must be required in the preconditions of $c.wait$ and $c.signal$. Notice that $\#c\neq 0$ holds until the process leaves c. Thus, the assumptions about waits and signals should be as follows:

$$\{I^{\#c}_{\#c+1}\}c.wait\{Bc\} \quad \text{and} \quad \{\#c\neq 0 \wedge Bc^{\#c}_{\#c-1}\}c.signal\{I\} \quad \text{for } \mathcal{V}[I, Bc] \subseteq \mathcal{C} \cup \mathcal{D}$$

where $\mathcal{C}$ is the set of *condition* variables declared in the monitor. The following axiom schema will cater for signals to empty *condition* queues:

$$\vdash \{\#c=0 \wedge P\}c.signal\{P\} \qquad \text{for arbitrary predicate } P$$

With this reasoning strategy it is easy to verify that the class *Sema* is partially correct with respect to an invariant strengthened as indicated above, $I: s \geq 0 \wedge (s = 0 \vee \#c = 0)$, but only if a proper *signal* operation occurs in the V procedure. An adequate await predicate may be found by left construction, $Bc: I^s_{s-1}$.
The new strategy can provide the same kind of guarantee for the following, possibly more efficient semaphore implementation:

Example 2

monitor **class** $Sema'(s_0: Int)\ \{s_0 \geq 0\}$ ==
beg var $s: Int = s_0$;
var c: *condition* $\{Bc: I\}$;
invar $I: s \geq 0 \wedge (\#c = 0 \vee s = 0)$;
*mon***proc** P ==
$\{I\}$ **beg if** $s = 0$ **th** $\{I^{\#c}_{\#c+1}\}$ *c.wait* $\{Bc\}$ **el** $s := s-1$ **fi** $\{I\}$;
*mon***proc** V ==
$\{I\}$ **beg if** $\#c \neq 0$ **th** $\{\#c \neq 0 \wedge Bc\}$ *c.signal*$\{I\}$
el $s := s+1$ **fi** $\{I\}$;
$\{I\}$**end**

In order to verify *Sema'* we must prove the following verification conditions: $I \wedge s = 0 \Rightarrow I^{\#c}_{\#c+1}$, $I \wedge s \neq 0 \Rightarrow I^s_{s-1}$, $I \wedge \#c \neq 0 \Rightarrow I^{\#c}_{\#c-1}$, and $I \wedge \#c = 0 \Rightarrow I^s_{s+1}$, all of which hold trivially, and prove that the invariant holds initially.

It should be noted that, although *Sema* and *Sema'* have been proven partially correct with respect to the same monitor invariant, it has not been proved formally that objects of both kinds have similar external behaviour, or indeed that they behave according to the standard definition of semaphores. (Actually, adding a *c.signal* after the *c.wait* would not harm the proof!)

5 Strategy 3

It is sometimes practical to use *signal* operations not immediately followed by *mon* procedure **end**. In the reasoning about such occurrences of *signal* we cannot in general require a postcondition describing monitor resting states. In order to improve our formal system we introduce a mythical variable $\mathcal{S}$: *Stack*(*condition*), modelling the LIFO stack of signallers by (references to) the targets of the unfinished *signal* operations.
Let I be a monitor invariant containing $\mathcal{S}$. Then $I \wedge \mathcal{S} = \varepsilon$, where ε is the empty stack, describes resting states, whereas $I \wedge \mathcal{S}$ **top** c, where the second conjunct states that $\mathcal{S}$ is non-empty and its top element is a reference to c, specifies states such that control is about to return to a c signaller.
Permitting $\mathcal{S}$ to occur in the monitor invariant, as well as in await conditions, we must take into account the fact that $\mathcal{S}$ changes within *signal* operations. If

also references to queue lengths are allowed, a decorated model of *c.signal* for $\#c \neq 0$ must be as follows:

$$\{\#c \neq 0 \wedge Bc^{\#c,\ \mathcal{S}}_{\#c-1,push(c,\mathcal{S})}\}\ \mathcal{S} := push(c,\mathcal{S})$$
$$\{\#c \neq 0 \wedge Bc^{\#c}_{\#c-1}\} < \text{wake up and wait} > \{I\}\ \mathcal{S} := pop(\mathcal{S})\ \{I^{\mathcal{S}}_{push(c,\mathcal{S})}\}$$

