Assertional Verification of a Reset Algorithm

Nicolien J. Drost and Anneke A. Schoone

RUU-CS-88-5
February 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

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

Abstract

This paper gives a formal correctness proof using system-wide invariants for a reset procedure presented by Afek et al. [AAG87]. A reset procedure serves to adapt an algorithm designed for fixed topology networks to networks where links may fail and come up again. As far as we know this is the first time an algorithm of this type is verified using this technique. So this paper also extends the method of assertional verification to a wider class of distributed algorithms and adds to a better understanding of advantages and disadvantages of this method.

Keywords: reset algorithm, assertional proof, distributed algorithm, fault-tolerance.

Contents

Chapter 1

Introduction

Many existing distributed algorithms are static: they only operate correctly under the assumption that the network topology remains fixed during the execution of the algorithm. A change of the network topology while the algorithm is executing could cause meaningless or even wrong results. This problem could be circumvented by resetting the algorithm to an initial or checkpoint state every time a change in the network is detected, followed by a restart. Note that it is essential that changes in network topology are detectable.

Messages of old versions of the algorithm should not be received by a new version of the algorithm. This problem can be solved by adding a version number [Fin79] or a timestamp [Lam78] to each message of the algorithm. This method has two drawbacks: it adds extra information to all messages of the algorithm, and there is no bound on the size of the sequence numbers. An advantage of this approach is that it works even if message delivery on links is not FIFO. To remove the second drawback, Finn [Fin79] also gave an algorithm that passes only differences in version numbers over a link. However, Soloway and Humblett [SH86] gave a counterexample to this algorithm.

Afek et al. [AAG87] recently published a reset procedure that does not use sequence numbers. They assume that the sequence of topological changes in the network is finite, and that in finite time after the last topological change every node is aware of the state of its incident links. Networks satisfying these two assumptions are called **dynamic**. They call the distributed algorithm that is being reset the π-algorithm . Their reset procedure only resets the subnetwork which was recently traversed by messages of the π-algorithm. The message complexity of the reset procedure, per topological change, is $O(|E_\pi|)$, where E_π is the set of links that is traversed by messages of π. The quiescence time (time from the last topological change until termination of the reset procedure) is $O(n_\pi)$, where n_π is the number of nodes to which the links in E_π are attached.

Afek et al. [AAG87] give a sketch proof of their reset procedure. In this article we give a formal proof of the correctness of this reset procedure, using system-wide invariants, following [Kro78] and [Knu81]. The proof consists of two parts:

1. Termination: The reset procedure terminates in finite time after the last topological change.

2. Correctness: After termination of the reset procedure the system is in a correct state.

But what is a correct state? Afek et al. [AAG87] state that a distributed algorithm is **fail-safe** if the algorithm terminates correctly when run in a dynamic network, i.e.

the result is consistent with the final topology of the network. They claim that their reset procedure , added to a static π-algorithm, transforms the π-algorithm to a fail-safe algorithm. But this cannot be proved if the precise form of the π-algorithm is unknown. So we prove that after termination of the reset procedure no node executes an incorrect version of the π-algorithm any more and no more messages from incorrect versions of the π-algorithm are still present.

Afek et al. [AAG87] call an incorrect version of the π-algorithm 'obsolete'. They state:

> Basically, if a node realizes that it ran with obsolete input, its computation and the computation of all nodes that have received a message from it are obsolete. Moreover, the computation of all nodes it received a message from are obsolete as well, since part of the 'effects' they were supposed to produce are lost in an obsolete computation, thus they are obsolete too. Thus, each computation that received a message from an obsolete computation or each computation whose message was received in an obsolete computation is also obsolete.

We define 'obsolete' more precisely, and prove that after termination of the reset procedure no more nodes executing an obsolete computation and no more obsolete messages are present.

The paper is organized as follows: In chapter 2 we give the model and reset procedure as given in [AAG87], in chapter 3 we give an event-oriented description of the behavior of nodes and links, and add an external bookkeeper, thus reformulating the procedure to a form suitable for the proof. In chapter 4 we give the proof , and chapter 5 contains the conclusions.

Chapter 2

The Reset procedure

2.1 Assumptions

Afek et al.[AAG87] model a dynamic network by incorporating a few variations into the standard model of static fault-free asynchronous networks. They give a number of assumptions, and four axioms (A1 to A4) which capture the effects of having an underlying handshake protocol between two adjacent nodes. Their assumptions are the following:
The network consists of nodes, and links between nodes. Nodes communicate only by exchanging messages over links. Links are bidirectional, and FIFO.
The axioms they use are:

A1) Messages at one end are received in the order they were sent at the other end.

If a message arrives at the other end of a link, it arrives error-free.

Links may go down or become operating again. If a link changes its state, the nodes at each end of the link are notified of this change in finite time:

A2) At each end point of a link, the link protocol generates an alternating sequence of 'WAKE' and 'FAIL' signals.

If a node does not yet know that a link is down and tries to send a message over this link, the message is lost.

The time intervals at a node between a 'WAKE' and the subsequent 'FAIL' from a link are called the **operating intervals** for that link at the node. An interval on one side of a link is a **mate** of an operating interval on the other side if a message sent during one interval was received during the other, or vice versa.

A3) An interval can have at most one mate. Moreover, for any two distinct intervals at the same side that do have mates, the mate of the earlier interval precedes (at the other side) the mate of the later interval.

A4) If an infinite operating interval has a mate, then this mate must be an infinite interval as well. Moreover, every message sent during an infinite interval arrives successfully at the other side.

The π-algorithm consists of a set of programs, one at each node of the network. The π-algorithm is started by an arbitrary set of nodes which are awakened by external signals at arbitrary times and begin running their programs. Other nodes may be awakened

by the reception of a π-message. Reception of a π-message triggers a local computation and possibly the transmission of new messages. Local computation is assumed to be instantaneous.

