A Note on Monitor Versions

Essay in the Honour of C.A.R. Hoare

Ole-Johan Dahl
Department of Informatics
University of Oslo, Norway

30th July 1999

Versions of Hoare Monitors have been included and implemented in several languages for parallel programming. Curiously enough all except one are nontrivial variations of Hoare's original proposal. We discuss the relative merits of the different versions from the standpoints of sequencing control, abstraction, and efficiency. Our conclusion is that the original proposal is superior in all three respects, with the possible exception of one variation. The latter, however, suffers from other disadvantages.

Keywords: monitors, concurrent processes, programming languages.

1 Introduction

Versions of the Hoare monitor concept, [Ho74], have been introduced and implemented in several programming languages. All versions agree that monitor procedures are the only items declared local to a monitor object which are accessible nonlocally. Furthermore, their bodies must be executed in critical regions (CR) with respect to the object. The versions differ, however, with respect to the semantics of *signal* operations. [An91] distinguishes between the following regimes, which have all been implemented:

- *Signal and continue* (SC): The signaller retains the CR, whereas the signalled process joins the main waiting queue to compete for CR.
- *Signal and exit* (SX): CR is given to the signallee, the signaller is forced to exit from its monitor procedure activation.
- *Signal and wait* (SW): CR is given to the singnallee, whereas the signaller joins the main waiting queue.
- *Signal and urgent wait* (SU): As SW, except that the signaller joins a separate waiting queue with priority over the main one.

Tony's own proposal was for the SU regime. He did not, however, provide a discussion justifying the choice, nor, as far as we know, has anyone else. Therefore the present essay. The SU regime has been implemented as part of the Pascal Plus language.

[An91] argues in favour of the SC regime. The main argument seems to be ease of programming.

In our view the following are the main criteria relevant for the quality of a signalling mechanism, apart from the ease of use.

- Controlling waiting and sequencing is of great importance in the design of multiprocessing systems, especially in cases of high density traffic.
- Abstract representation of monitors is a prerequisite for high level specification.
- Run time efficiency.

It follows from the discussions below that the SU regime is much superior to the SC and SW regimes in all three respects. It is superior also with respect to ease of programming whenever efficiency and sequencing control are important.

The SU and SX regimes are fairly similar with respect to the criteria listed. SX is in general somewhat more efficient, however, it enforces a restricted programming style which is not easy to comply with, and which sometimes requires more complicated external interfaces than single procedure calls. As mentioned already by Hoare in [Ho74], an implementation shortcut is possible for those *signal* operations that immediately precede monitor procedure **end**. This means that the efficiency gain of SX will be obtained in SU as well, to the extent that the programming restriction enforced in SX is adhered to. For these reasons the SX regime is not considered in the following discussions.

It may be useful to specialize the SU regime by applying the LIFO structure to the signaller queue, thus making *signal* imperatives akin to semi-coroutine calls in the sense of [DH72].

2 Program Logic

The programming strategy advocated in [Ho74] is to specify an invariant I for the variables declared local to the monitor (in addition to their declared types) to hold between monitor procedure calls, as well as an "await condition" Bc associated with each *condition* variable c. Then the following rules of thought for *wait* and *signal* operations in a certain sense extend conventional Hoare logic for partial correctness:

$$\{I\}c.wait\{Bc \wedge I\} \qquad \{Bc \wedge I\}c.signal\{I\} \qquad \text{for } \mathcal{V}[I, Bc] \subseteq \mathcal{D}$$

Here $\mathcal{V}[E]$ is the set of free variables of the expression E, and $\mathcal{D}$ is the set of variables local to the monitor, parameters included, but *condition* variables excluded. It is convenient to assume that variables nonlocal to the monitor are not accessed. The rules may be embellished by conjoining an arbitrary assertion about strictly local variables to the pre- and postconditions of a *wait* or *signal* in order to express the constancy of such variables.