We have to assume that the monitor invariant holds whenever a process gives up critical region; it follows that it will hold when a signaller regains control. The decoration of the model determines the form of assumptions about *signal* operations. The assumption about *c.wait* is as for strategy 2, but the fact that the waking up is caused by a *signal* on c is now expressible:

$$\{I^{\#c}_{\#c+1}\}c.wait\{Bc \wedge \mathcal{S}\ \mathbf{top}\ c\} \text{ and } \{\#c \neq 0 \wedge Bc^{\#c,\ \mathcal{S}}_{\#c-1,push(c,\mathcal{S})}\}c.signal\{I^{\mathcal{S}}_{push(c,\mathcal{S})}\}$$

The axiom for signalling on an empty *condition* remains unaltered. In addition we may state axioms expressing the fact that local variables remain unchanged over *wait* and *signal* operations:

$$\vdash \{P\}c.wait\{P\} \qquad \text{for } \mathcal{V}[P] \cap (\mathcal{C} \cup \mathcal{D} \cup \{\mathcal{S}\}) = \emptyset$$
$$\vdash \{P\}c.signal\{P\} \quad \text{for } \mathcal{V}[P] \cap (\mathcal{C} \cup \mathcal{D}) = \emptyset$$

Notice that $\mathcal{S}$ also remains unchanged over *signal*s. Obviously $I \wedge \mathcal{S} = \varepsilon$ may be assumed in the precondition of any *mon* procedure.

Example 3

We show a decorated version of a readers-and-writers monitor class, conforming to that of [4] in that it is fair to both sides. The variables r and w count the number of active readers, resp. writers, and the *condition* variables *cr* and *cw* are the queues of waiting readers and writers. The monitor invariant is in the form of four conjuncts: I_1 expresses the obvious exclusion conditions, I_2 and I_3 are the conditions that writers, resp. readers may be waiting, and I_4 states that there are active readers and none waiting when control returns to a *cr* signaller.

The second and third conjuncts are sufficient to prove adequate signalling. The last one makes it possible to prove the postcondition $r \neq 0$ of the *startread* procedure. Thus, the post- and preconditions of the *startread* and *endread* procedures express a requirement for parenthetic usage, which is also the case for *startwrite* and *endwrite*.

monitor **class** *Readers-and-Writers* ==
beg var r, w: $Nat = 0$,
cr, cw: *condition* $\{Bcr: w = 0,\ Bcw: r + w = 0\}$
invar I: $(I_1: ((r = 0 \vee w = 0) \wedge w \leq 1) \wedge$
$I_2: (\#cw \neq 0 \Rightarrow r + w \neq 0) \wedge$

$$I_3 : (\#cr \neq 0 \Rightarrow w + \#cw \neq 0) \wedge$$
$$I_4 : (\mathcal{S}\ \mathbf{top}\ cr \Rightarrow r \neq 0 \wedge \#cr = 0))$$

*mon***proc** *startread* == **beg if** $w + \#cw \neq 0$ **th** *cr.wait* **fi** $\{w = 0\}$
$r := r+1\ \{w = 0 \wedge r \neq 0\}$ *cr.signal* **end** $\{I \wedge r \neq 0\}$;
*mon***proc** *endread* == $\{r \neq 0\}$ **beg** $r := r-1\ \{I_3 \wedge w = 0 \wedge \mathcal{S} = \varepsilon\}$
if $r = 0$ **th** $\{I_3 \wedge r + w = 0 \wedge \mathcal{S} = \varepsilon\}$ *cw.signal* **fi end**$\{I\}$;
*mon***proc** *startwrite* == **beg if** $r + w \neq 0$ **th** *cw.wait* **fi**
$\{r + w = 0 \wedge \neg \mathcal{S}\ \mathbf{top}\ cr\}\ w := w+1$ **end** $\{I \wedge w = 1\}$;
*mon***proc** *endwrite* == $\{w = 1\}$ **beg** $w := w-1\ \{r + w = 0 \wedge \mathcal{S} = \varepsilon\}$
if $\#cr \neq 0$ **th** *cr.signal* **el** *cw.signal* **fi end**$\{I\}$;
$\{I\}$**end**

