Assertional Verification of a Termination Detection Algorithm

A.A. Schoone, G. Tel

RUU–CS–88–6
March 1988

Rijksuniversiteit Utrecht

Vakgroep Informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Assertional Verification of a Termination Detection Algorithm

A.A. Schoone, G. Tel

Technical Report RUU-CS-88-6
March 1988

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands

Assertional Verification of a Termination Detection Algorithm

Anneke A. Schoone, Gerard Tel

Department of Computer Science, University of Utrecht,
P.O. Box 80.012, 3508 TA Utrecht, The Netherlands.

Abstract: We present a protocol skeleton for Termination Detection and verify its partial correctness by means of system-wide invariants (assertions). We also give two algorithms based on this skeleton, and prove their total correctness.

1 Introduction

The problem of Termination Detection is one of the most intensively studied problems in the field of distributed algorithms. Its practical interest asks for efficient solutions for different models of computation. The problem is easily explained and non-trivial, yet it allows elegant solutions. Thus the problem is suitable to compare the merits of design and verification techniques for distributed algorithms. Hence the problem is very interesting from a theoretical point of view, too. This paper is about "Assertional Verification" rather than about "A Termination Detection Algorithm".

The need for rigorous verification of Termination Detection algorithms is clearly illustrated by the recent publication of incorrect solutions [ARG86, TTL86, HZ87, TvL87]. Of course this applies for algorithms for other purposes as well. It is still common practice that the correctness of a distributed algorithm is motivated by reasoning about all (or some of the) executions of the algorithm and its environment. However, due to the inherent non-determinism of a distributed system (and, in the case of a Termination Detection algorithm, the unpredictability of the "basic computation"), the collection of all possible executions is large, and hard, if not impossible, to identify. In one case confusion about the model of computation led to an incorrect solution.

In the past we have advocated the use of assertional proofs for non-deterministic systems and given examples of assertional proofs for several communication protocols [SvL85, Sc87, Te87]. In these proofs the program as well as its environment are unambiguously described in

The work of the second author was supported by the Foundation for Computer Science (SION) of the Netherlands Organization for Scientific Research (NWO).

The authors UUCP addresses are *..!mcvax!ruuinf!anneke* and *..!mcvax!ruuinf!gerard.*

terms of a set of (atomic) actions. The partial correctness of this set of actions is expressed in terms of invariants. The correctness of each invariant is established by showing that (1) the invariant holds in the initial state of the system and (2) each atomic action preserves the invariant.

The set of atomic actions is not considered as the complete program, but rather as a program skeleton. As a next step in the design process, actions of the skeleton are scheduled in some manner as to satisfy total correctness also.

In this paper we apply this design and verification technique to a termination detection algorithm due to Mattern [Ma87]. Its verification in [Ma87] is rather informal and relies heavily on a graphical representation of an execution.

This paper is organized as follows. In the remainder of this section we introduce the model of computation, define the problem to be solved, and give the protocol skeleton. In section 2 we prove the partial correctness of the skeleton. In section 3 we give some complete protocols and prove their total correctness. In section 4 we compare our work with other work on assertional verification and on termination detection.

1.1 Termination Detection

In this paper we assume there is a set $\mathbb{P}$ of processes. Processes in $\mathbb{P}$ do not share memory, but communicate by exchanging messages only. Message delay is unpredictable and not bounded, but every message is guaranteed to arrive eventually, and unaltered. We do not assume that the links satisfy a FIFO (*First-In-First-Out*) discipline. The only assumption we make about the network topology is that the underlying graph is strongly connected. It is not essential for the correctness of the algorithms that the network is static, but for ease of presentation we assume $\mathbb{P}$ is fixed.

The processes in $\mathbb{P}$ cooperate to accomplish some task. For our purpose it is irrelevant what this task is and what algorithm is used for it. We assume that the operation of the processes is message driven. Only upon receipt of a message, belonging to the computation, a process performs some computation and sends zero or more messages. This basic action is atomic. It can be described by the following code (for a process p):

```
$\mathbf{B}_p$:  { A message $M$ arrives at $p$ }
     begin receive ($M$) ;
           compute ;
           forall $x \in X$ do send ($M_x$, $x$)
     end
```

Here *receive* (M) is a primitive to consume the message from the incoming message buffer.

The local computation is represented by the statement *compute*. X is a finite sequence of processes, generated during this computation. Finally a message M_x is sent to each process x in this sequence, using the send primitive *send* (M_x, x). In each invocation of $\mathbf{B}_p$ a different X can be generated. The process p may appear in the sequence, processes may appear more than once, or the sequence may be empty. It is assumed that during the execution of the system no messages are created or sent otherwise than during a **B**-action. The system as described here will be referred to as the *basic system* or *basic computation*, and messages belonging to it as *basic messages*.

