Derivation of a Simple Synchronization Algorithm

Leslie Lamport

Digital Equipment Corporation
Systems Research Center

17 November 1986
revised 4 February 1987

What

The following synchronization problem was posed to me by Chuck Thacker. Consider a collection of *worker* processes that communicate with one another by sending messages, where each worker has its own input-message buffer. A worker responds to an input message by performing some computation and sending a (possibly empty) set of messages to other workers. The problem is to detect when the system has reached *termination*, which occurs when there is no more computing to be done. This is, of course, a standard distributed computing problem. However, in Thacker's problem, the system is implemented on a shared-memory multiprocessor, and there is a single *detector* process devoted to detecting termination.

This shared-memory version of the termination detection problem is much easier than the distributed version, and, with Jim Saxe's help, I had little difficulty devising a simple, efficient, and unsurprising solution. What I did find surprising is that, although it seems like a common enough problem, I could think of no standard algorithm that solves it; I had to devise a new solution. Moreover, I did so by the same intuitive, trial-and-error approach that I have been using for fifteen years. I made fewer trials and errors than I would have fifteen years ago, but that was because my intuition has been refined by experience, not because I used any rigorous methods for deriving the algorithm. I therefore felt it would be interesting to see if I could have derived the algorithm from general principles. The result follows.

How

I assume that the detector halts when it detects termination. There are two properties required of the algorithm:

- If the detector has halted then the system has really terminated. This is a safety property.
- If the system has terminated, then the algorithm eventually detects that it has. This is a liveness property.

Any sensible attempt at a solution seems to satisfy the liveness property, so I will consider only the safety property, which I denote *Safe*. Stated more pedantically, it is:

$$\mathit{Safe}\text{: detector halted} \Rightarrow \mathit{termination}$$

The predicate *termination* still has to be defined. The exact definition is not important; we require only that *termination* imply that all workers have finished their computations and can generate no further messages. We are free to choose the most convenient definition of *termination* with this property.

A safety property like *Safe* is proved by finding an invariant that is true initially and implies *Safe*, where an invariant is a predicate I with the property that any atomic action executed when I is true leaves I true. I will construct the invariant and the algorithm together.

It is usually helpful to start with a coarse-grained program (one with big atomic actions) and transform it to a suitably fine-grained one. Coarse-grained programs are easier to understand, and the fine-grained program can't be correct unless the coarse-grained version is. So, we begin by assuming that a worker's entire operation of removing an input message, computing, and writing output messages is a single atomic action. The following trivial algorithm is then obtained by having the detector repeatedly examine all queues with a single atomic action. Angle brackets enclose atomic actions, $Q[i]$ denotes worker i's input queue, and an action of the form $\langle P \rightarrow S \rangle$ is performed by executing S only if P is true.

Algorithm 1:

```
Worker i                                    Detector
repeat forever                              repeat ⟨skip⟩
  $\langle Q[i] \neq \emptyset \rightarrow$                         until $\langle \forall i : Q[i] = \emptyset \rangle$
    remove head of $Q[i]$;
    compute;
    add msgs to tail of other queues $\rangle$
```

For this program, no further computation is possible if all input queues are empty. We therefore define *termination* to be true if and only if all queues are empty, since every queue is some worker's input queue. In other words, *termination* equals $\forall i : Q[i] = \emptyset$.

The assertion *Safe* itself is the invariant used to prove the correctness of Algorithm 1. It is implied by the initial condition and it obviously implies itself. The workers leave *Safe* invariant because they cannot make *termination* false once it becomes true. The detector leaves *Safe* invariant because it can halt only if *termination* is true. Hence, *Safe* is an invariant of Algorithm 1.

Let us now attempt to refine this program. Our first observation is that *Safe* will remain true throughout the execution if the worker adds its messages to the output queues before removing the message from its input queue. This leads to the following refinement of Algorithm 1:

Algorithm 2:

```
Worker i                                      Detector
repeat forever                                repeat ⟨skip⟩
$\alpha$: ⟨$Q[i] \neq \emptyset \rightarrow$                      until ⟨$\forall i : Q[i] = \emptyset$⟩
      compute;
      add msgs to tail of other queues ⟩;
$\beta$: ⟨remove head of $Q[i]$⟩
```