2.2 The Reset procedure

The reset procedure removes an obsolete execution of the π-algorithm, resetting all participating nodes to their initial states, and purging all obsolete π-messages. Initially all nodes are in Ready mode. Some nodes may start executing the π-algorithm. If a node executing the π-algorithm discovers a change in the status of an incident link, the node stops the execution of the π-algorithm, goes into Aborting mode and sends Abort messages over all links currently in use for the π-algorithm. A node in Ready mode receiving an Abort message over link L marks this link as Parent link, goes into Aborting mode and sends Abort messages over all links used by the π-algorithm. In this way a tree of Aborting nodes and Parent links is formed. If branches of this tree were still growing while nodes at the top already returned to Ready mode a nonterminating execution could result: the nodes returned to Ready mode could again be used by the same obsolete execution of the π-algorithm and again be Aborted et cetera. Therefore Aborting nodes are only allowed to return to mode Ready when the tree of Aborting nodes does not grow any more. An Aborting node waits until it has received an acknowledgement from each link over which it sent an Abort message. An Aborting node receiving an Abort message sends back an Ack message over the same link. When an Aborting node receives its last pending Ack, it sends an Ack to its parent. This causes a flow of Acks from the leaves of the tree to the root.

When an Aborting node having no parent receives its last pending Ack, it knows that the tree is no longer expanding. So it resets the variables of the π-algorithm to their initial states, sends Release messages over all used links and goes back to Ready mode. An Aborting node receiving a Release message over its Parent link also resets the variables of the π-algorithm, sends Release messages, and goes back to Ready mode. So the Release messages spread through the tree from the root to the leaves.

Topological changes may also happen during the execution of the reset-procedure. If an Aborting node discovers a link recovery, it records that link as being up. No further action is needed, because this link was not used for the execution of the π-algorithm. If an Aborting node discovers that a link has gone down, it records the link as being down. No Ack can arrive any more from this link, so the node acts as if a pending Ack was received. If this link is the parent link of this node, the node now assumes it has no parent.

Ready nodes executing the π-algorithm record which links they use during the execution of the π-algorithm. If an Aborting node receives a π-message, it stores the π-message in a special queue called Q. When the node returns to mode Ready these π-messages are forwarded to the π-algorithm. If an Aborting node receives an Abort over a link, all π-messages in the queue received from this link are obsolete, and are therefore removed from the queue.

The reset algorithm consists of two modules: The reset Procedure and the Manager process. The Manager Process receives messages and passes them to the reset Procedure or the π-algorithm or places them into the queue Q. It also resets the variables of the π-algorithm to their initial state.

In Figure 2.1 the code for the reset procedure and the Manager Process is given as specified by Afek et al. [AAG87].

```
Messages: ABORT, ACK, RELEASE;
Node Variables:
State ∈ Aborting, Ready
Used: set of links used by π
AckPend(e) : Boolean ∀e ∈ Used, init. false;
Parent: link from child to parent node;

For Down, Up on link L or any input change:
  if Down on link L
  then Used := Used - L;
  if State = Ready
  then Parent := nil;
       Call Procedure Broadcast-Abort;
  if Down on link L
  then if Parent = L
       then Parent := nil;
       if State-Aborting
            and (AckPend(L) | Parent=nil)
       then AckPend(l) := false;
            Call Converge-Abort;

Procedure Broadcast-Abort
  State := Aborting;
  for all e ∈ Used do :
       AckPend(e) := true;
       send ABORT over e;
  if Used = ∅
  then Call Procedure Release;

For ABORT message on link L
  if State = Ready
  then Parent := L;
       Call Broadcast-Abort;
  else send ACK on L;

Procedure Converge-Abort
  if for all e ∈ Used not AckPend(e)
  then if Parent ≠nil
       then send ACK on Parent;
       else Call Procedure Release;

For ACK message on link L
  AckPend(L) := false;
  Call Procedure Converge-Abort;

Procedure Release
  for all e ∈ Used send RELEASE over e;
  State := Ready /*trigger the Manager*/

For RELEASE message arriving on link L
  if State=Aborting & L = Parent
  then Call Procedure Release;
```

The Fail-Safe Reset Procedure

```
/* Variables */
initiator : boolean; /*Initially false, tells
       whether received start signal*/
Operational : the set of up links;

For ACK on link L:
 Deliver (ACK,L) to Reset

For RELEASE on link L:
  Deliver (RELEASE, L) to Reset

For ABORT on link L:
  Q:=Q-Q|L
  Deliver (ABORT,L) to Reset

For Link L going down:
  Q:=Q-Q|L
  Operational:=Operational-L;
  Deliver (Down,L) to Reset;

For link L coming up:
  Operational:=Operational+L;
  Deliver (Up,L) to Reset;

For start signal from outside:
  initiator := true;
  if State=Ready
  then deliver start signal to π;

For message M of π arriving on link L:
  if State=Aborting then queue(M,L);
  else Used:=Used+L;
       Deliver (M,L) to π

For Aborting to Ready state change by Reset:
  Used := ∅;
  State π:= initial state; /*reset π locally*/
  while not empty(Q) do
       (M,e) := dequeue(Q);
       Used := Used + e;
       Deliver (M,e) to π;
  if initiator
  then deliver start signal to π;
```

The Manager Process

Figure 2.1: The Reset Procedure and Manager Process of Afek et al. [AAG87]

Chapter 3

Event-oriented description of the system

We now reformulate and extend the reset procedure as given in [AAG87] to a version that is sufficiently exact and complete to prove correct. We need a description of everything that happens in the nodes and links, not only the actions of the reset algorithm. The description should be in the form of state variables and events of nodes and links. An event is considered as atomic. It is either executed fully, or not executed at all. An event may consist of a number of actions, like receiving a message from a link, changing the value of a local variable, and sending a message over a link. The order of events is arbitrary. If an event starts with a condition, it may take place any moment this condition is true. If it does not start with a condition, the implicit condition is "true".

Node variables and reset events of nodes are derived from the variables and pieces of code given in [AAG87]. We added π-events of nodes. All events of the nodes start with the reception of a message (with as condition that a message of this type is the first message in the queue specified), perform some internal computations, and send zero or more messages.

We assume that there is at most one link between two nodes, because this simplifies our notation. We denote a link between node i and node j as (i,j) and also as (j,i). Thus (i,j) and (j,i) are the same link. This assumption does not influence our proof. If more links between nodes are allowed, the way to denote a link should be adapted, e.g. (i,j,n), where n is a number that is unique for each link between node i and node j. We model links as follows:
A link (i,j) is a process with three variables: a variable LinkState((i,j)) with values Up and Down, and two message queues: Queue(i,j), containing messages sent by i to j, and Queue(j,i), containing messages sent by j to i. A Receive action takes the first message from a queue. A Send action places the message at the end of the queue if the link is up, and does nothing if the link is down. Nodes only try to send if they think the link is up, but there is a delay between a link going down and a node being notified of this.