To start the computation it is necessary that at least one basic message it created "spontaneously", i.e., otherwise than by being sent during the execution of some **B**-action. These messages are created by some environment, E, but we do not assume, as in [DS80], that this environment is itself a process. Because X in $\mathbf{B}_p$ can be empty, it is possible that at some moments there are no basic messages in transit in the system. We denote this situation by TERM.

TERM ≡ There are no basic messages underway.

When TERM holds, all basic actions are disabled, and thus no basic messages will be sent anymore. So TERM is a stable predicate: once it holds, it remains true forever. The purpose of a termination detection algorithm is to signify the processes eventually that TERM is true. For this purpose the code of a termination detection algorithm is superimposed on the code of the basic algorithm. The detection of TERM generally requires the exchange of extra messages, but these are of course not included in the definition of TERM.

For ease of presentation we assume the added code contains a special statement *detect*, which is to be executed when termination is detected. To be correct, the termination detection algorithm must satisfy the following three requirements:

(1) The correctness of the basic algorithm is not disrupted;

(2) (Partial correctness) If *detect* is executed TERM holds; and

(3) (Total correctness) If TERM holds, *detect* will eventually be executed.

1.2 Protocol Skeleton

In order to detect that there are no more messages in transit, processes must count messages they send and receive. A token traverses (part of) the network and collects the counts. We will show that termination can be decided on the value of the token only. The token Q is an array, containing one entry for each $p \in \mathbb{P}$. In the entry $Q[p]$ messages are counted whose destination is p. We assume that the environment that creates the messages also creates the token, such that initially

$Q[p]$ = the number of messages, created with destination p.

Each process p maintains a local count array L_p, also with one entry for each process in $\mathbf{P}$. Initially the value of these local arrays is $\vec{0}$. We add statements to $\mathbf{B}_p$ to update the local counters:

```
B_p:  { A message $M$ arrives at $p$ }
      begin receive ($M$) ; $L_p[p] := L_p[p] - 1$ ;
            compute ;
            forall $x \in X$ do
                  begin send ($M_x, x$) ; $L_p[x] := L_p[x]+1$ end
      end
```

Assuming that L is not a variable of the basic algorithm, adding these assignments to its code does not disrupt the correctness of the algorithm. Of course we can make this assumption without loss of generality, because variables in the basic algorithm (or in the superimposition described here) can always be renamed.

We introduce two new actions, one to update the token, and one to (eventually) decide termination.

```
T_p:  { Process $p$ holds the token $Q$ }
      begin $Q := Q + L_p$ ;
            $L_p := \vec{0}$
      end
```

```
D_p:  { Process $p$ holds the token $Q$ }
      begin if $Q \le \vec{0}$ then detect end
```

Here :=, +, and ≤ work on vectors in a componentwise manner. Because these actions do not operate on the state space of the basic algorithm they again do not disrupt its correctness. According to action $\mathbf{D}_p$ *detect* can be executed when all components of Q are non-negative. It is the subject of section 2 to show that

$$\neg\text{TERM} \Rightarrow \exists p : Q[p] > 0.$$

so that *detect* will be executed only when TERM holds. The token can be passed from one process to another. We do not yet specify how movement of the token is determined, because this is not essential for the partial correctness of the skeleton. It should be noted however, that a move of the token from one process to another does not alter its value or the value of another variable introduced so far.

2 Proof of Partial Correctness

To prove the correctness of the skeleton in section 1.2 we introduce a more complex skeleton. In this new skeleton messages are counted not only separatedly for each destination, but also for each source. Thus the token Q is now a two dimensional matrix with an entry $Q[p,q]$for each $p,q \in \mathbb{P}$. In this entry messages are counted that are sent from p to q. We will show that TERM holds if the sum over each column in this matrix is non-positive. In the new skeleton two matrices, S and R, take the place of the local arrays L, to register sent and received messages locally. Of these matrices, the entries $S[p,q]$ and $R[p,q]$ are local to p, and q, respectively.

In the proof we make use of auxiliary variables f_p, O, and N. The ghost variable N is matrix valued, and $N[p,q]$ denotes at any instant the actual number of massages that is currently in transit from p to q. Thus, the sending by p of a message to q implicitly has the same effect as $N[p,q] := N[p,q]+1$. The receipt of this message has the effect $N[p,q] := N[p,q]-1$, and this can happen only if $N[p,q] > 0$.