The procedure preconditions I and $\mathcal{S} = \varepsilon$ have been omitted in the text. The decorations are otherwise sufficient for proving all verification conditions. Actually, a stronger postcondition is provable for *startread*: the postcondition of the *signal* operation on cr is $I^{\mathcal{S}}_{push(cr,\mathcal{S})}$ by assumption in the case $\#cr \neq 0$, which implies $I \wedge r \neq 0 \wedge \#cr = 0$, and the latter condition also holds if $\#cr = 0$. Thus, whenever a reader is allowed to become active, all of them are.

It is possible to let the *endwrite* procedure do all necessary signalling to readers, by removing the (recursive) *cr.signal* of *startread* and replacing that of *endwrite* by the loop: **while** $\#cr \neq 0$**do** *cr.signal* **od**. Since a return to *endwrite* will now occur after every *cr.signal*, the monitor invariant must be adjusted by weakening I_3 and I_4:

$$I'_3 : \neg \mathcal{S}\ \mathbf{top}\ cr \Rightarrow I_3 \quad \text{and} \quad I'_4 : \mathcal{S}\ \mathbf{top}\ cr \Rightarrow r \neq 0$$

Using the loop invariant $w = 0 \wedge r + \#cr \neq 0$, the required precondition of *cr.signal* is implied, and its postcondition is $(I')^{\mathcal{S}}_{push(cr,\mathcal{S})}$, which implies $r \neq 0$ by the fourth conjunct and then $w = 0$ by the first conjunct. The postcondition of the loop implies the original postcondition of *startread*, $I \wedge r \neq 0 \wedge \#cr = 0$ as it should.

6 Strategy 4

Sometimes the await predicate Bc associated with a *condition* variable c must depend on local variables, e.g. procedure parameters. Then Bc cannot be expressed outside that particular procedure instance without remodelling the programming language. In [4] a number of such cases have been elegantly treated by defining *condition* variables to be priority queues. Here we mention another, more explicit scheduling technique.
Let Bc depend on a variable $t : T$ (possibly a tuple), $t \in \mathcal{V}[Bc]$. One way of communicating the required await predicate to other procedure instances is to include in the monitor a data structure implementing a function

$c\colon\ T \longrightarrow$ *condition* variable, such that
$\forall t\colon T \bullet$ the *condition* variable $c(t)$ has the associated await predicate Bc.

c will usually have to be a one-to-one function, so that $c(t_1)$ and $c(t_2)$ are distinct variables for $t_1 \neq t_2$, associated with distinct await predicates, $Bc^t_{t_1}$ and $Bc^t_{t_2}$ respectively.
A simple reasoning strategy which caters for this programming technique may be characterized by sequencing assumptions of the following form:

$\{I\}c(e).wait\{Bc^t_e\}$ and $\{c(d)\neq 0 \wedge Bc^t_d\}c(d).signal\{I\}$
for $\mathcal{V}[I] \subseteq \mathcal{D},\ \mathcal{V}[Bc] \subseteq \mathcal{D}\cup\{t\}$, and $\mathcal{V}[e] \cap (\mathcal{C} \cup \mathcal{D}) = \emptyset$,

where $c\ \epsilon\ \mathcal{C}$. The requirement on the argument expression e, that it should depend only on local variables, ensures that its value remains unchanged over the *wait* operation.