Events of links are 'coming up' and 'going down'. If a link between node i and j comes up an 'Up' message is enqueued in Queue(i,j) and in Queue(j,i). A 'going down' event can be modeled in two ways. If a link goes down, the queues could be emptied, except for an Up message if present, and then a 'Down' message is enqueued in each queue. Here messages in the queue correspond to messages present in the link itself. Or a 'Down'

message could be placed at the end of both queues without emptying them. Here the queues correspond to queues present in the communication software. We shall use the first model in our proof, and afterwards verify that the second model also leads to a valid proof.

The links should behave according to axioms A3 and A4. That an interval should have no more than one mate means that an operating interval at one side of a link should not overlap with more than one interval at the other side. Such an overlap could occur if a link goes down and comes up again before a node is informed of the going down. This node may then send a sequence of messages that arrive at the node at the other end of the link in two distinct intervals. So axioms A3 and A4 translate into:

> A link is not allowed to come up, if there is still a Down message in one of its queues.

Figure 3.1 gives the variables and the reset events of the nodes with some procedures, and Figure 3.2 gives the π-events of the nodes and the variables and events of the links. We use a kind of pseudo-Pascal to describe events. Instead of if-then-begin-end-else-begin-end we use if-then-else-fi. We also assume that we have variables and operators for sets. A+a means: add element a to set A. A+:=a means: A:=A+a. A-a means: remove element a from set A if present. **for all** x $\in$ A **do** S **od** performs statement S for each element x of set A in an arbitrary order.

Because an Aborting node is not influenced by π-messages (an Aborting node upon receiving a π-message only puts it into a queue and takes no other actions) the reset procedure may be called an algorithm.

Our notation differs slightly from the one used in [AAG87]. As we did not consider the reception of a message by the Manager a separate event, we merged the Manager actions with the reset events. All events start with the reception of a message. We added to nodes i a variable UpLinks(i) that contains the set of all adjacent nodes j such that link (i,j) is up as far as node i knows. Two nodes i and j are **adjacent** if there exists a link (i,j) between nodes i and j. (It does not matter if this link is up or down.) This variable UpLinks(i) is needed by the π-algorithm to decide if a π-message may be sent over a link. Also we divided the queue Q for all π-messages of a node i of [AAG87] into a number of queues called πqueues, one for each link (i,j) adjacent to i. As we do not specify in which order these queues are emptied, this does not change the reset algorithm, and it simplifies the notation while describing the events.

3.1 The Global Observer

For the proof we also need a bookkeeping mechanism, that records facts about the history of an execution of the reset algorithm. This bookkeeping mechanism should be seen as an external global observer, who perceives all states and events of the system, but is not able to influence the system.

Actions of the Global Observer are triggered by events of nodes and links. Therefore we added these actions to the events that trigger them. Figure 3.3 contains the reset events of the nodes with procedures. Figure 3.4 contains the π-events of the nodes and the events of the links, and the variables and procedures of the Global Observer. Actions of the Global Observer are in italic.

Variables of node i:
State(i)
Parent(i)
Used(i)
initiator(i)
For all adjacent nodes j:
$AckPend_i(j)$
$\pi queue_i(j)$

Events of node i:
Reset events:

```
1) if Receive(Down) from (j,i)
   then πqueue_i(j):= ∅;
        Used(i) := Used(i) - j;
        UpLinks(i) := UpLinks(i) - j;
        if State(i) = Ready
        then Parent(i) := nil;
             Broadcast-Abort
        else if Parent(i) = j
             then Parent(i) := nil
             fi;
             if AckPend_i(j) or Parent(i)=nil
             then AckPend_i(j):= false;
                  Converge-Abort
   fi   fi   fi;

2) if Receive(Up) from (j,i)
   then UpLinks(i) := UpLinks(i) + j;
        if State(i) = Ready
        then Parent(i) := nil;
             Broadcast-Abort
   fi   fi;

3) if Receive(Abort) from (j,i)
   then πqueue_i(j):= ∅;
        if State(i) = Ready
        then Parent(i) := j;
             Broadcast-Abort
        else Send(Ack) to (i,j)
   fi   fi;

4) if Receive(Ack) from (j,i)
   then AckPend_i(j):= false;
        Converge-Abort
   fi;

5) if Receive(Release) from (j,i)
   then if j = Parent(i)
        then Release
   fi   fi;
```

```
procedure Broadcast-Abort:
   State(i) := Aborting;
   for all j ∈ Used(i)
   do  AckPend_i(j):= true;
       Send(Abort) to (i,j)
   od ;
   if Used(i) = ∅
   then Release
   fi;

procedure Converge-Abort:
   if ∀j ∈ Used(i)
        AckPend_i(j)= false
   then if Parent(i) ≠nil
        then Send(Ack) to (i,Parent(i))
        else Release
   fi   fi;

procedure Release:
   for all j ∈ Used (i)
   do  Send(Release) to (i,j)
   od ;
   State(i) := Ready;
   Used(i) := ∅;
   State π := initial state;
   for all j with πqueue_i(j)≠∅
   do  Used(i) := Used(i) + j;
       while πqueue_i(j)≠∅
       do dequeue(M,πqueue_i(j));
            deliver M to π
   od  od ;
   if initiator(i)
   then deliver Start to π
   fi;
procedure Send(M) to (i,j):
   if LinkState((i,j)) = Up
   then enqueue(M,Queue(i,j))
   fi;

procedure Receive(M) from (j,i):
   if first message in Queue(j,i) is M
   then dequeue(M,Queue(j,i))
   fi;
```

Figure 3.1: Variables and Reset events of nodes with procedures

```
6) if Receive(π-message) from (j,i)
  then if State(i) = Ready
       then Used(i) := Used(i) + j;
            compute;
            for some k ∈ UpLinks(i) do
                Send(π-message) to (i,k);
                Used(i) := Used(i) + k
            od
       else enqueue(π-message,πqueue_i(j))
  fi   fi;

7) if Receive(Start) from (Outside,i)
  then initiator(i) := true;
       if State(i) = Ready
       then compute;
            for some k ∈ UpLinks(i) do
                Send(π-message) to (i,k);
                Used(i) := Used(i) + k
            od
  fi   fi;
```