We maintain a sequence or total ordering O of "badges" b_p and t_p. Thus, the "type" of O is a permutation of $2 \bullet |\mathbb{P}|$ elements. We need this variable to be able to say something about what has happened in the past. Whenever process p updates the token, t_p is replaced to the end of the sequence. The same happens with b_p when $\mathbf{B}_p$ is executed for the first time or the first time after a token update in p. On this occasion f_p takes the value of the sender of the message that is received by p. For badges x,y, by $x<y$ we mean that x is further away from the end of the queue than y.

The new protocol skeleton, with updates of ghost variables, now looks like this:

```
$\mathbf{B}_p$:  { A message $M$, sent by $q$, arrives at $q$ }
      { i.e., $N[q,p] > 0$ }
      begin $receive(M)$ ; (* i.e., $N[q,p] := N[q,p]-1$ *)
            $R[q,p] := R[q,p]+1$ ;
            if $f_p$ = $nil$ then
                  begin move $b_p$ to the end of $O$ ; $f_p := q$ end ;
            $compute$ ;
            forall $x \in X$ do
                  begin $send$ $(M_x, x)$ ; (* i.e., $N[p,x] := N[p,x]+1$ *)
                        $S[p,x] := S[p,x]+1$ end
      end
```

```
$T_p$   { Process $p$ holds the token }
        begin $f_p$ := nil ; move $t_p$ to the end of $O$ ;
            forall $q \in P$ do
                begin $Q[p,q] := Q[p,q]+S[p,q]$ ; $S[p,q] := 0$ ;
                    $Q[q,p] := Q[q,p]-R[q,p]$ ; $R[q,p] := 0$
                end
        end
```

```
$D_p$:  { Process $p$ holds the token }
        begin if $\forall r,s : Q[r,s] \le 0$
            then detect
        end
```

We assume that initially for all p, q $Q[p,q] = N[p,q] \ge 0$, $S[p,q] = 0$, $R[p,q] = 0$, $f_p = nil$, O is such that $b_p < t_q$. Note that we can now express TERM as $\forall p,q : N[p,q] = 0$.

We formulate and prove a series of invariants of the system. The first lemma gives some elementary relations between message counts. Note that by definition of $N[p,q]$, $N[p,q] \ge 0$ always.

Lemma 2.1: For all p and q the following holds invariantly:

(i) $R[p,q] \ge 0$,

(ii) $S[p,q] \ge 0$, and

(iii) $Q[p,q]+S[p,q] = N[p,q]+R[p,q]$.

Proof: (i) Initially $R[p,q] = 0$ so (i) holds. A $\mathbf{B}_q$ action does not decrease R and hence leaves (i) true. A $\mathbf{T}_q$ action resets $R[p,q]$ to 0 and hence makes (i) true. Other actions do not change $R[p,q]$.

(ii) Initially $S[p,q] = 0$ so (ii) holds. A $\mathbf{B}_p$ action does not decrease $S[p,q]$ and hence leaves (ii) true. A $\mathbf{T}_p$ action resets $S[p,q]$ to 0 and hence makes (ii) true. Other actions do not change $S[p,q]$.

(iii) Initially $Q[p,q] = N[p,q]$ and $R[p,q] = S[p,q] = 0$, so (iii) holds. Upon sending a message from p to q (an action that implicitly increments $N[p,q]$), $S[p,q]$ is incremented so that (iii) is maintained. Upon receiving this message $R[p,q]$ is decremented so that (iii) is maintained. In action $\mathbf{T}_p$ $Q[p,q]$ is increased, and $S[p,q]$ is decreased by the same amount (possibly 0), so that (iii) is maintained. In $\mathbf{T}_q$ $Q[p,q]$ and $R[p,q]$ are decreased by the same amount so that (iii) is maintained. □

Informally speaking, we say that a message is registered (viz., in the token) when the information about its sending has been copied into the token, but information about its receipt has not. We must prove that if there are no registered messages ($Q \leq \vec{0}$), then there are no unregistered messages also (TERM). The following lemma says something about the value of the auxiliary variables in case an unregistered message from p to q exists.

Lemma 2.2: For all p and q the following holds invariantly:

(i) $S[p,q] > 0 \Rightarrow f_p \neq nil$,

(ii) $f_p \neq nil \Leftrightarrow t_p < b_p$,

(iii) $f_p \neq nil \Rightarrow R[f_p,p] > 0$.