It turns out that this strategy of formal reasoning is not very strong. For instance, in the following implementation of a semaphore initialized to zero sufficient signalling cannot be proved. To wit: the invariant is provable even if the *signal* operation of the V procedure is omitted.

monitor *Sema* ==
beg var $s{:=}0,\ c$: *condition*$\{Bc{:}\ s{=}1\}$
  invar $I{:}\ s{\geq}0$
  proc P == **beg if** $s{=}0$ **th** *c.wait* **fi**; $s := s{-}1$ **end**
  proc V == **beg** $s := s{+}1$; *c.signal* **end**
end

The fact that all actions caused by a *signal* operation take place in an unbroken critical region, can better be taken advantage of by allowing the invariant and await conditions to refer to *condition* queues, e.g. through a length function, #, and by not insisting that the invariant must hold at the time of signalling. A corresponding reasoning strategy can be formulated as follows:

$$\{I^{\#c}_{\#c+1}\}c.wait\{Bc\} \quad \{\textbf{if } \#c{\neq}0 \textbf{ th } Bc^{\#c}_{\#c-1} \textbf{ el } I \textbf{ fi}\}c.signal\{I\}$$

for $\mathcal{V}[I, Bc] \subseteq \mathcal{D} \cup \mathcal{C}$, where $\mathcal{C}$ is the set of declared *condition* variables. (Notice that we are treating *c.wait* and *c.signal* as assignments to a "variable" $\#c$. That is logically sound provided that the length operator is the only status observing function applicable to *condition* variables.)

Now the monitor invariant of *Sema* may be strengthened by adding the conjunct $(\#c{\neq}0 \Rightarrow s{=}0)$ stating that no unnecessary waiting will take place, or in other words: P operations, for a given queuing discipline, wait only as long as is logically necessary for the variable s to remain nonnegative.

The use of signalling is hampered by the demand that the monitor invariant must hold in the postcondition. That nearly amounts to a requirement that *signal* operations precede immediately procedure **end**. In [Da94] is shown how more flexibility may be obtained by defining the signaller queue as a LIFO stack, and introducing a mythical variable representing the essence of that stack.

It happens that await conditions have to depend on data local to the procedure body containing the *wait*. If so, the strategies for programming and reasoning sketched above do not immediately apply. A general solution then is to resort to semi-active waiting:

while $\neg Bc$ **do** *c.wait* **od**

combined with "sufficient" signalling. This is fairly awkward within the SU regime, but a partial remedy is to introduce "priority waiting", see [Ho74]. As an alternative, [Da94] explains a general technique of enriching the monitor data structure which enables controlled passive waiting supported by formal reasoning.

For programming under the SC regime semi-active waiting as above is recommended. This is an easy way of achieving a certain degree of correctness, to the extent that "sufficient signalling" can be ascertained. A "signal_all" mechanism can help in that respect. Little if any sequencing control is possible with this style of programming, although priority waits may help to some extent. Full control of sequencing can be achieved by means of precarious coding, often involving the use of auxiliary variables, not supported by formal reasoning (at any rate according to [WB79]), and probably very difficult to debug. Under the SW regime, control of sequencing is better, but full control as in SU generally requires unnatural coding using an auxiliary variable.

3 Abstract Representation

Let X be a class object accessible only through its declared operators ("methods"). In a monoprogrammed environment the abstraction of X would be a variable whose value is the abstraction of the current state of the object. Each call for an operator, say $X.p(\mathsf{params})$ may change the state of X. In the abstract view each such call is an atomic event, characterized by the parameterized call. Notice that the historic sequence of parameterized operator calls is an *entirely abstract* representation of the variable X, in the sense that only information observable outside X is included. It is also *complete,* in the sense that the state of X is uniquely determined by the current history (assuming that no reference to variables nonlocal to the object occurs within any of its operators). In the abstract view of X there is a coarsening of time, such that each event represents an *instantaneous* state transition.

In a multiprocessing environment the state of X is again uniquely determined by the sequence of calls for X-operators, given that each invocation is a *critical region* with respect to the object.

For a monitor object X the abstract view is somewhat more complicated, because *wait* operations may incur waiting periods of unbounded length, during which other X-events may take place. Thus, the initiation and termination of an operator invocation may be widely separated in time, and occur in separate critical regions. Therefore the invocation of an operator p must be abstractly represented by a pair of events, p^i, p^t (parameterized), representing the initiation and termination of the invocation, respectively.