Variables of link (i,j):

LinkState((i,j))
Queue(i,j)
Queue(j,i)

Link Events:

```
8) if LinkState( (i,j) ) = Up
  then LinkState((i,j)) := Down;
       DiscardAllExceptUp(Queue(i,j));
       DiscardAllExceptUp(Queue(j,i));
       enqueue(Down,Queue(i,j));
       enqueue(Down,Queue(j,i))
  fi;

9) if LinkState( (i,j) ) = Down
       and Down ∉ Queue(i,j)
       and Down ∉ Queue(j,i)
  then LinkState( (i,j) ) := Up;
       enqueue(Up,Queue(i,j));
       enqueue(Up,Queue(j,i))
  fi;
```

Figure 3.2: π-events of nodes and variables and events of links

The Global Observer records which topological change in the network caused a node now in Aborting state to enter this state. To do this the observer has a variable AbortId(i) for each node i. The observer associates a new unique identification a to each 'Down' or 'Up' message that is generated in the network. If a Ready node i receives a Down-a or Up-a, a is assigned to AbortId(i). a is also attached to Abort messages sent by i. If a Ready node j receives an Abort-a message, a is assigned to AbortId(j), and attached to Abort messages sent by j. Old values of AbortId(i) are kept in AbortSet(i). We will use State(i)=Aborting-a as a shorthand notation for: State(i)=Aborting and AbortId(i)=a.

To describe the other actions of the observer we need some definitions:

Definition A node j is a **child** of node i if i and j are both Aborting with the same AbortId, and there is a link between i and j marked as parent link by j, and there is no Down or Release message in Queue(i,j).

Definition The **Aborting-a Group** consists of all nodes i with AbortId(i)= a.

Definition The **Aborting-a Graph** consists of all nodes i in the Aborting-a Group and all links (i,j) with Parent(i)=j and Down $\notin$ Queue(j,i) and Release $\notin$ Queue(j,i).

The Aborting-a Graph has directed links. If link (i,j) is a link of the graph and Parent(j)=i, the link is directed from j to i. We will need this to prove that this Aborting Graph is indeed a graph. (It will even turn out to be a forest.)

If a link goes down that is marked as parent by an Aborting node i, i obtains a new unique id, and also nodes that are children of a node with this new id obtain this new id. This new id is also attached to Abort messages sent by these nodes. The old id is not added to the AbortSets of the nodes.

Events of node i:
Reset events:

```
1) if Receive(Down-a) from (j,i)
   then πqueue_i(j):= ∅;
        Used(i) := Used(i) - j;
        UpLinks(i) := UpLinks(i) - j;
        if State(i) = Ready
        then Parent(i) := nil;
             Broadcast-Abort(a)
        else if Parent(i) = j
             then Parent(i) := nil
             fi;
             if AckPend_i(j)
             then Count(i):=Count(i)-1;
             fi;
             if AckPend_i(j)
                  or Parent(i)=nil
             then AckPend_i(j):=false;
                  Converge-Abort
   fi   fi   fi;

2) if Receive(Up-a) from (j,i)
   then UpLinks(i):=UpLinks(i)+j;
        if State(i) = Ready
        then Parent(i) := nil;
             Broadcast-Abort(a)
   fi   fi;

3) if Receive(Abort-a) from (j,i)
   then πqueue_i(j):= ∅;
        if State(i) = Ready
        then Parent(i) := j;
             Broadcast-Abort(a)
        else Send(Ack) to (i,j)
   fi   fi;

4) if Receive(Ack) from (j,i)
   then AckPend_i(j):= false;
        Count(i) := Count(i) - 1;
        Converge-Abort
   fi;

5) if Receive(Release) from (j,i)
   then if j = Parent(i)
        then Release
   fi   fi;
```

```
procedure Broadcast-Abort(a):
   State(i) := Aborting;
   AbortId(i) := a;
   πState := Inactive;
   Count(i) := #{ j | j ∈ Used(i) }
   for all j ∈ Used(i)
   do  AckPend_i(j):= true;
       Send(Abort-a) to (i,j)
   od ;
   if Used(i) = ∅
   then Release;
   fi;

procedure Converge-Abort:
   if ∀j ∈ Used(i)
        AckPend_i(j)= false
   then if Parent(i) ≠nil
        then Send(Ack) to (i,Parent(i))
        else Release
   fi   fi;

procedure Release:
   for all j ∈ Used (i)
   do  Send(Release) to (i,j)
   od ;
   State(i) := Ready;
   Used(i) := ∅;
   State π := initial state;
   AbortSet(i) +:= AbortId(i);
   AbortId(i) := Undefined;
   for all j with πqueue_i(j)≠∅
   do  Used(i) := Used(i) + j;
       while πqueue_i(j)≠∅
       do dequeue(M,πqueue_i(j));
            deliver M to π
   od  od ;
   if initiator(i)
   then deliver Start to π
   fi;

procedure Send(M) to (i,j):
   if LinkState((i,j)) = Up
   then enqueue(M,Queue(i,j))
   fi;

procedure Receive(M) from (j,i):
   if first mess in Queue(j,i) is M
   then dequeue(M,Queue(j,i))
   fi;
```

Figure 3.3: Reset events of nodes with actions of the Global Observer incorporated

```
6) if Receive(π-message) from (j,i)
  then if State(i) = Ready
       then Used(i) := Used(i) + j;
            if πState = Inactive
            then πState := πexec
            fi;
            for some k ∈ UpLinks(i) do
                 Send(π-message) to (i,k);
                 Used(i) := Used(i) + k
            od
       else enqueue(π-message,πqueue_i(j))
  fi   fi;

7) if Receive(Start) from (Outside,i)
  then initiator(i) := true;
       if State(i) = Ready
       then compute;
            for some k ∈ UpLinks(i) do
                 Send(π-message) to (i,k);
                 Used(i) := Used(i) + k
            od
  fi   fi;
```

