ON THE DESIGN OF CONCURRENT PROGRAMS*

ERIC C.R. HEHNER

Computer Systems Research Group, University of Toronto

ABSTRACT

By developing a concurrent program from a sequential one in the form of recursive refinement, we gain three benefits. The methodological benefit is that we proceed in small steps from an understood sequential program. The correctness benefit is that the program is starvation-free without appealing to a fair scheduler. The efficiency benefit is that less mutual exclusion is required than with other concurrent programming methods.

RÉSUMÉ

Trois avantages découlent du développement d'un programme simultané à partir d'un programme séquentiel. Du côté méthodologique on peut s'éloigner à petits pas d'un programme séquentiel bien établi. Du côté exactitude, le programme ne sera jamais affamé, et celà sans qu'on ait à avoir recours à un programmateur équitable. Du côté efficacité, cette méthode demande moins d'exclusions mutuelles que d'autres méthodes de programmation simultanée.

INTRODUCTION

My concern is to find methods of programming that allow us to construct programs in small, sure steps, with no large leaps of intuition required. For sequential programming, this has been the subject of many texts and papers. It is now time to bring this concern to the design of concurrent programs. I restrict myself to programs that accomplish something, and terminate. Thus I exclude, for example, the dining philosophers, and the view of operating systems in which processes concurrently run forever.

There are two general approaches.

(1) Write a sequential program to accomplish the task. Then "optimize" the program by introducing concurrency where it is possible to do so.

(2) Write a set of equations that are "concurrently true" of the desired result, and that together define the result. Then make the program executable by introducing sequence where it is needed.

The second of the approaches is called "data flow" programming, and its second step may be performed automatically. This approach is attractive, and is currently an interesting research area. In this paper I

*Received 15 October 1979; revised 19 March 1980.

take the first approach, not because it is better, but because methods of sequential programming are already well developed.

Notation

The connective "//" will be used for statements that may be executed concurrently. "Concurrent process" or "process" will mean a statement connected to another by "//". For synchronization, P and V operations on binary semaphores will be used. The statement repertoire is as follows:

(a) assignment: $x := E$

(b) selection: **if** B **then** S_1 **else** S_2 **fi**
if B **then** S **fi**

(c) loop: **while** B **do** S **od**

(d) sequence: $S_1; S_2$

We use $\overset{n}{\underset{i=1}{;}} S_i$ to stand for $S_1; S_2; \ldots; S_n$ (this is a "for" construct).

(e) concurrency: $S_1 // S_2$

This connective has higher precedence than ";". Statement grouping parentheses "{" and "}" may be used to alter the precedence. We use $\overset{n}{\underset{i=1}{//}} S_1$ to stand for $S_1 // S_2 // \ldots // S_n$.

(f) synchronization: P(sem)
V(sem)

A semaphore is a binary variable whose value is initially 1, and that is subject only to P and V operations. "P(sem)" means "wait until the value of semaphore sem is 1, then change it to 0 without interference from concurrent P operations." "V(sem)" means "change the value of semaphore sem from 0 to 1." If several P operations are waiting for the same semaphore to become 1, then when it does, an arbitrary one of the P operations will proceed and the others will continue to wait.

(g) call: N

The name N is used in place of a statement (list) that is specified in a refinement.

A refinement, which specifies the statement S that name N stands for, has the syntax $N:S$. A program is a set of refinements. Declaration of variables will be omitted.

The Algorithm

I began the exercise with the final form of a concurrent program in mind, taken from Dijkstra;[2] in this paper I shall use a structurally similar

example. One need not, of course, have the final form in mind in order to begin the design. But I did, and I display it first because the final form at which I actually arrived is surprisingly different from that at which I was aiming, and I think the difference is instructive.

An array $a[1], \ldots, a[n]$ is to be sorted in ascending order by concurrent processes $S_1, \ldots, S_{n-1}$. Each process S_i is associated with a pair of array elements $a[i]$, $a[i+1]$, and is responsible for the order of the pair. A process repeatedly checks the order of its pair, swapping the values, if necessary, to put them in order. Each process must continue until all pairs are in order because disorder in any pair may propagate to any other pair. For the purpose of controlling termination, we have boolean variables $h_1, \ldots, h_{n-1}$, one for each process. "h_i = true" indicates the need for process S_i to check the order of its pair, and hence the need for all processes to continue. The h_i are initially all true, and finally all false. Formally, the system maintains the invariant

$$\forall i \in [1 \,..\, n-1]: h_i \vee a[i] \leqslant a[i+1].$$

Except for the placement of necessary P and V operations, the program is as follows.

```
[program 0]
                n-1                    n-1
sort: // h_i: = true; // S_i
                i=n                    i=1
S_i: while ∃j: h_j
       do if h_i then if a[i] > a[i + 1]
                        then swap_i;
                             h_{i-1}: = true; h_i: = false; h_{i+1}: = true
                        else h_i: = false
                      fi
          fi
       od
```