Example 4

We formulate a concept of "generalized semaphore", where an operation $GP(n)$, $1 \leq n \leq N$, is defined as n P operations on an ordinary semaphore, but executed as a single atomic operation not violating the semaphore invariant, and similarly for $GV(n)$.

monitor **class** $Gsema(s_0\colon Nat)\{s_0 \leq N\}$ ==
beg var $s\colon Nat = s_0$;
  var $c\colon$ **array** $1..N$ **of** *condition* $\{Bc[i\colon 1..N]\colon i \leq s \leq N\}$;
  invar $s \leq N$;
*mon***proc** $GP(k : Nat)\{k \leq N\}$ ==
  beg if $k > s$ **th** $c[k].wait$ **fi** $\{k \leq s\}$ $s := s-k$ **end**;
*mon***proc** $GV(k\colon Nat)\{k \leq N-s\}$ ==
  beg var $j\colon Nat1 = s+1$; $s := s+k$; **while** $j \leq s$ **do**
    if $\#c[j]\neq 0$ **th** $c[j].signal$ **el** $j := j+1$ **fi od end**;
end

The reasoning strategy 4 is sufficient for a verification with respect to the stated invariant. (The monitor invariant may in this case serve as the loop invariant as well.) It may, however, be combined with strategy 2 or 3 in order to provide for more powerful reasoning. Thus, the combination of strategies 3 and 4 enables the verification of the following monitor invariant:

$$s \leq N \wedge \forall i, j\colon Nat1 \bullet i \leq s \wedge (\mathcal{S} = \varepsilon \vee (i < j \leq N \wedge \mathcal{S}\ \mathbf{top}\ c[j])) \Rightarrow \#c[i] = 0,$$

with await predicates $Bc[t\colon 1..N]\colon t \leq s \leq N \wedge \forall u\colon Nat1 \bullet u < t \Rightarrow \#c[u] = 0$,
and loop invariant $s \leq N \wedge j \leq N+1 \wedge \forall i\colon Nat1 \bullet i < j \Rightarrow \#c[i] = 0$.

The monitor invariant expresses sufficient signalling for the scheduling principle "cheapest need first" (the second disjunct of the parenthesis is needed in order to verify the signalling loop). Programming according to the combination of strategies 2 and 4 leads to a simpler monitor invariant for adequate signalling (without the disjunction), but to a more complicated program. We leave it to the reader to program and verify according to such a strategy.

In the example we have implemented the function c as an array. For large N it would be better to use a more dynamic data structure, in which only the non-empty *condition* variables were represented at any time.

7 External specification of monitors

It is an old idea (see e.g. [8]) that any variable, including an arbitrary class object, of a sequential program can be abstractly represented by the sequence of updates, parameter values included, since its declaration. This provides an entirely black box representation, since only aspects visible from the outside occur. As far as monitor variables are concerned, a more detailed representation is needed in order to cater for the internal delays which may occur.
It is sufficient, however, to distinguish between events corresponding to operation initiation and operation termination, as in [8]. Let a monitor procedure be of the form

$$mon\ \textbf{proc}\ p(\textbf{in}\ x{:}T_x,\ \textbf{out}\ y{:}T_y) == \ldots\ \ldots$$

where x and y are (lists of) input and output parameters. Then we may represent the events of initiating and terminating a p operation by structures of the form $p^i(x{:}T_x)$ and $p^t(y{:}T_y)$, respectively, where p^i and p^t are record classes.
Let a *monitor* class with *mon* procedures $p_1, \ldots, p_n$ be decorated by introducing a mythical history variable $H : Sequence(p_1^i \cup p_1^t \cup \ldots \cup p_n^i \cup p_n^t)$ and let the **begin** and **end** of *mon* **proc** p_k, $k = 1, 2, \ldots, n$, have the mythical effects of updating H by appending respectively $p_k^i(x_k)$ and $p_k^t(y_k)$, where x_k and y_k are the values of the input and output parameters.
Now an external specification of a monitor class M is a predicate *Spec* on the history variable H and the formal parameters of M. A verification of that specification requires a monitor invariant I such that $\vdash I \wedge \mathcal{S} = \varepsilon \Rightarrow Spec$. Notice that I may refer to variables hidden from the external specification.