Link Events:

```
8) if LinkState( (i,j) ) = Up
  then LinkState((i,j)) := Down;
       Fetch new unique ids a and b;
       RenameIds(a,i,j);
       RenameIds(b,j,i);
       MakeObsolete(i);
       MakeObsolete(j);
       DiscardAllExceptUp(Queue(i,j));
       DiscardAllExceptUp(Queue(j,i));
       enqueue(Down-a,Queue(i,j));
       enqueue(Down-b,Queue(j,i))
  fi;

9) if LinkState( (i,j) ) = Down
       and Down ∉ Queue(i,j)
       and Down ∉ Queue(j,i)
  then LinkState( (i,j) ) := Up;
       Fetch new unique ids a and b;
       MakeObsolete(i);
       MakeObsolete(j);
       enqueue(Up-a,Queue(i,j));
       enqueue(Up-b,Queue(j,i))
  fi;
```

Variables of Global Observer for each node i:

AbortId(i)
AbortSet(i)
πState(i)
Count(i)

Procedures for Global Observer:

```
Procedure MakeObsolete(i):
  if State(i) = Ready and
       πState(i) = πexec
  then πState(i) := Obsolete;
       while ∃ link (j,k) with
            (j,k) is π-marked and
            πState(j) = Obsolete
       do   πState(k) := Obsolete
       od
  fi;

Procedure RenameIds(a,i,j):
  if Abort ∈ Queue(i,j)
  then the id of this Abort becomes a
  fi;
  if State(j)=Aborting
       and Parent(j)=i
       and Down ∉ Queue(i,j)
       and Release ∉ Queue(i,j)
  then AbortId(j):= a
       for all nodes k adjacent to j
       do RenameIds(a,j,k)
       od
  fi;
```

Figure 3.4: π-events of nodes and events of links with actions of The Global Observer incorporated

The state of each node alternates between Ready and Aborting. A Ready node may be inactive, executing the π-algorithm, or participating in an obsolete π-computation. The Global Observer records this in a variable πState, with values Inactive, πexec and Obsolete. A Ready node i executing the π-algorithm becomes **obsolete** if:
1) An adjacent link changes its state
2) A π-message, sent by i, is received by an obsolete node j
3) i receives an obsolete π-message
4) A node j that received or dequeued a π-message from i in j's current Ready interval becomes obsolete
5) A node j that sent a π-message received or dequeued from πqueue$_i$(j) by i in its current Ready interval becomes obsolete.
An Aborting node is never obsolete.

Definition If an Obsolete node sends a π-message, this π-message is obsolete.

Definition A π-message becomes obsolete if the node that sent it becomes obsolete.

The observer groups nodes executing the π-algorithm into so-called connected groups of π-nodes:

Definition A **π-marked link** is a link between two Ready nodes i and j, where i received in its current Ready interval a π-message sent by j, or vice versa, and LinkState((i,j)) = Up.

Definition A **connected group of π-nodes** is a maximal connected subgraph of the network consisting of Ready nodes executing the π-algorithm and π-marked links.

In the proof we often need to know if a node i is still waiting for Acks.

Definition Acked(i) = $\forall$j $\in$ Used(i): AckPend$_i$(j)=false.

The Global Observer also counts for each node i in variable Count(i) the number of receptions of Acks and Downs over used links while i is Aborting. This is used to prove termination of the algorithm.

3.2 Initialization

The system should start with no messages on links or in πqueues. Links may be up or down, but the values of UpLinks of the adjacent nodes must reflect the states of the link. All nodes are Ready and inactive, so no links are used for a π-computation and no Acks are pending.

The variables of node i are initialized to:
State(i):=Ready,
UpLinks(i):={nodes j connected to i such that link (i,j) is up},
Used(i):= $\emptyset$,
Parent(i):=nil,
for all nodes j: AckPend$_i$(j):= false.

The variables of link (i,j) are initialized to:
LinkState((i,j)):= Up or Down,

Queue(i,j):= $\emptyset$,
Queue(j,i):=$\emptyset$.

The variables of the Global Observer for each node i are initialized to:
AbortId(i):=Undefined,
πState(i) :=Inactive,
AbortSet(i):=$\emptyset$,
Count(i) := 0.

Chapter 4

The proof

To prove the correctness of the reset algorithm, we prove two things:

1. The reset algorithm terminates after the last topological change of the network,
2. After the termination of the reset algorithm there are no more obsolete nodes or obsolete π-messages in the system.

The reset algorithm terminates if in finite time after the last topological change there are no nodes executing the reset algorithm and the links contain no reset messages. The reset algorithm is correct if after termination there are no more obsolete nodes or obsolete π-messages.

To prove termination, we first prove that an Aborting-a Graph is a forest (section 1). Then we prove that an Aborting-a Graph is a tree if there is still an unacked Aborting-a node (section 2). We use these results to prove that a node never participates more than once in the same Abort Group (section 3). By proving that only a finite number of reset events of a node happen while this node participates in an Aborting-a Group (section 4) we prove that after the last topological change only a finite number of reset events is possible. Deadlock could prevent termination of the reset algorithm. In section 5 we prove that deadlock does not occur. In section 6 we prove correctness of the reset algorithm. And finally in section 7 we adapt the proof of termination and correctness to the second model for the behavior of links.

Often the invariants we prove are of the form A $\Longrightarrow$B,
e.g. Ack $\in$ Queue(j,i) $\Longrightarrow$State(i) = Aborting and AckPend$_i$(j)= true. Proving such invariants consists of:

1. Checking that the invariant holds after initialization. As this is often trivial because the premise does not hold, we omit this step in most proofs.
2. Verifying that an event cannot make the premise A true while the conclusion B is false.
3. Verifying that an event cannot make the conclusion B false while the premise A is true.

For step 2 and 3 we list all actions that could make the invariant false, and check each event in which one or more of these actions could occur.

4.1 An Aborting-a Graph is a forest

First we prove some lemmas about links being up or down, and messages present in the link queues and in πqueues.

Lemma 1.1 Down $\in$ Queue(i,j) $\Longrightarrow$LinkState((i,j))=Down.

Proof:
The dangerous actions are:

- A Down is put into Queue(i,j).
  8) Link (i,j) goes down. Then LinkState((i,j)):=Down.
- LinkState((i,j)):=Up.
  9) Link (i,j) comes up. This is only possible if Down $\notin$ Queue(i,j). □

Lemma 1.2 LinkState((i,j))=Down $\Longrightarrow$
Down $\in$ Queue(i,j) as the last message
or Queue(i,j)=$\emptyset$.