For this algorithm, *termination* is defined to be true if and only if every worker is at statement α with its input queue empty. Letting $at(\alpha_i)$ be the predicate asserting that worker i is at statement α, we can write *termination* as $\forall i : at(\alpha_i) \wedge Q[i] = \emptyset$

Although *Safe* is true throughout the execution of Algorithm 2, it fails to be an invariant because of a technicality. To be an invariant, *Safe* must be left true by executing a program action in *any* state that satisfies it, not just in states that can be reached by executing the algorithm. Consider a state in which worker i is at statement β and its input queue is empty. (This is an unreachable state.) The action of removing a message from an empty queue is undefined, so executing β could do anything—including falsify *Safe*. We can rule out such unreachable states by strengthening *Safe*, obtaining the following invariant.

I2: $\mathit{Safe} \wedge (at(\beta_i) \Rightarrow Q[i] \neq \emptyset)$

It is easy to check that I2 is an invariant of Algorithm 2, is true initially, and implies *Safe*.

We now try to refine the detector so that instead of examining all queues as a single action, it examines them one at a time. The following is essentially the only way to do this, where $\mathcal{W}$ denotes the set of all workers:

```
Detector
$\mathcal{U} := \mathcal{W}$;
while ⟨$\mathcal{U} \neq \emptyset$⟩
  do ⟨choose $u$ in $\mathcal{U}$;
      if $Q[u] = \emptyset$ then $\mathcal{U} := \mathcal{U} - \{u\}$
                   else $\mathcal{U} := \mathcal{W}$ ⟩
```

The set of queues $\mathcal{U}$ contains workers whose queues the detector has not yet examined. The detector can race ahead and finish whenever all the queues $Q[u]$ with u in $\mathcal{U}$ are empty. Therefore, for *Safe* to remain true, the following predicate would have to be invariant:

I2a: I2 $\wedge$ $[(\forall u \in \mathcal{U} : Q[u] = \emptyset) \Rightarrow \mathit{termination}]$

However, I2a is not left invariant by the workers. A worker can falsify its second conjunct by removing the last message from the input queue of a worker in $\mathcal{U}$ after having added a message to the input queue of a worker not in $\mathcal{U}$. It is easy to see that this algorithm is incorrect; the detector could observe all the queues empty when the system has not yet terminated. This is the case even for the worker of Algorithm 1 that removes and adds the messages as a single atomic action.

A little thought reveals that there is no way to correct this algorithm without additional communication between the workers and the detector. The trouble arises because the last conjunct of I2a is falsified when a worker removes a message from its input queue. Suppose we weaken I2a by disjoining a Boolean b to this conjunct, yielding

I2b: I2 $\wedge$ $[(\forall u \in \mathcal{U} : Q[u] = \emptyset) \Rightarrow (\mathit{termination} \vee b)]$

To maintain the invariance of I2b, a worker must make b true whenever it removes the last element from a queue. It is easier to set b true unconditionally than to check if the queue is empty, so we let the worker set b true whenever it removes any message from its input queue.

Recall that I2 includes the conjunct *Safe*, which asserts that if the detector has halted then the system has terminated. To keep the detector from making *Safe* false, it must not terminate if b is true. This suggests the following algorithm:

Algorithm 3:

```
Worker i
repeat forever
α ⟨$Q[i] \neq \emptyset \rightarrow$
      compute;
      add msgs to tail of other queues ⟩;
β: ⟨$b$ := true ;
    remove head of $Q[i]$⟩
```

```
Detector
repeat
   ⟨$\mathcal{U} := \mathcal{W}$;
    $b$ := false ⟩;
   while ⟨$\mathcal{U} \neq \emptyset$⟩
     do ⟨choose $u$ in $\mathcal{U}$;
          if $Q[u] = \emptyset$
            then $\mathcal{U} := \mathcal{U} - \{u\}$
            else $\mathcal{U} := \mathcal{W}$;
                   $b$ := false ⟩
until η: ⟨$\neg b$⟩
```

This algorithm can falsify I2b in an uninteresting case—namely, if the detector executes its **until** test η in a state with $\mathcal{U} \neq \emptyset$ and with b and *termination* both false. This case is uninteresting because it cannot occur in an actual execution; the detector can reach η only when $\mathcal{U} = \emptyset$. Such an impossible case is ruled out by strengthening the invariant, which we do as follows:

$$\begin{aligned} \text{I3: } & \text{I2} \wedge [(\forall u \in \mathcal{U} : Q[u] = \emptyset) \Rightarrow (\mathit{termination} \vee b)] \\ & \quad \wedge [\mathit{at}(\eta) \Rightarrow \mathcal{U} = \emptyset] \end{aligned}$$

We must now check that Algorithm 3 maintains the invariance of I3. It is clear that the workers still leave I2 invariant and they don't affect the third conjunct of I3. A worker's α action cannot falsify the second conjunct of I3, and its β action makes the second conjunct true by setting b true. Hence, the workers leave I3 invariant.