Proof: (i) Initially $S[p,q] = 0$ so (i) holds. After a $\mathbf{T}_p$ action $S[p,q] = 0$ so (i) holds. After a $\mathbf{B}_p$ action $f_p \neq nil$ so (i) holds. Other actions do not change $S[p,q]$ or f_p.
(ii) Initially $f_p = nil$ and $b_p < t_p$ so (ii) holds. After a $\mathbf{T}_p$ action $f_p = nil$ and $b_p < t_p$ so (ii) holds. Consider $\mathbf{B}_p$. If $f_p \neq nil$ prior to it, $t_p < b_p$ already by (ii), and neither f_p nor O change in the execution of $\mathbf{B}_p$, so (ii) remains true. If $f_p = nil$ prior to it, f_p is now set to some $q \neq nil$, and b_p is moved to the end of the queue so $t_p < b_p$ becomes true. Again (ii) holds. Other actions do not change f_p or the relative ordering of t_p and b_p.
(iii) Initially $f_p = nil$ so (iii) holds. After $\mathbf{T}_p$ $f_p = nil$ so (iii) holds. Consider $\mathbf{B}_p$. If $f_p \neq nil$ prior to it, f_p does not change and $R[f_p,p]$ does not decrease, so (iii) remains true. If $f_p = nil$ prior to it, f_p is now set to some $q \neq nil$ and $R[q,p]$ is incremented and hence becomes positive, so (iii) holds after it. Other actions do not change f_p or $R[f_p,p]$. □

The third lemma is particularly important. If a process has performed a basic action since its recentmost token update, then either an entry of the token is positive, or there is yet another process that has performed a basic action since its recentmost token update; moreover, in the latter case this process is smaller in the ordering of b-badges. This implies of course that there can be no such process if all fields of the token are non-positive.

Lemma 2.3: $f_p = nil \ \vee \ Q[f_p,p] > 0 \ \vee \ (S[f_p,p] > 0 \wedge t_{f_p} < b_{f_p} < b_p)$ holds invariantly.

Proof: Initially $f_p = nil$.
After a $\mathbf{T}_p$ action $f_p = nil$ so the statement holds.
Consider $\mathbf{B}_p$. If $f_p \neq nil$ prior to its execution, the second or third disjunct holds. Neither of these is violated by $\mathbf{B}_p$, so the invariant remains true. If $f_p = nil$ prior to its execution, f_p is now set to q, the sender of the message, and $R[q,p]$ is set to a positive value. By lemma 2.1.iii $Q[q,p]+S[q,p] > 0$ follows, so we have $Q[q,p] > 0$ or $S[q,p] > 0$. $S[q,p] > 0$ implies $t_q < b_q$ by lemma 2.2.ii, and $q \neq p$ by 2.2.i, so after appending b_p to the end of O we have $t_q < b_q < b_p$.
If $f_p \neq nil$ we must consider actions $\mathbf{T}_{f_p}$ and $\mathbf{B}_{f_p}$ also.
After $\mathbf{T}_{f_p}$, $S[f_p,p] = 0$ and hence, by 2.2.iii and 2.1.iii, $Q[f_p,p] > 0$.

Consider $\mathbf{B}_{f_p}$. It does not violate the second disjunct because it does not change Q. It does not violate the second disjunct because $\mathbf{B}_{f_p}$ does not decrease $S[f_p,p]$, and if $t_{f_p} < b_{f_p}$ then (by 2.2.ii) $f_{f_p} \neq nil$, and $\mathbf{B}_{f_p}$ does not change O.
All other actions do not change f_p, $Q[f_p,p]$, $S[f_p,p]$, or the relative ordering of t_{f_p}, b_{f_p}, and b_p. □

Lemma 2.4 is a technical ingredient for theorem 2.6. This theorem will allow us to transform the skeleton in this section to the skeleton in section 1.2.

Lemma 2.4: For all p, q, $Q[p,q] \geq 0 \vee (t_p < b_p < t_q)$ holds invariantly.
Proof: Initially $Q[p,q] \geq 0$.
After $\mathbf{T}_p$ $S[p,q] = 0$ so again $Q[p,q] \geq 0$ follows.
Consider $\mathbf{B}_p$. $\mathbf{B}_p$ does not violate the first disjunct because it does not change Q. It does not violate the second disjunct because if $t_p < b_p$, $f_p \neq nil$, so $\mathbf{B}_p$ does not change O at all.
Consider $\mathbf{T}_q$ $(q \neq p)$. After $\mathbf{T}_q$ we have $t_p < t_q$, $b_p < t_q$, and $R[p,q] = 0$. By 2.1.iii, $Q[p,q] \geq 0 \vee S[p,q] > 0$ follows, and $S[p,q] > 0$ implies $t_p < b_p$.
Other actions do not change $Q[p,q]$ or the relative ordering of t_p, b_p, and t_q. □

The next theorem says that the *detect* statement is not executed if there is no termination, thus the partial correctness of the skeleton in this section in established.