The quantifier takes j over the range $[1 \,..\, n-1]$. In process S_1 the assignment "h_{i-1}: = true" is missing; in process S_{n-1} the assignment "h_{i+1}: = true" is missing. The refinement of "swap$_i$" is obvious.

In the absence of interference from other processes, we can see that

(a) the invariant is made true by the initial assignments.

(b) it is maintained by the body of the loop. Thus it is never destroyed.

(c) the termination condition and invariant imply the desired result: a sorted array.

To show termination, still in the absence of interference, we need a measure of computational progress. One such measure is $\#T + 2*\#S$ where $\#T$ is the number of true h_i, and $\#S$ is the maximum number of correcting swaps that can be made to sort the array. Clearly, this measure

is finite, integer-valued, and bounded below by 0. Clearly, it is decreased by each of the alternatives in the inner **if** statement, and therefore by the body of the loop if h_i is true. But when h_i is false, no progress is made; this is a "busy-wait" loop. It is not possible to have $\exists j: h_j \wedge \neg h_i$ for all processes S_i, so not all processes can simultaneously be busy-waiting. Under the assumption that those processes not busy-waiting are proceeding at a non-zero rate, the system as a whole makes progress towards termination.

To prevent interference, i.e., to ensure that the preceding arguments hold in the presence of concurrent computation, we introduce one semaphore sema_i for each array element $a[i]$, and one semaphore sem$h$ for the collection of all the h_i. The program becomes

```
[program 1]
sort: //(i=1 to n-1) h_i: = true; //(i=1 to n-1) S_i
S_i: P(semh);
    while ∃j: h_j
        do V(semh);
            if h_i then P(sema_i); P(sema_{i+1});
                        if a[i] > a[i + 1]
                          then swap_i;
                                P(semh); h_{i-1}: = true; h_i: = false;
                                h_{i+1}: = true; V(semh)
                          else P(semh); h_i: = false; V(semh)
                        fi; V(sema_i); V(sema_{i+1})
            fi; P(semh)
        od; V(semh)
```

It is essential for correctness that the critical regions (between a P and corresponding V) contain as much as they do (by removing the assignment "h_i: = false" from the first alternative of the inner **if**, the grain can be made finer). It is important for efficiency that they contain no more. If, for example, the critical region for setting the three hs also contains the swap, the algorithm is quadratic even with $n - 1$ processors, instead of the hoped-for linear algorithm.

Now no process interferes with (the proof of) any other (proof left to the reader). The system is free from deadlock (proof to be given later), but the danger of starvation is present. Consider two processes that are busy-waiting. If the implementation grants precedence to their alternating P(sem$h$) operations, all other processes are blocked. This is, of course, unfair; in a fair implementation, e.g., one that schedules waiting P operations as first-come-first-served, starvation will not occur.

I do not consider a program to be correct if it relies on the fairness of a scheduler. I take this to be analogous to the sequential program, taken from Dijkstra[1] in guarded-command notation:

```
go_on := true; x := 1;
do go_on → x := x + 1
 [] go_on → go_on := false
od
```

which, for termination, relies on the fairness of a daemon in selecting guards (a true guard must not go forever unselected). Dijkstra's semantics allow, and his programming methods produce, only programs that are immune to even a malicious daemon. The semantics of the above program make it equivalent to "abort"; I demand the same for a concurrent program in which deadlock or starvation is a possibility. (See Dijkstra[1], ch. 9 and p. 214, and Hoare,[5] p. 676.)

The goal of this exercise, of which we now resume pursuit, is to see how to develop a correct program based on the algorithmic idea presented in this section.

The Development

The development of the concurrent sorting program begins with the well-known sequential insertion sort. The sort is expressed as it may result from the programming methods of Hehner,[4] except for the use of the abbreviated "for" notation.