The detector can falsify I2 only by making *Safe* false, which it can do only by terminating. However, the last two conjuncts of I3 imply that the **until** test can find b false only when the system has terminated. Thus, starting in a state satisfying I3, the detector cannot falsify I2. The detector obviously cannot falsify the third conjunct of I3. It can falsify the second conjunct only when it decreases $\mathcal{U}$ or sets b false. However, it decreases $\mathcal{U}$ only by removing a worker with an empty queue, which cannot falsify the conjunct. When it sets b false it also sets $\mathcal{U}$ equal to the set of all workers. If all the queues are empty, then I2 implies that every worker is at statement α. Since *termination* is defined to be $\forall i : \mathit{at}(\alpha_i) \wedge Q[i] = \emptyset$, it is true if all the queues are empty. Hence, the second conjunct of I3 remains true in this case, completing the proof that I3 is an invariant of Algorithm 3.

For future reference, observe that executing the detector's **else** clause maintains the invariance of I3 even if $Q[u] = \emptyset$. In other words, the detector must execute the **else** clause when the **if** test is false, but it can execute either clause when the **if** test is true. (Executing the **else** clause when $Q[u] \neq \emptyset$ can affect liveness, but that does not concern us.)

We next refine the worker to make the setting of b true and the removal of the message from its input queue be two separate actions. To maintain the truth of the last conjunct of I3, it is clear that the worker should first set b true, then remove the message, which gives the following algorithm:

Algorithm 4:

```
Worker i
repeat forever
α ⟨$Q[i] \neq \emptyset \rightarrow$
      compute;
      add msgs to tail of other queues ⟩;
β: ⟨$b :=$ true⟩;
γ: ⟨remove head of $Q[i]$⟩
```

```
Detector
repeat
   ⟨$\mathcal{U} := \mathcal{W}$;
    $b$ := false ⟩;
   while ⟨$\mathcal{U} \neq \emptyset$⟩
     do ⟨choose $u$ in $\mathcal{U}$;
          if $Q[u] = \emptyset$
             then $\mathcal{U} := \mathcal{U} - \{u\}$
             else $\mathcal{U} := \mathcal{W}$;
                    $b$ := false ⟩
until η: ⟨$\neg b$ ⟩
```

We must redefine *termination* so it is true if the only message in any input queue is one that is about to be removed by statement γ. In other words,

$$\textit{termination} \;\equiv\; \forall i : (at(\alpha_i) \wedge Q[i] = \emptyset) \vee (at(\gamma_i) \wedge \text{tail}(Q[i]) = \emptyset)$$

We can apply a standard trick to show that Algorithm 4 satisfies the same safety property as Algorithm 3. Define $Q'[i]$ to equal $Q[i]$ except when worker i is at statement γ, in which case it equals the tail of $Q[i]$. In other words, $Q'[i]$ is an imaginary version of the real input queue $Q[i]$ such that the worker removes the message from the head of $Q'[i]$ in the same action with which it sets b true. Let I3$'$ be the formula I3 with $Q[i]$ replaced everywhere by $Q'[i]$, and with *termination* defined as above rather than with the definition used in Algorithm 3. I claim that I3$'$ is an invariant of Algorithm 4.