It is reasonable to regard an operation performing no *wait* as atomic, and according to the SU and SC regimes the initiation and termination will then in fact occur within the same critical region. That does not in general hold for the SW regime.

According to the program logic an operation invocation, say p, can by signalling cause other procedure invocations, $q, r, \ldots$ to resume operation and possibly to terminate. Under the SU regime the corresponding events $p^i, q^t, r^t, \ldots$ will all occur in the same critical region. In abstract terms they should therefore be viewed as *taking place simultaneously.* If follows that a meaningful abstract representation of a monitor object is an historic sequence $\mathcal{H}$ of event groups:

$$\mathcal{H}\colon Seq(Seq(E)), \quad \text{where } E \text{ is the set of event types for the object.}$$

Each group will consist of one initiation event and zero or more terminations. In concrete terms the time between groups are intervals of quiescence, the object waiting for another procedure initiation. Assuming that the queuing disciplines of all *condition* queues are given, the contents of each event group is completely determined by its initiating event together with the history at the time. In that sense, under the SU regime, the history is a complete representation of the monitor object.

Sometimes the mutual order of the events of a group is irrelevant and can be abstracted away. That is e.g. the case for semaphore objects implemented as above:

$$\mathcal{H}\colon Seq(Set(P^i \cup P^t \cup V^{it}))$$

where V^i and V^t events for convenience are combined into single events of the class V^{it}. A possible history could be as follows, where each call is identified by

tagging the events by the length of the history at the time of initiation:

$$\langle\{V_0^{it}\},\{P_1^i,P_1^t\},\{P_2^i\},\{P_3^i\},\{V_4^{it},P_2^t\}\rangle$$

It shows that one P operation could terminate immediately, the next one after some waiting, whereas the last one still waits.

Let $\#(\mathcal{H},E)$ denote the number of events of the class E occurring in $\mathcal{H}$. Then $\#(\mathcal{H},P^t)\leq\#(\mathcal{H},P^i)$ holds by a general property of monitor histories. Furthermore, a semaphore property gives $\#(\mathcal{H},P^t)\leq\#(\mathcal{H},V^{it})$. We can, however, formulate a stronger abstract specification which in addition expresses the absence of unnecessary waiting:

$$\#(\mathcal{H},P^t)=min(\#(\mathcal{H},P^i),\#(\mathcal{H},V^{it}))$$

For the SC and SW regimes a fully abstract and complete representation of a monitor object is not possible in general. The sequence of terminations following a given initiation event will not be determined by the past history alone, but may in addition depend on subsequent initiations. For the SC regime a semaphore specification can say nothing about waiting times. Thus, the equality above would have to be replaced by a less-than-or-equals relation. (For the SW regime an abstract specification could resemble the one above, provided *signal* operations immediately preceding procedure **end** were subjected to a suitable ad hoc formal treatment.)

Returning to the SU regime, we indicate briefly how the semaphore implementation above may be verified with respect to the given abstract specification. The main idea is to decorate the program text with certain "mythical" declarations and operations. They can be supplied mechanically. In the program text below the mythical text is marked by curly brackets (which are also used to denote singleton sets). The operator $\vdash$ indicates extending a sequence by a term at its right end. The functions rt and lr stand for the rightmost term of a nonempty sequence, respectively its "left rest". ε denotes an empty sequence.

monitor *Sema* ==
beg {**var** $\mathcal{H}$: $Seq(Set(P^i\cup P^t\cup V^{it}))=\varepsilon$}
  var $s:=0$, c: *condition* {Bc: $s=\#(\mathcal{H},V^{it})-\#(\mathcal{H},P^t)=1\wedge$
    $\#c=\#(\mathcal{H},P^i)-\#(\mathcal{H},P^t)-1\geq 0$}
  invar I: $s=\#(\mathcal{H},V^{it})-\#(\mathcal{H},P^t)\geq 0\wedge(\#c\neq 0\Rightarrow s=0)\wedge$
    $\#c=\#(\mathcal{H},P^i)-\#(\mathcal{H},P^t)\geq 0$
  proc P == {I}**beg** {**const** $\mathcal{T}ag:=\#\mathcal{H}$; $\mathcal{H}:=\mathcal{H}\vdash\{(P^i)_{\mathcal{T}ag}\}$}
    if $s=0$ **th** *c.wait* **fi**; $s:=s-1$
    {$\mathcal{H}:=lr(\mathcal{H})\vdash(rt(\mathcal{H})\cup\{(P^t)_{\mathcal{T}ag}\})$} **end**{$I$}
  proc V == {I}**beg** {$\mathcal{H}:=\mathcal{H}\vdash\{(V^{it})_{\#\mathcal{H}}\}$ $s:=s+1$; *c.signal* **end**{I}
end

