Christian-Albrechts-Universität zu Kiel
Institut für Informatik und Praktische Mathematik
Prof. Dr. W.-P. de Roever
Martin Steffen, Immo Grabe


# Verteilte Algorithmen

Wintersemester 2003/04

# Serie 8

7. Januar 2004

**Thema :** **(Aufgaben mit Lösungshinweisen)**

**Ausgabetermin: 7. Januar 2004**

**Abgabe: 12. Januar 2004**

**Aufgabe 1 (Terminierung (6 Punkte))** Terminierungsbeweise erfordern andere Argumente als die für Safety oft angewandten Induktionen über die Anzahl der Runden oder die Anzahl der Schritte. Eine Möglichkeit ist, *wohlgeordnete Mengne* zu verwendenrt. Eine *Wohlordnung* (*well-founded set*) ist eine Halbordnung, bei der es keine unendlich absteigenende Ketten gibt, d.h., keine unendliche Folge $n_1 > n_2 > n_3 > \ldots$.

- Zeigen Sie, daß das Kreuzprodukt von $n$ Wohlordnungen eine Wohlordnung ist.
- Beweisen Sie, daß der aynchrone LCR-Algorithmus (Le Lann, Chang, Roberts) für Leader-Election im Ring terminiert. Überlegen Sie sich dazu ein geeignetes Terminierungsmaß.

**Solution:**

1. Let's first define how to lift an ordering to products. Basically, there are two intuitive ways to do it: *lexicographic* or *pointwise.* Both can be defined inductively on the dimension of the product. Let $<$ be the strict order on the underlying domain.[1] Then we can define the strict *lexicographic order* $(A^n, <_l)$ as the smallest relation given by:

   $$\frac{a < b}{a\vec{a} <_l b\vec{b}}\ \textsc{First} \qquad \frac{\vec{a} <_l \vec{b}}{a\vec{a} <_l a\vec{b}}\ \textsc{Rest}$$

   Alternatively, we could define the non-strict order $\leq_l$, adding an axiom $\epsilon \leq_l \epsilon$, when $\epsilon$ is the empty tuple. Anyway, it's easy to show that $\leq_l$ is reflexive, transive, and anti-symmetric, using the fact the underlying $\leq$ is.

   Not to the question of infinite descending $<_l$-chains. The base case of $n = 1$ is immediate, since $<_l$ on $A^1$ equals $<$, wich is a well-ordering. For $n + 1$, assume for a

[1] For notational simplicity, we assume a uniform product $A^n$; the very same construction works for $\prod_i A_i$, where each $A_i$ is equipped with a well-ordering $<_i$ of its own.

contradiction, that there exists an infinite descending chain $a_0\vec{a}_0 >_l a_1\vec{a}_1 \ldots$. Therefore there must be an *infinite* number of applications of First or Rest (or both, of course). That's impossible, however, since $<$ is a well-ordering by assumption, and $<_l$ is a well-ordering by induction.

We can also lift $\leq$ *pointwise* to $A^n$. An inductive definition of $<_p$ looks as follows:

$$\frac{a < b}{a\vec{a} <_p b\vec{a}}\ \textsc{First}_1 \qquad \frac{a < b \qquad \vec{a} <_p \vec{b}}{a\vec{a} <_p b\vec{b}}\ \textsc{First}_2 \qquad \frac{\vec{a} <_p \vec{b}}{a\vec{a} <_p a\vec{b}}\ \textsc{Rest}$$

It states that at least one component most properly decrease, maybe more than one. The argument that $<_p$ is again a well-ordering works as for $<_l$, using induction.

2. Both well-orderings can be used to prove termination of the LCR-algorithm. Which one is appropriate depends on which information about the state of the processes one measures.

   **Travel length of token** This is the easier termination measure, in the sense that it works with the *point-wise ordering.* Each process sends around it's user id as token in, say, clock-wise direction around the ring, until it is swallowed by a process with higher id. We can take the *maximal traveling length* for each token as measure, where we must not forget to count the *channel process.* Initially we can thus take as measure vector:[2]

   $$(2n, 2n, \ldots, 2n)$$

   Each send or recieve action decreases exactly one place in the vector (= *pointwise*). In case of a receive action, when the received value is swallowed and not relayed, we can decrease the corresponding value to 0.