Example 5

A semaphore class *monitor* **class** $Sema(s_0 : Int)\{s_0 \geq 0\} == \ldots\ \ldots$ could have the following external specification:

$$s_0 \geq 0 \Rightarrow \#H/P^t = \min(\#H/P^i,\ \#H/V^i + s_0) \wedge \#H/V^t = \#H/V^i$$

where H/C stands for that subsequence of H which consists of all the elements of class C, and $\#\hat{}$ is the length operator. The specification states that the number of terminated P operations is equal

to either the number of initiated P's or the number of initiated V's plus s_0, whichever is smaller. Thereby the delaying of P operations is specified up to an arbitrary queuing discipline. V operations are terminated immediately.

The monitor classes of examples 1 and 2 both satisfy that specification. This fact can be proved by choosing the following monitor invariant, which requires strategy 3 reasoning for its proof:

$$s \geq 0 \ \wedge \ (s = 0 \vee \#c = 0) \ \wedge \ s = s_0 + \#H/V^i - \#H/P^t \ \wedge \\ \#c = \#H/P^i - \#H/P^t \ \wedge \ \#H/V^i = \#H/V^t + \#\mathcal{S}$$

8 Conclusion

We have demonstrated reasoning techniques within the framework of Hoare logic for proving adequat signalling, arbitrary use of signal operations, and dealing with await predicates depending on local variables. In some cases ad-hoc preconditions have been prescribed for monitor procedures, indicating restrictions on the intended usage. If so, it has been tacitly assumed that any user would correctly observe these restrictions. An alternative could be to check such preconditions explicitly, possibly reacting by introducing necessary delays using waits (and signals). Another possibility is to replace monitor objects by processes deciding when and what kind of external calls to accept, thereby introducing implicit delays for unwanted calls. The internal logic would be similar, and so would the technique of external specification.
The reasoning about concurrent processes interacting through monitors can be in two steps: local reasoning according to Hoare logic, each process having a mythical history variable accumulating monitor calls, and a rule for parallel composition which requires compatibility in a certain sense of the history variables occurring in the system. See e.g. [9]. Although the results are restricted to partial correctness, the use of historic sequence variables implies that many aspects of behaviour, usually classified as liveness properties, can be reasoned about.

References

[1] C.A.R. Hoare: An Axiomatic Basis for Computer Programming. *Comm. ACM* **12**(10)(1969), pp. 576–580.

[2] C.A.R. Hoare: Proof of Correctness of Data Representations. *Acta Informatica* **1**(1972), pp. 271–281.

[3] C.A.R. Hoare: Towards a Theory of Parallel Programming. In C.A.R. Hoare and R.H. Perrott, eds.: *Operating System Techniques* (Academic Press 1972) pp. 61–71.

[4] C.A.R. Hoare: Monitors: an Operating System Structuring Concept. *Comm. ACM* **17**(10)(1974) pp. 549–557.

[5] E.W. Dijkstra: Co-operating Sequential Processes. In F. Genuys, ed.: *Programming Languages*, (Academic Press 1968), pp. 43–113.

[6] P. Brinch Hansen: *Operating System Principles* (Prentice Hall 1973).

[7] O.-J. Dahl and C.A.R. Hoare: Hierarchical Program Structures. In O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare: *Structured Programming* (Academic Press 1972) pp. 175–220.

[8] O.-J. Dahl: Can Program Proving be made Practical? In M. Amirchahy and D. Neel, eds.: *Les Fondements de la Programmation*, (IRIA 1978), pp. 57–113.

[9] N. Soundararajan: A Proof Technique for Parallel Programs. *Theoretical Computer Science 31.* (1984) pp.13–29.