Theorem 2.5: $\neg$TERM $\Rightarrow \exists\, p,q : Q[p,q] > 0$.
Proof: Suppose $\neg$ TERM, i.e., there is a message underway from (say) r to s. $N[r,s] > 0$ implies (by lemma 2.1) that $Q[r,s] > 0$ or $S[r,s] > 0$. In the first case we are ready (for p,q take r,s). In the second case, let q be the process with $\exists p : S[q,p] > 0$ and minimal b_q. By lemma 2.2.i, $f_q \neq nil$, so by lemma 2.3, $Q[f_q,q] > 0$ or $S[f_q,q] > 0 \wedge b_{f_q} < b_q$. The latter contradicts with the choice of q, so $Q[f_q,q] > 0$ follows. □

The skeleton in section 1.2 used an array token instead of a matrix valued token. The q^{th} entry of this array represents $\sum_p Q[p,q]$. The next theorem states that this array can be used for detection purposes as well.

Theorem 2.6: $\exists\, u,v : Q[u,v] > 0$ implies $\exists\, q : \sum_p Q[p,q] > 0$.

Proof: Assume $\exists\, u,v : Q[u,v] > 0$, now let q be the process with $\exists\, p : Q[p,q] > 0$ and minimal t_q. We will prove $\forall p : Q[p,q] \geq 0$, which together with the premise implies $\sum_p Q[p,q] > 0$.
By choice of q, we have that for all p either $t_q \leq t_p$ or $(\forall s : Q[s,p] \leq 0 \wedge t_p < t_q)$. $t_q \leq t_p$

implies $Q[p,q] \geq 0$ by lemma 2.4.
$\forall s: Q[s,p] \leq 0$ implies $f_p = nil$ or $S[f_p,p] > 0 \wedge t_{f_p} < b_{f_p} < t_p$ by lemma 2.3. Assume (†) the latter, let r be the process with $\exists s: S[r,s] > 0$ and minimal b_r. Note that $t_r < b_r \leq b_{f_p} < t_p$ so $b_r < b_p$, and $t_r < t_q$. Since $f_r \neq nil$, we have by 2.3 that $Q[f_r,r] > 0$ or $S[f_r,r] > 0 \wedge t_{f_r} < b_{f_r} < b_r$. The former contradicts with the choice of q, the latter with the choice of r. Thus (†) leads to a contradiction, $f_p = nil$ follows and hence (lemma 2.2.i) $S[p,q] = 0$ and (by lemma 2.1) $Q[p,q] \geq 0$. □

We will now transform the skeleton in this section to the skeleton in section 1.2. First, remove all assignments to the auxiliary variables f and O. Further, by theorem 2.6 we can weaken the condition in the **D**-action to $\forall q: \sum_p Q[p,q] \leq 0$. But then we can replace each column of Q by its sum, i.e., replace the matrix by a one-dimensional array, where $Q[q]$ represents $\sum_p Q[p,q]$. In each process we can replace the variables S and R by one local array L, such that $L_p[q]$ represents the update p will make to $Q[q]$, i.e.,

$$L_p[p] = S[p,p] - \sum_q R[q,p]$$

$$L_p[q] = S[p,q] \qquad \text{if } p \neq q.$$

Now the actions of our skeleton transform to

```
B_p: { A message $M$ arrives at $p$ }
    begin receive $(M)$ ; $L_p[p] := L_p[p] - 1$ ;
        compute ;
        forall $x \in X$ do
            begin send $(M_x, x)$ ; $L_p[x] := L_p[x] + 1$ end
    end

T_p: { Process $p$ holds the token $Q$ }
    begin $Q := Q + L_p$ ;
        $L_p := \vec{0}$
    end

D_p: { Process $p$ holds the token $Q$ }
    begin if $Q \leq \vec{0}$ then detect end
```

In this section we only considered messages that are sent from one process to another. To handle the initialization messages of the system, we can conceptually treat the environment as a process E. Thus, initially, for the matrix token, $Q[p,q] = 0$ for all $p \neq E$, and $Q[E,q]$

equals the number of messages, created by the environment with destination q. For the array token this means that initially $Q[q]$ equals the number of messages, created by the environment with destination q. Because no messages are ever sent to the environment, we have $L_p[E] = 0$ for all p and $Q[E] = 0$ always. Thus these entries do not have to be implemented.