[program 2]

sort: $\overset{n-1}{\underset{i=1}{;}} S_i$

S_i: **if** $a[i] > a[i+1]$ **then** swap_i; S_{i-1} **fi**

In S_1, the call S_0 is missing, or else we add the refinement S_0: skip. The name "S_i" means "Put the pair $a[i]$, $a[i+1]$ in order without destroying the order of any pair $a[j]$, $a[j+1]$ for $j < i$."

The object now is to replace ";" with "//" wherever possible. To do this, we must remove from the program its dependence on the sequential ordering. That dependence is expressed clearly in the inequality "$j < i$" at the end of the previous paragraph. Each S_i is obliged to put one pair in order without destroying the order of pairs that have already been put in order. The swap may destroy the order of pairs on either side, but we were concerned to reinstate, by calling S_{i-1}, only one of these. The following program is correct even if sort calls the S_i (sequentially) in arbitrary order.

[program 3]
sort: $\overset{n-1}{\underset{i=1}{;}} S_i$
S_i: **if** $a[i] > a[i+1]$ **then** swap$_i$; S_{i-1}; S_{i+1} **fi**

In S_1 the call S_0 is missing and in S_{n-1} the call S_n is missing, or else we add the refinements S_0: skip and S_n: skip. The name "S_i" means "Put the pair $a[i]$, $a[i+1]$ in order without destroying the order of any pair." We can now make the transition to a concurrent program, supplying the necessary protection against interference.

[program 4]
sort: $\overset{n-1}{\underset{i=1}{//}} S_i$
S_i: $P(\text{sema}_i)$; $P(\text{sema}_{i+1})$;
if $a[i] > a[i+1]$ **then** swap$_i$; $V(\text{sema}_i)$; $V(\text{sema}_{i+1})$; $S_{i-1}//S_{i+1}$
else $V(\text{sema}_i)$; $V(\text{sema}_{i+1})$
fi

It is clear that the program is free from interference. That the program is free from deadlock is proved as follows. Assume that some process is blocked (waits forever) at one of its P operations, say $P(\text{sema}_j)$, because of deadlock. This implies that some other process (possibly another incarnation of the "same" process) has already performed the operation $P(\text{sema}_j)$ but is blocked from performing $V(\text{sema}_j)$. The text of the program tells us that this other process is blocked at $P(\text{sema}_{j+1})$. Repeating the argument, this implies that some process is blocked at $P(\text{sema}_{j+2})$, etc. But there are only n semaphores, hence we have a contradiction. (This proof applies also to the previous concurrent program.)

It follows from the rules of program composition used to produce it that the program is free from starvation. One process calls another only after making progress – here the computational progress of a correcting swap. Thus no non-terminating loops are formed. No matter how unfair a scheduler may be, a waiting process has to wait at most until all competing processes have terminated. (I take this to be further confirmation of the thesis of Hehner.[4])

Before we compare this solution with the original solution, we make one further transformation based on the idempotence of S_i with respect to the order of its pair. Though we have not given a formal mathematical semantics for the concurrent connective, we shall nonetheless define idempotence formally in terms of it.

Definition
Statement S is idempotent with respect to predicate P if

$$wp(S//S, P) = wp(S, P),$$

where wp is the weakest precondition predicate transformer.[1]

Informally, a statement is idempotent with respect to the result that it is intended to establish if calling it twice (concurrently) is no different from calling it once. Calling S_i once puts $a[i]$, $a[i+1]$ in order; a second call, in the absence of intervening computation, does nothing.

The transformation is as follows. For idempotent process S_i, introduce boolean variable h_i that is initially false to indicate that S_i has not yet been called. Replace calls of S_i (as many as desired) with

$$\textbf{if } \neg h_i \textbf{ then } h_i := \text{true}; S_i \textbf{ fi}$$

and begin the refinement of S_i with

$$h_i := \text{false}.$$

A would-be caller, finding that S_i has already been called and is waiting to be executed, does not call S_i. (We are now assuming that the setting and testing of a single boolean variable are indivisible operations.) Our final solution follows.