Proof:
The dangerous actions are:

- LinkState((i,j)):=Down.
  8) Link (i,j) goes down. Then a Down is put into Queue(i,j) as the last element.
- a Down is removed from Queue(i,j).
  1) j receives a Down from Queue(i,j). Then Queue(i,j):=$\emptyset$.
- M is put into Queue(i,j).
  This is not possible if LinkState((i,j))=Down. □

Lemma 1.3 M $\in$ Queue(i,j) and M$\neq$Down and M $\neq$Up
$\Longrightarrow$LinkState((i,j))=Up.

Proof:
The dangerous actions are:

- M ($\neq$Down and $\neq$Up) is put into Queue(i,j).
  This is only possible if LinkState((i,j))=Up.
- LinkState((i,j)):=Down.
  Then all messages except Up are discarded. □

Lemma 1.4 Up $\in$ Queue(i,j) $\Longrightarrow$Up is the first message in Queue(i,j).

Proof:
The dangerous actions are:

- An Up is put into Queue(i,j).
  9) Link (i,j) comes up. Link (i,j) was down and Down $\notin$ Queue(i,j), so Queue(i,j) was empty (1.1). □

If an Aborting node has a parent link and this link is down, there is a Down message on this link on its way to the node:

Lemma 1.5 State(i)=Aborting and Parent(i)=j and LinkState((i,j))=Down
$\Longrightarrow$Down $\in$ Queue(j,i).

Proof:
The dangerous actions are:

- State(i):=Aborting
  1) 2) i is Ready and receives a Down or Up. Then Parent(i):=nil.
  3) i is Ready and receives an Abort from Queue(k,i). Then Parent(i)=k. If k$\neq$j: Parent(i)$\neq$j. If k=j holds, then LinkState((i,j)) $\neq$Down (1.3).
- Parent(i):=j.
  3) is already treated.
- LinkState((i,j)):=Down.
  8) A Down is placed into Queue(j,i).
- A Down disappears from Queue(j,i).
  1) i is Aborting and Parent(i)=j and i receives a Down from Queue(j,i). Then Parent(i):=nil. □

If there is an Abort message on a link with id *a*, the node that sent it is Aborting with AbortId=*a*, or a Release message is behind the Abort message.

Lemma 1.6 Abort-*a* $\in$ Queue(i,j) $\Longrightarrow$
State(i)=Aborting and AbortId(i)=*a* and j $\in$ Used(i)
or Release $\in$ Queue(i,j), behind Abort-*a*.

Proof:
The dangerous actions are:

- Abort-*a* is put into Queue(i,j).
  1)2)3) i is ready and receives a Down-*a*, Up-*a*, or Abort-*a*. Then State(i):=Aborting and AbortId:=*a* and i only sends an Abort-*a* on link (i,j) if j $\in$ Used(i).
- Abort-*b* in Queue(i,j) becomes Abort-*a*.
  8) A link goes down and the id of the Abort message becomes *a*. Then also AbortId(i) := *a*.
- State(i):= Ready.
  1)4)5) i is Aborting and AbortId(i)=*a* and j $\in$ Used(i) and i receives a Down, Ack, or Release, and calls procedure Release. Then i places a Release message into Queue(i,j). Since j $\in$ Used(i), procedure Release is not called from Broadcast-Abort.
- j is removed from Used(i).
  1)4)5) i is Aborting and calls procedure Release. Is already treated.
  1) i receives a Down from Queue(j,i). Then LinkState((i,j)) = Down (1.3), so there is no Abort in Queue(i,j).

- AbortId(i):=b, $b \neq a$.
  8) i is Aborting, AbortId(i)=a, some link goes down, and AbortId(i) := b. Then also the id of the Abort message becomes b. □

A Parent link of an Aborting node points to a parent of the node, or a Down or Receive is on its way to the node.

Lemma 1.7 State(j)=Aborting and AbortId(j)=a and Parent(j)=i $\Longrightarrow$
State(i)=Aborting and AbortId(i)=a and j $\in$ Used(i)
or Down $\in$ Queue(i,j)
or Release $\in$ Queue(i,j).

Proof:
The dangerous actions are:

- State(j):=Aborting and AbortId(j):=a and Parent(j):=i.
  3) j is Ready and receives an Abort-a from Queue(i,j). Then State(i)=Aborting and AbortId(i)=a and j $\in$ Used(i), or Release $\in$ Queue(i,j) (Lemma 1.6).

- AbortId(j):=a.
  8) AbortId(j)=b, link (i,j) goes down and AbortId(j):=a. Then a Down is put into Queue(i,j).
  8) Another link goes down and AbortId(j):=a. Then also AbortId(i):=a, unless Down or Release $\in$ Queue(i,j).

- State(i):=Ready.
  1)4)5) i is Aborting and j $\in$ Used(i) and i receives a Down, Ack or Release and calls procedure Release. Then i sends a Release message to j. If link (i,j) is down Queue(i,j) contains a Down (Lemma 1.5).

- AbortId(i):=b, $b \neq a$.
  8) i is Aborting, AbortId(i)=a, some link goes down, and AbortId(i):=b. Then AbortId(j):=b, or Queue(i,j) contains a Down or Release message.

- j is removed from Used(i).
  1)4)5) i is Aborting, receives a Down, Ack, or Release, and calls procedure Release. This is already treated.
  1) i is Aborting and receives a Down from Queue(j,i). Then Queue(i,j) contains a Down(Lemma 1.5).

- Down is removed from Queue(i,j).
  1) j is Aborting and Parent(j)=i and j receives a Down from Queue(i,j). Then Parent(j):=nil.

- Release is removed from Queue(i,j).
  8) Link (i,j) goes down. Then a Down is placed into Queue(i,j).
  5) j is Aborting and Parent(j)=i and j receives a Release from Queue(i,j). Then State(j):=Ready. □

Now we are able to prove that an Aborting Graph is a forest.

Theorem 1 Nodes i with AbortId=a, plus links (i,j) with Parent(i)=j and Down $\notin$ Queue(j,i) and Release $\notin$ Queue(j,i), form a forest.