The proof that Algorithm 4 leaves I3$'$ invariant is essentially the same as the proof that Algorithm 3 leaves I3 invariant and is left to the reader. Here, I will explain intuitively why we expect I3$'$ to be invariant. A worker changes $Q'[i]$ in Algorithm 4 exactly the same way it changes $Q[i]$ in Algorithm 3. Since Algorithm 3 leaves I3 invariant, and I3$'$ is the same as I3 except for the substitution of $Q'[i]$ for $Q[i]$, the workers in Algorithm 4 should leave I3$'$ invariant. By similar reasoning, the detector would also leave I3$'$ invariant if its **if** condition were replaced by $Q'[u] = \emptyset$. However, $Q'[u]$ is always a subset of $Q[u]$, so $Q[u] = \emptyset$ implies $Q'[u] = \emptyset$. Hence, using the test $Q[u] = \emptyset$ means that the detector may execute the **else** clause even though the condition $Q'[u] = \emptyset$ is true, but it will never execute the **then** clause if the test $Q'[u] = \emptyset$ is false. As we observed above, Algorithm 3 maintains the invariance of I3 even if the **else** clause is executed when the **if** condition is true. Therefore, since the detector would maintain the invariance of I3$'$

if it used the **if** condition $Q'[u] = \emptyset$, it maintains the invariance of I3$'$ with the actual condition $Q[u] = \emptyset$.

The intuitive argument that Algorithm 4 leaves I3$'$ invariant comes close to being a formal proof, but it ignores the fact that Algorithm 4 has an extra control point γ and a slightly different definition of *termination.* One could prove formally that Algorithm 4 correctly implements Algorithm 3. However, it is easier to prove invariance directly. The purpose of the intuitive argument was to indicate how one arrives at such an invariant. Observe the technique of modifying the invariant of the coarser-grained algorithm by replacing a variable $Q[i]$ with a state function $Q'[i]$ whose value changes in the finer-grained algorithm the same way the value of $Q[i]$ changes in the coarser-grained one. This technique of replacing variables with state functions always works when the refinement does not significantly change the behavior of the algorithm. In cases such as the refinement from Algorithm 3 to Algorithm 4, where the the algorithm's behavior is changed, the technique may or may not work.

We could further refine Algorithm 4—for example, by splitting statement α so the test, the "compute" statement, and the sending of each message are separate actions. However, the behavior of the resulting algorithms remain essentially the same so long as we maintain the atomicity of the operations of sending a message, removing a message from a queue, and testing if a queue is empty. The invariant of the refined algorithm can be obtained directly from I3$'$ by the technique of replacing variables with state functions.

A more interesting refinement is to replace the single variable b with a more complicated data structure, so that setting b and testing its value are no longer atomic actions. For example, we could maintain a separate bit $b[i]$ for each worker and define b to be the disjunction of the $b[i]$. Worker i would make b true by setting $b[i]$ true; the detector would read or write b by reading or writing all the $b[i]$. The algorithm works with any reasonable implementation of b, though defining what "reasonable" means and giving a formal proof require the introduction of concepts that are beyond the scope of this short note.

Observe that as we refined the algorithm, our correctness condition became more complicated. (The condition was stated in terms of *termination*, whose definition also had to be refined.) We have ignored the question of whether this condition really implies that the algorithm is correct. It is characteristic of conventional program verification that the correctness condition is stated in terms of the program itself. Stating and proving a more abstract correctness property that is independent of the implementation would have required an extended detour into the realm of concurrent program specification.

Why

One should be skeptical of after-the-fact derivations of algorithms, and there is good reason for skepticism in this case. Why did I refine the worker before refining the detector? What led me to add the Boolean *b* to I3? The derivation is not as straightforward as the exposition may suggest; there is plenty of intuition at work. Still, when comparing this derivation with the way it really happened, I am struck by how much of my original intuition can be explained in terms of maintaining an invariant. For example, consider the decision to have the worker set *b* before removing the message from its input queue. My original intuition told me the actions should occur in this order to make it easier for the detector to discover the existence of a nonempty queue. In the above derivation, this ordering was obvious from the need to maintain the invariance of the last conjunct of I3.

Before writing this note, I thought I derived concurrent algorithms by behavioral reasoning—reasoning about the possible orders in which actions could occur. I regarded assertional reasoning—reasoning in terms of invariants—to be an unintuitive process that I employed only afterwards in the correctness proof. I now understand that the intuition I use when devising a new algorithm involves preconscious assertional reasoning. The intuition that led me to avoid false termination detection by having the worker set *b* before removing the input message was based upon a fuzzy notion of invariance, not upon any consideration of sequences of actions.

I still don't believe that interesting algorithms can be derived using formal methods alone; one does need intuition. It was intuition that led me to add the Boolean *b*. However, I now believe that thinking in terms of invariance can be useful even at the earliest stage of algorithm development. Rather than being counter to intuition, assertional reasoning is often the basis for intuitive reasoning. Making the assertional reasoning explicit should aid intuition.