[program 5]

sort: $\overset{n-1}{\underset{i=1}{//}} h_i := \text{false}; \overset{n-1}{\underset{i=1}{//}} IS_i$

IS_i: **if** $\neg h_i$ **then** $h_i := \text{true}; S_i$ **fi**

S_i: $h_i := \text{false}; P(\text{sema}_i); P(\text{sema}_{i+1});$
if $a[i] > a[i+1]$ **then** $\text{swap}_i; V(\text{sema}_i); V(\text{sema}_{i+1}); IS_{i-1}//IS_{i+1}$
else $V(\text{sema}_i); V(\text{sema}_{i+1})$
fi

This program need not be proven correct, or understood as a whole; the proof and understanding of the previous version [program 4], and separately of the transformation, are sufficient.

The transformation based on idempotence is sometimes an optimization. In this program, it is not clear that an advantage has been gained, and therefore the simpler version [program 4] is preferred. We have made it only for the purpose of comparison in the next section. It is an important optimization when the second, redundant execution of an idempotent statement would be expensive.

Comparison of the Two Styles

In this comparison, the iterative style is represented by [program 1], hereafter referred to as the "original" program, and the recursive re-

finement style is represented by [program 5], hereafter referred to as the "final" program. The two important differences are in the grain of concurrency, and in the busy-waiting. Let us dispose of two superficial differences first.

(1) The first line of the final program can be optimized to

$$\text{sort}: \underset{i=1}{\overset{n-1}{//}} h_i = \text{true}; \underset{i=1}{\overset{n-1}{//}} S_i$$

the same as in the original program.

(2) In the original program, the statements "$h_{i-1} := \text{true}$" and "$h_{i+1} := \text{true}$" can be replaced by IS_{i-1} and IS_{i+1} respectively, where

$$IS_i: \textbf{if } \neg h_i \textbf{ then } h_i := \text{true } \textbf{fi}.$$

(3) The difference in the grain of concurrency is to some extent a difference in the styles, and to some extent merely a difference in the particular representatives. It is essential to the iterative style that in the original program, when testing "$\exists j: h_j$", the h_j are prevented from changing. By contrast, in the final program, testing one h_j does not exclude another from changing. This difference is to the credit of the style of the final program.

The critical region for the testing and swapping of array elements is larger in the original program: there it includes also the assignments to three hs. This difference is not to the credit of either style. The original could be finer-grained; for example, this critical region can be replaced by

$P(\text{sem}a_i)$; $P(\text{sem}a_{i+1})$;
if $a[i] > a[i+1]$
  then swap$_i$; $V(\text{sem}a_i)$; $V(\text{sem}a_{i+1})$;
    $P(\text{sem}h)$; $h_{i-1} := \text{true}$; $h_{i+1} := \text{true}$; $V(\text{sem}h)$
  else $P(\text{sem}h)$; $h_i := \text{false}$; $V(\text{sem}h)$; $V(\text{sem}a_i)$; $V(\text{sem}a_{i+1})$
fi

but the mathematical proof is then beyond my ability.*

A finer grain of concurrency is not always more efficient. In the final program, we can save a would-be caller of S_i from making a superfluous call by moving the statement "$h_i := \text{false}$" inside the critical region. The coarser grain is more efficient! If P and V operations are not too expensive, we may introduce semaphores semh_i, and revise the final program's second line as

IS_i: $P(\text{sem}h_i)$; **if** $\neg h_i$ **then** $h_i := \text{true}$; $V(\text{sem}h_i)$; S_i
  else $V(\text{sem}h_i)$
fi

*Dijkstra[3] has proven correct except for starvation a structurally identical program using n auxiliary boolean arrays and an invariant defined as the minimal solution of a pair of equations.

to prevent other would-be callers from making superfluous calls. This coarser grain is not required for correctness, but it may improve the efficiency.

(4) The "real work" of process S_i is to test the order of one pair of elements and, if necessary, swap them. In both the original and final programs this is done repeatedly, but the two programs differ (and this is their main difference) in how they control the repetition. In the original program, control is in a busy-wait loop until either there is some real work to do, or the process can terminate. It must not be thought that this busy-waiting harmlessly occupies an otherwise unoccupied processor; that thought would be mistaken for two reasons. (a) We must not assume that we have one processor for every process. The connective "//" specifies what concurrency is allowable so that an implementation can make the best use of the number of processors available, however many or few that may be.* (b) Even if there are many processors, when one process tests the expression of its **while** loop, it excludes other processes from changing the hs (and, unfortunately, from testing them), thus slowing them down. Starvation is just the extreme case of this inefficiency.

In the final program, a process does not test whether there is real work for it to do; it is called only when there is real work. After doing the work, it terminates; if there is more work later, it will be called again later. This is more straightforward, and more efficient.