3 Applications

In this section we will give some complete algorithms for termination detection, based upon the skeleton of section 1.2. The partial correctness of this skeleton was proved in section 2. We will prove the total correctness of the algorithms also. We use the following theorem:

Theorem 3.1: TERM $\wedge$ $(\forall p : L_p = \vec{0})$ implies $Q = \vec{0}$.

Proof: We refer here to the notations of lemma 2.1. TERM means that for all q, $\sum_p N[p,q] = 0$, and $\forall p : L_p = \vec{0}$ means that for all q, $\sum_p S[p,q] + \sum_p R[p,q] = 0$. It follows that for all q, $\sum_p Q[p,q] = 0$. □

Furthermore, suppose TERM holds and p updates the token. After the update L_p is set to $\vec{0}$, and, since all basic actions are disabled, L_p will remain $\vec{0}$ forever. Thus, we only need to make sure that after TERM has become true, every p with $L_p \neq \vec{0}$ will do a token update, and subsequently a $\mathbf{D}_p$ action will be done.

3.1 A Ring of Processes

Assume the processes know a logical ring in the network. That is, each processor knows of a neighbor process S_p, the *successor* of p, such that a path, starting at some process l and stepping from each process to its successor, passes through every process and returns to l. If the token circulates around this ring, it will visit every process in $|\mathbb{P}|$ steps. Thus, if every process updates the token, after at most $|\mathbb{P}|$ steps after TERM has become true we have $Q = \vec{0}$. This leads to the following termination detection algorithm:

```
when $p$ receives the token $Q$ do
    begin $Q := Q + L_p$ ; $L_p := \vec{0}$ ;
        if $Q \leq \vec{0}$ then detect
              else send $(Q, S_p)$
    end
```

It is allowed in the skeleton that some processes are visited by the token more often than others. Therefore this algorithm is easily generalized for use in any network in combination with a (serial) traversal scheme for this network. If the token traverses the network in K steps, termination will always be detected at most K steps after its occurrence. A collection of serial traversal algorithms for several classes of networks can be found in [KKM87].

3.2 Complete Networks

In the algorithm in the previous section the route of the token is fixed. It is possible to adapt the route of the token dynamically to the execution of the basic algorithm. The idea is to send the token to a process q such that $Q[q] > 0$. (If no such process exists, termination is concluded.)

```
when $p$ receives the token $Q$ do
    begin $Q := Q + L_p$ ; $L_p := \vec{0}$ ;
        if $Q \leq \vec{0}$ then detect
              else begin take $q$ s.t. $Q[q] > 0$ ;
                      send $(Q, q)$
                 end
    end
```

Lemma 3.2: For $q \neq p$, $L_p[q] \geq 0$ holds invariantly.

Proof: For $q \neq p$, $L_p[q] = S[p, q]$, use lemma 2.1. □

Lemma 3.3: TERM implies $Q[q] + L_q[q] \leq 0$.

Proof: TERM implies $Q[q] + \sum_p L_p[q] = 0$, with 3.2 $Q[q] + L_q[q] \leq 0$ follows. □

Theorem 3.4: In the algorithm above, after TERM has become true, the token is sent only to a process q if $L_q \neq \vec{0}$.

Proof: From $Q[q] > 0$ and lemma 3.3 follows $L_q[q] < 0$. □

Thus, after TERM has become true, processes can be visited at most once, and exactly those processes are visited whose L was non-zero at the time TERM became true. Note that we can view the network as "logically complete" if a routing mechanism allows any process p to sent messages to any other process q, even if no direct link pq exists. An interesting property of this algorithm is, that only those processes take part in it, that took part in the basic algorithm also. The algorithm of Mattern [Ma87] does not have this property. As far as we know the Dijkstra and Scholten algorithm [DS80] is the only published algorithm so far that has this property. It makes the algorithm particularly efficient for "small" computations on "large" networks, i.e., computations that use only a few processes of the network. For these applications Q can be coded in such a way that only its non-zero elements are transmitted. Extension to dynamic networks now becomes trivial.