   **Queue lengths** Alternatively, one can take a more complex measure, namely the vector of all queue length for all processes and channels. In this set-up, the point-wise ordering does *not work,* because removing a message from one queue puts the message in another one, whose value thus goes up.

   If we order the processes and channels in the order given by the directed graph, we obain an $2n$-tuple. Each time a message is send, it decreases a queue length in slot, say $i$, and increases it at $i + 1$. In case the massage it swallowed, it only decreases.

   That makes it possible to use the *lexicographic* order, *provided* that the last slot in the array, at position $2n - 1$ —we assume that the numbering starts at 0— never *increases* his "right neighbor" which would again be process 0 (we have a ring). If we place the process with the maximal user id, which never relays messages, as first at the beginning of the array, then that's ok.

[2] One could make it more complex in that one argues, that the distance from any process to the process with the maximal id, who swallows everything, is taken, but that's not necessary. Indeed, one can go even a step further: when one assumes a particular arrangement of id's as given, then one knows a priori the travel distance of each token which is determined by the arrangement, and can take that as a more refine measure, instead of $(2n, 2n, \ldots, 2n)$.

**Aufgabe 2 (HS Leader Election (6 Punkte))** Bearbeiten Sie Aufgabe 15.3(a) + (b) auf Seite 525 von [1], d.h., programmieren Sie eine asynchrone Variante des Leader-Election Algorithmus' von Hirschberg und Sinclair uns beweisen Sie ihn korrekt.

**Solution: 3 + 3 Punkte.** The algorithm is not much different from the synchrous one (cf. [1, p. 33]). We have, of course, adhere to the declaration conventions for I/O automata etc, but the basic idea remains the same. The `send+` and `send-` storage cells of the synchronous algorithm are replace by queues, as is typical for asynchronous algorithms. Furthermore, of course, additionaly channel I/O automata are added between, in this case, each neighboring automata in the ring.

One difference, of course, is that in the asynchronous setting, a process cannot rely on the fact that the two exploratory message in a phase come back as in-messages at the same time. Therefore, the process must "*synchronize*" by waiting until it has received both. In the synchronous setting, this synchronization was for free by the fact that the two exploratory messages travel tracel the same distance and the the execution proceeds in rounds.

```
Process(i)    // HS

....
 // Transitions

 send(m)_{i,i+1}
   precondition: m is first on send+
   effect      : remove first element from send+
                 and send it to i+1

send(m)_{i,i-1} .... analogous

receive(m)_{i-1,i}
  precondition:
     message from (i-1) =  (v,out,h)              // out-reception from lower neighbor

  effect:
    case v > u and h > 1:  add (v,out,h-1) to send+   //  relay out
         v > u and h = 1:  add (v,in,1)    to send-   //  reflect back
         v = u          :  status :=  chosen          // getting own id -> leader
                                                      // if v<u: out is ''dropped''
    endcase
    ...

receive(m)_{i-1,i}
  precondition:
     message from (i-1) =  (v,in,h)              // in-reception from lower neighbor
     precondition
         message from i-1 is (v,in,1) and v ≠ u  // in reception
         effect  add (v,in,1) to send+                  // relay in

     message from (i-1) =  (u,in,1)              // in-reception from lower neighbor
     precondition                                // I use pattern matching...
     effect       returned_from_left:= true


phase_i():
    precondition:
        returned_from_left = true and returned_from_right = true
     effect:
        returned_from_right := false;
        returned_from_left  :- false;
        phase_i      := phase_i + 1;
        add (u, out, 2^phase) to send+;
```

```
        add (u, out, 2^phase) to send-;

leader_i:

   precondition:
        status = chosen


   effect:
        status := reported
```

For correctness we should prove two things, safety and liveness.

**Safety** Safety is always by induction on the steps. the goal is to show that no one other than the process with the maximal id reports leader. This happens if one out-token travels the full circle. For all processes different from $P_{i_{max}}$, this intuitively cannot happen since, at least, the maximal process will swallow it, and that is independant from the phase (therefore, the inductive variants need not mention the phase).

We can phrase the above observation

> "the out message of any non-maximal process never reaches beyond the maximal one"

more formally as:

$$j \in [i_{max}, i[$$

**Liveness**

# References

[1] Nancy Lynch. *Distributed Algorithms.* Kaufmann Publishers, 1996.