One More Example

In the first example, we developed a concurrent sorting program from a sequential insertion sort. In this example, we begin with a sequential bubble sort.

An insertion sort, in the worst case, makes the following sequence of calls:

$$S_1;$$
$$S_2; S_1;$$
$$S_3; S_2; S_1;$$
$$S_4; S_3; S_2; S_1$$

for an array of five elements.† In order to allow progress along all rows

*Some language designers have suggested a sequential programming construct called "coroutines." I prefer to consider coroutines as an implementation of multiple processes on a single processor. In general, the implementation becomes more concurrent and less coroutine-like when more processors are available.

†According to Knuth,[6] it is this pattern of comparisons that defines an insertion sort, and not whether movement of elements is by swapping or otherwise.

concurrently, we augmented this pattern. A bubble sort makes the following sequence of calls:

$$S_1; S_2; S_3; S_4$$
$$S_1; S_2; S_3;$$
$$S_1; S_2;$$
$$S_1$$

for an array of five elements. Our starting sequential program is

[program 6]
sort: $\overset{n-1}{\underset{j=1}{;}} S_1$
S_i: **if** $a[i] > a[i+1]$ **then** swap$_i$ **fi**;
if $i + j < n$ **then** S_{i+1} **fi**.

In order to allow progress along all rows concurrently, we augment this pattern by extending each row to the end. As before, we accomplish this in two steps. First we write a sequential program in which j can take values from 1 to $n - 1$ in any order.

[program 7]
sort: $\overset{n-1}{\underset{j=1}{;}} S_1$
S_i: **if** $a[i] > a[i+1]$ **then** swap$_i$ **fi**;
S_{i+1}

In S_{n-1} the call S_n is missing, or else we add the refinement S_n: skip. Now we proceed to concurrency, placing the P and V operations as required.

[program 8]
sort: $\overset{n-1}{\underset{j=1}{//}} \{P(\text{sem}a_1); S_1\}$
S_i: $P(\text{sem}a_{i+1})$;
if $a[i] > a[i+1]$ **then** swap$_i$ **fi**;
$V(\text{sem}a_i); S_{i+1}$

This second example serves both a positive and a negative purpose. It is a second illustration that a starvation-free concurrent program can be developed from a sequential program expressed in recursive refinement form. It is also a warning that the placement of synchronization to preserve the sequential proof from concurrent interference can be difficult. We have not addressed that issue; instead we refer the reader to Lamport[7] and Owicki and Gries.[8]

Conclusion

Sequential and concurrent programming are usually placed in different worlds: one begins a problem knowing which kind of program to produce, and uses different programming methods and notations for each kind. There is usually no transition from one to the other.

This paper shows how, at least in some cases, a concurrent program can be achieved in steps from a sequential one. Our sequential style is recursive refinement, and as a result, our concurrent programs provide the following benefits.

(a) The well-understood programming methods and proof techniques for sequential programs allow us to begin with a complete, correct program.

(b) The resulting concurrent program is starvation-free without appealing to a fair scheduler.

(c) Less synchronization is required, and as a result, the program is more efficient. This is because some of the control has moved from the testing of shared variables to the mutually recursive calling structure.

The choice of synchronization primitives is not at issue. The use of P and V allowed the above points to be made with a minimum of explanation, but the points remain valid with other primitives.

References

(1) E.W. Dijkstra, *A discipline of programming*. Englewood Heights, Prentice-Hall, NJ: 1976.

(2) E.W. Dijkstra, "On making solutions more and more fine-grained." Rept. EWD622, May 1977.

(3) E.W. Dijkstra, "Finding the correctness proof of a concurrent program." Rept. EWD640a, 1977.

(4) E.C.R. Hehner, "*do* consider *od*: a contribution to the programming calculus." Acta Informatica, vol. 11, 1979, 287–304.

(5) C.A.R. Hoare, "Communicating sequential processes." CACM, vol. 21, no. 8, August 1978, 666–677.

(6) D.E. Knuth, *The art of computer programming*, Vol. 3, *Searching and sorting*. Reading, MA: Addison-Wesley, 1973.

(7) L. Lamport, "Proving the correctness of multiprocess programs." IEEE *Transactions on Software Engineering SE*-3, no. 2, March 1977, 125–143.

(8) S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs I." Acta Informatica, vol. 6, 1976, 319–340.