3.3 An Optimization

To prove total correctness, it suffices to show that the algorithm behaves well *after* TERM has become true. From the viewpoint of efficiency however, it is desirable to minimize operations of the token *before* TERM has become true. We add statements of the form **wait until** $Q[q]+L_q[q] \leq 0$ to the text of the termination detection algorithm. From lemma 3.3 we know that this change in the algorithm does not affect the operation of the algorithm when TERM holds. Thus, the termination detection algorithm can be suspended only when there is no termination. During the wait, basic actions can be performed. When TERM becomes true operation of the termination detection will resume. This argument applies irrespective of where in the code the statement is added. It is most efficient to do it before the token update. For example the program in section 3.2 becomes

```
when $p$ receives the token $Q$ do
    begin wait until $Q[p]+L_p[p] \leq 0$ ;
        $Q := Q+L_p$ ; $L_p := \vec{0}$ ;
        if $Q \leq \vec{0}$ then detect
                else begin take $q$ s.t. $Q[q] > 0$ ;
                        send $(Q, q)$
                    end
    end
```

4 Conclusions

4.1 Comparison with related work on assertional verification

We gave a protocol skeleton (for termination detection) and a proof of its total correctness. The protocol skeleton consists of a set of atomic actions. A rather similar approach to programs and their verification was recently proposed by Shankar and Lam [SL87] and Chandy and Misra [CM88].

The three methods differ in the way atomic actions are expressed. In the work of Shankar and Lam an atomic action (event) is not expressed algorithmically, but in terms of its *input-output-predicate*. This is a boolean formula in the variables before and after the occurrence of the event, which is supposed to be satisfied whenever the event occurs. In the work of Chandy and Misra an action is always a multiple conditional assignment. We chose to express actions algorithmically to make the gap between the skeleton and its actual implementation as small as possible. This choice also allows for an easier and more natural way of modeling real-time and channel errors, see [Te87]. We feel, however, that this difference is not fundamental: the same theory, inference rules, etc., applies in all cases.

A fundamental difference lies in the attitude towards partial versus total correctness. Both [SL87] and [CM88] identify the set of actions with the final program. This set must satisfy total correctness criteria under some reasonable fairness assumption. We take a radically different point of view here. In our view the set of atomic actions is no more than a skeleton on which a complete program can be based. Only this complete program should satisfy total correctness. The advantage of this approach over the others is threefold:

(1) Our design technique separates concerns about partial correctness from concerns about total correctness. Partial correctness is proven for one object (viz., the skeleton), and total correctness for another object (viz., the complete program). It should of course be noted that the relation between the two objects is such that essential properties of the first are shared by the second. In both other approaches it is necessary to find one object, satisfying both requirements. This is often a difficult task.

(2) Based upon one skeleton, several complete programs can be constructed, with varying properties. This is illustrated, for example, in this paper, [SvL85], and [Sc87]. Thus, the proof of partial correctness is "shared" by different programs.

(3) A skeleton can be designed with a lot of internal non-determinism or with parameters in it. The execution of the skeleton is always non-deterministic, because any scheduling of atomic actions is admissible. In a later design stage this non-determinism can be resolved, certain sequences of actions can be prescribed, and parameters can be specified. Thus, the algorithm can be tuned to high efficiency or to satisfy some desirable extra properties. See section 3 and for example [SvL85] and [Te87].

4.2 Comparison with Work on Termination Detection

This research was started as an attempt to verify the vector-counter termination detection algorithm of Mattern [Ma87] by means of invariants. It was felt that the (partial) correctness of the algorithm relied in no way on the circular arrangement of the processes, or on the fact that every process should be visited by the token at least once. Note that our correctness argument is independent of the circular arrangements of processes, and that in our algorithm it is not necessary that every process is visited at least once.

It is interesting to compare our work with the algorithm of Plouzeau [Pl87]. This algorithm is matrix based and sent and received messages are counted separately, as in section 2. Also in the "token" they are counted separately, so that the test $Q \leq 0$ is replaced by $Total\_S \leq Total\_R$. A more interesting difference is, that the "token" does not move, but the value of the local counters is transmitted repeatedly to the place where it resides. Updates of one process are guaranteed to be received in correct order.

The algorithm of Hélary *et al.* [He87] uses a similar approach. Here, however, the local counters are not reset after sending an update. New values, received by the token, overwrite old values rather than being added to it. Thus the algorithm is easily adapted to be resilient against loss, duplication, or resequencing of messages. The algorithm can detect deadlock also. Two things are important to note.

(1) The algorithms differ in the way information is collected, but in all algorithms the same information is collected and it is used in the same way. The differences result in different robustness against process or communication failure. If we abstract away these differences, the three algorithms are the same. With some changes our correctness proof applies to the other algorithms as well.

(2) The use of alternative ways of information gathering to achieve fault tolerance is not limited to this particular application (termination detection). It can be used to make other token based algorithms fault tolerant as well.