Proof:
The dangerous actions are:

- A new Aborting-a Graph comes into existence.
  1)2) Node i is Ready and receives a Down-a or Up-a and a is a new unique id. Node i becomes a tree of one node.
  8) A link in an Aborting-b Graph goes down and the Global Observer marks part of the forest. The Global Observer only marks a subtree of the Aborting-b forest.

- A link is added to an existing forest of nodes with AbortId a.
  3) Node i is Ready and receives an Abort-a from Queue(j,i). Then State(j)=Aborting and AbortId(j)=a, or Release $\in$ Queue(j,i) (Lemma 1.6). So the new link is adjacent to two nodes of the Abort Graph, and no circuit is introduced into the forest, because node i was not in the Abort Graph before the event.

- A node is removed from a forest of nodes with AbortId a.
  1)4)5) Node i is Aborting, receives a Down or Ack and Parent(i)=nil, or receives a Release over its parent link, and calls procedure Release. Then all links (i,j) that belong to the forest lead to children of i. If j is a child of i, then j $\in$ Used(i) (Lemma 1.7). So i sends a Release to all its children, and so removes all links incident to itself from the forest.
  8) A link from the forest goes down and part of the Aborting-a nodes are renamed. Then only a subtree is renamed, so the forest stays a forest. □

4.2 An Aborting-a Group is a tree if there is an unacked Aborting-a node

We first need a small lemma:

Lemma 2.1 AckPend$_i$(j)=true $\Longrightarrow$j $\in$ Used(i) and not Acked(i).

Proof:
The dangerous actions are:

- AckPend$_i$(j):=true.
  1)2)3) i is Ready and receives an Up,Down or Abort. Then: AckPend$_i$(j):=true $\Longleftrightarrow$j $\in$ Used(i) and not Acked(i).

- j is removed from Used(i).
  1) i receives a Down from Queue(j,i). Then AckPend$_i$(j):=false.
  1)2)3)4)5) i calls procedure Release. Then Used(i) was empty or i becomes Acked. So AckPend$_i$(j) is false.

- i becomes Acked.

  1)4) i receives a Down or Ack from Queue(j,i). Then $AckPend_i(j)$:=false. □

We now prove that there are no straggling messages during execution of the reset algorithm. A node that enters Aborting state sends Abort messages, waits for all Acks (or Downs), and only then sends an Ack to its parent, or releases if it has no parent. If an Abort is present in Queue(i,j), then no straggling Ack is present in Queue(j,i). If an Ack is present in Queue(j,i), this Ack is expected by i: i is Aborting and $AckPend_i(j)$= true. If Queue(i,j) contains a Release and j is Aborting and Parent(j)=i, then j is Acked. We prove thirteen of these statements simultaneously.

Lemma 2.2

a) Abort-$a \in$ Queue(i,j) $\Longrightarrow$State(i)=Aborting and AbortId(i)=a and $AckPend_i(j)$= true.

b) Abort $\in$ Queue(i,j) $\Longrightarrow$Ack $\notin$ Queue(j,i).

c) Abort $\in$ Queue(i,j) $\Longrightarrow$not(Release behind Abort in Queue(i,j)).

d) Queue(i,j) never contains two Aborts.

e) Ack $\in$ Queue(j,i) $\Longrightarrow$(State(j)=Aborting and Parent(j)=i and Acked(j)) or Parent(j) $\neq$i or State(j)= Ready.

f) Ack $\in$ Queue(j,i) $\Longrightarrow$State(i)=Aborting and $AckPend_i(j)$= true.

g) Ack $\in$ Queue(j,i) $\Longrightarrow$Release $\notin$ Queue(i,j).

h) Queue(i,j) never contains two Acks.

i) Release $\in$ Queue(i,j) and State(j)=Aborting and Parent(j)=i $\Longrightarrow$Acked(j).

j) State(j)=Aborting and Parent(j)=i and not Acked(j) $\Longrightarrow$Abort $\notin$ Queue(i,j).

k) State(j)=Aborting and Parent(j)=i and not Acked(j) $\Longrightarrow$$AckPend_i(j)$= true or Down $\in$ Queue(i,j).

l) State(j)=Aborting and Parent(j)=i and $AckPend_i(j)$= false $\Longrightarrow$Acked(j) or Down $\in$ Queue(i,j).

m) State(i)=Ready $\Longrightarrow$Acked(i).

Proof:
Assume all statements are true for all nodes before an event. Check that all statements remain true after each possible event.

a) Abort-$a \in$ Queue(i,j) $\Longrightarrow$State(i)=Aborting and AbortId(i)=a and $AckPend_i(j)$= true

The dangerous actions are:

- Abort-a is put into Queue(i,j).
  1)2)3) i is Ready and receives an Up-a, Down-a, or Abort-a.
  Then State(i):=Aborting-a and if i places an Abort-a into Queue(i,j) then j $\in$ Used(i) so AckPend$_i$(j):= true.

- State(i):=Ready.
  1) i receives a Down over its last unacked link. If AckPend$_i$(j) was true before the event, (i,j) was this last unacked link (2.1). Then (i,j) is down, and Queue(i,j)=$\emptyset$.
  4) i is Aborting and receives an Ack over its last unacked link. If AckPend$_i$(j)=true this link is (j,i) (2.1). But Queue(j,i) contains no Ack (2.2.b). 5) i receives a Release over its parent link. Then i is Acked (2.2.i). So AckPend$_i$(j)= false (2.1). Contradiction.

- AbortId(i):=b, $b \neq a$.
  8) i is Aborting-a , some link goes down, and AbortId(i):=b. Then also the id of the Abort message in Queue(i,j) becomes b.

- AckPend$_i$(j):=false.
  1) i receives Down from Queue(j,i). Then Abort $\notin$ Queue(i,j).
  4) i receives an Ack from Queue(j,i). Then there is no Abort in Queue(i,j) (2.2.b).

b) Abort $\in$ Queue(i,j) $\Longrightarrow$Ack $\notin$ Queue(j,i) .

The dangerous actions are:

- Abort is put into Queue(i,j).
  1)2)3) i is ready and receives an Up, Down, or Abort. Then Ack $\notin$ Queue(j,i) (2.2.f).