Reasoning according to the second strategy mentioned in section 2 is adequate. The correct await condition may be deduced by left construction of I from the **end** of P. The specified preconditions of *c.wait* and *c.signal* must be verified. The fact that I implies the above abstract specification verifies the semaphore implementation.

Hoare-like reasoning strategies which permit detailed access to the contents of condition queues can be devised, which are complete in the sense that any

valid monitor implementation may be formally verified with respect to its abstract specification, see [Gj88] and [Ba99]. The rules for *wait* and *signal* operations will play the roles of assumptions for the proof of each monitor procedure. (For soundness the Hoare logic applicable to monitor procedures must be reduced by cutting out the rule of constancy and rules derived from it. Thereby the verification of the assumption preconditions will be enforced.)

4 Implementation and Efficiency

For evaluating the run time efficiency of Hoare monitors the main overheads are the cost C per entry/exit of critical region and the cost E per environment update. Rough counting gives the following overhead for a monitor procedure invocation causing n effective *signal* operations, directly or indirectly.

For SU:	$C + 2(n+1)E$
For SC and SW:	$(n+1)C + (n+2)E$

For semi-active waiting the cost per retest is $2E$, respectively $C{+}E$. We may safely assume C to be much larger than E. Thus, the SU regime is superior.

It is true that semi-active waiting is sometimes more awkward in the SU regime, however, as already mentioned, it can always be avoided, sometimes at the cost of additional data structure in the monitor. Anyway, the use of semi-active waiting should be discouraged in serious systems programming. It can lead to catastrophic overheads for traffic densities approaching saturation.

5 Conclusion

Of the four signalling regimes discussed in [An91] we have shown that the SU regime is definitely superior to either of SC and SW, with respect to sequencing control and programming logic, abstract representation and specification, as well as run time efficiency. Our analysis thus strongly supports Tony's original choice of regime. It is all the more surprising that both the SC and SW regimes have been implemented in major programming languages. For instance, the JAVA concept of signalling is based on the SC regime.

The SX regime can be seen as a restriction of SU. In our opinion it offers no advantages, but it does restrict programming in a way sometimes difficult to comply with, and which can require more complicated user interfaces. The reader is invited to program (and prove) a strictly FIFO version of a readers/writers control satisfying the usual exclusion and concurrency requirements, assuming FIFO *condition* queues.

Acknowledgements

Tore Jahn Bastiansen has contributed to this essay through many stimulating discussions and some well deserved criticism.

References

[DH72] O.-J. Dahl, C.A.R. Hoare: Hierarchical Program Structures. In Dahl, Dijkstra, Hoare: *Structured Programming*, Academic Press 1972.

[Ho74] C.A.R. Hoare: Monitors: an Operating System Structuring Concept. *Comm. ACM* 17(10)(1974), pp. 549–557.

[WB79] J. Welsh and D.W. Bustard: Pascal Plus, another Language for Modular Multi-programming. *Software—Practice and Experience*, 9:947–957, 1979.

[Gj88] S. Gjessing: Semantics and verification of monitors and systems of monitors and processes. *Distributed computing* (1988) 2, pp. 190–200.

[An91] G.R. Andrews: Concurrent Programming, Principles and Practice. The Benjamin/Cummings Publ. Comp., 1991.

[Da94] O.-J. Dahl: Monitors Revisited. In A.W. Roscoe, ed.: *A Classical Mind. Essays in Honour of C.A.R. Hoare*, Prentice Hall 1974, pp. 93–103.

[Ba99] T.J. Bastiansen: In preparation.