4.3 Models for the Distributed Termination Detection Problem

The problem of termination detection has been studied under several models of computation for the basic algorithms. In this paper we used the *atomic* model. We now discuss the *transactional* and the *synchronous* model also. The transactional model is the most general one. It is assumed that processes can be either *active* or *passive*, that only active processes send messages, and that a state change from passive to active takes place only upon receipt of a message. This informal description is made precise in the following three basic actions.

```
$S_p$:  { $state_p = active$ }
        begin $send\ (M)$ end

$R_p$:  { A message $M$ arrives at $p$ }
        begin $receive\ (M)$ ; $state_p := active$ end

$I_p$:  begin $state_p := passive$ end
```

In the transactional model termination is characterized by

$$\text{TERM}_{\text{trans}} \equiv \text{There are no messages underway and } \forall p : state_p = passive .$$

When $\text{TERM}_{\text{trans}}$ holds, all S_p and R_p actions are disabled and I_p actions have no effect. Hence $\text{TERM}_{\text{trans}}$ is stable.

In the synchronous model it is assumed that communication is synchronous, i.e., a send action and the corresponding receive action are considered as one communication action. Thus the actions in the system above are replaced by

```
$C_{pq}$:  { $state_p = active$ }
           begin $state_q := active$ end

$I_p$:  begin $state_p := passive$ end
```

Because the new action C_{pq} is a combination of actions from the transactional model (viz., S_p and R_q), $\text{TERM}_{\text{trans}}$ is stable in the synchronous model also. In the synchronous model there are never messages in transit, so $\text{TERM}_{\text{trans}}$ reduces to

$$\text{TERM}_{\text{synch}} \equiv \forall p : state_p = passive .$$

In the atomic model it is assumed that send actions and internal computation are triggered by receive actions only. After receiving a message, a process sends zero or more messages, and becomes passive. The three actions are replaced by

```
$B_p$:  { A message $M$ arrives at $p$ }
        begin $receive\ (M)$ ;
            $compute$ ;
            forall $x \in X$ do $send\ (M_x, x)$
        end
```

Again, because the new action is a combination of actions from the transactional model, $\text{TERM}_{\text{trans}}$ is stable in this model also. In the atomic model $state_p = passive$ always (outside the actions), so this variable is not implemented and $TERM_{trans}$ reduces to

$$TERM_{atom} \equiv \text{There are no messages underway.}$$

We can say that the synchronous model ignores communication, the atomic model ignores computation. It is felt that the essential difficulties of the problem of termination detection lie in communication. Indeed, solutions for the synchronous model are simpler than solutions for the transactional model, see Tel [Te86]. On the other hand, ignoring computation does not hide essential difficulties of the problem, because the state of a process is always locally visible. As Mattern [Ma87] argues, in most existing termination detection algorithms for the transactional model it is assumed that the detection algorithm is suspended in active processes. Thus the detection algorithm treats the computation as atomic anyhow. In fact any solution for the atomic model is easily adapted to the transactional model by suspending the detection algorithm in active processes. In this paper we adopted the atomic model because of its simplicity.

5 References

[ARG86] Arora, R.K., S.P. Rana, M.N. Gupta, *A distributed termination detection algorithm for distributed computations*, Inf. Proc. Lett. 22 (1986) 311-314.

[CM88] Chandy, K.M., J. Misra, *A foundation of parallel program design*, Addison Wesley, 1988.

[DS80] Dijkstra, E.W., C.S. Scholten, *Termination detection for diffusing computations*, Inf. Proc. Lett. 4 (1980) 1-4.

[He87] Hélary, J.-M., C. Jard, N. Plouzeau, M. Raynal, *Detection of stable properties in distributed systems*, Proceedings 6th PoDC, Vancouver, Canada, 1987.

[HZ87] Hazari, C., H. Zedan, *A distributed algorithm for termination detection*, Inf. Proc. Lett. 24 (1987) 293-297.

[KKM87] Korach, E., S. Kutten, S. Moran, *A modular technique for the design of efficient distributed leader finding algorithms*, Technical Report, Department of Computer Science, The Technion, Haifa, Israel, 1987.

[Ma87] Mattern, F., *Algorithms for distributed termination detection*, Distributed Computing 2 (1987) 161-175.

[Pl87] Plouzeau, N., *Détection de la terminaison: un algorithme fondé sur une approche observationelle*, Rapport No 610, INRIA, Rocquencourt, France, 1987.

[Sc87] Schoone, A.A., *Verification of connection management protocols*, Techn. Rep.