- An Ack is put into Queue(j,i).
  1) j is Aborting, receives a Down from the last unacked link, and sends an Ack to parent i. Then Abort $\notin$ Queue(i,j) (2.2.j).
  3) j is Aborting, receives an Abort from Queue(i,j) and places an Ack into Queue(j,i). Then there is no second Abort in Queue(i,j) (2.2.d).
  4) j receives an Ack over link k. Then j was Aborting and AckPend$_j$(k) was true (2.2.f), so k $\in$ Used(j) (2.1). Then Abort $\notin$ Queue(i,j) 2.2.j).

c) Abort $\in$ Queue(i,j) $\Longrightarrow$not (Release behind Abort in Queue(i,j)).

The only dangerous action is:

- Release is put into Queue(i,j)
  i is Aborted and AckPend$_i$(j)= true. (2.2.a). If there is an Abort in Queue(i,j), then there is no Down in Queue(j,i), nor an Ack in Queue(j,i) (2.2.b). And also there is no Release in the parent link of i (2.2.i). So events that send Releases are not possible.

d) Queue(i,j) never contains two Aborts.

The only dangerous action is:

- Second Abort is put into Queue(i,j).
  If there is an Abort in Queue(i,j), then i is Aborting (2.2.a). So sending of a second Abort is not possible.

e) Ack $\in$ Queue(j,i) $\Longrightarrow$(State(j)=Aborting and Parent(j)=i and Acked(j))
or Parent(j) $\neq$i or State(j)=Ready.

The dangerous actions are:

- An Ack is put into Queue(j,i).
  1)4) j is Aborting, receives a Down or Ack from the last unacked link, and sends an Ack to its parent i. Then j is Aborting, Acked, and Parent(j)=i.
  3) j is Aborting and receives an Abort from Queue(i,j). If j is unacked then Parent(j) $\neq$i (2.2.j).

- j becomes Aborting and unacked and Parent(j):=i.
  1)2)3) j is Ready and receives an Abort from Queue(i,j). Then there is no Ack in Queue(j,i) (2.2.b).

f) Ack $\in$ Queue(j,i) $\Longrightarrow$State(i)=Aborting and $AckPend_i(j)$= true.

The dangerous actions are:

- An Ack is put into (j,i).
  1) j is Aborting and parent(j)=i, j receives a Down over the last unacked link, and places an Ack into Queue(j,i). There is no Release in Queue(i,j) (2.2.i), so i is Aborting (1.7) and $AckPend_i(j)$= true (2.2.k).
  3) j is Aborting and receives an Abort from Queue(i,j). Then i is Aborting and $AckPend_i(j)$ is true (2.2.a).
  4) j is Aborting and receives an Ack from Queue(k,j). Then $AckPend_j(k)$ was true (2.2.f for k,j). Then j was unacked (2.1). So there is no Release in Queue(i,j) (2.2.i), hence i is Aborting (1.7) and $AckPend_i(j)$= true (2.2.k).

- i becomes Ready.
  1) i receives a Down over the last unacked link. If $AckPend_i(j)$= true this is link (j,i). Then there is no Ack in Queue(j,i).
  4) i receives an Ack from Queue(j,i). There is no second Ack in Queue(j,i) (2.2.h).
  5) i receives a Release from its parent link. This is not possible if i is unacked (2.2.i).

- $AckPend_i(j)$:= false.
  1) i receives a Down from Queue(j,i). Then there is no Ack in Queue(j,i).
  4) i receives an Ack from Queue(j,i). There is no second Ack in Queue(j,i) (2.2.h).

g) Ack $\in$ Queue(j,i) $\Longrightarrow$Release $\notin$ Queue(i,j).

The dangerous actions are:

- An Ack is put into Queue(j,i).
  1) j is Aborting, Parent(j)=i, and j receives a Down over the last unacked link. Then Release $\notin$ Queue(i,j) (2.2.i).

3) j is Aborting and receives an Abort from Queue(i,j). Then Release $\notin$ Queue(i,j) (2.2.c).
4) j is Aborting, parent(j)=i, and j receives an Ack from Queue(i,j).
Then $AckPend_j(i)$ was true (2.2.f). Then j was unacked (2.1). Then Release $\notin$ Queue(i,j) (2.2.i).

- A Release is put into Queue(i,j).
  1) i is Aborting and receives a Down from the last unacked link. If Ack $\in$ Queue(j,i) then $AckPend_i(j)$= true (2.2.f). So this last unacked link is (i,j). But this link is down, so no Release is placed into Queue(i,j).
  4) i is Aborting , Parent(i)=nil, i receives an Ack, becomes Acked and sends a Release to j. If Ack $\in$ Queue(j,i) then $AckPend_i(j)$=true (2.2.f). So i is unacked (2.1). So i received this Ack from Queue(j,i). Then there is no second Ack in Queue(j,i) (2.2.h).
  5) i is Aborting, Parent(i)=k, and i receives a Release from Queue(k,i). Then i is Acked (2.2.i), so Ack $\notin$ Queue(j,i) (2.2.f, 2.1).

h) Queue(j,i) never contains two Acks.

The only dangerous action is:

- A second Ack is put into Queue(j,i).
  1) j is Aborting and Parent(j)=i and j receives a Down from the last unacked link. If Queue(j,i) contained an Ack already then j was Acked(2.2.e). Contradiction.
  3) j is Aborting and receives an Abort from Queue(i,j). Then there was no Ack in Queue(j,i) (2.2.b).
  4) j receives an Ack from Queue(k,j). Then $AckPend_j(k)$was true (2.2.f), and so j was unacked (2.1), so Ack $\notin$ Queue(j,i) (2.2.e).

i) (Release $\in$ Queue(i,j) and State(j)=Aborting and Parent(j)=i) $\Longrightarrow$Acked(j).

The dangerous actions are:

- A Release is put into Queue(i,j).
  1)4) i is Aborting and Parent(i)=nil and i receives a Down from its last unacked link, or an Ack, and sends a Release to j. Then j $\in$ Used(i), so $AckPend_i(j)$ was false. Then Acked(j) holds (2.2.l).
  5) i receives a Release from its parent. If i was Aborting then i was Acked before the event (2.2.i). Also if i was Ready then i was Acked before the event (2.1). So $AckPend_i(j)$was false before the event (2.2.m). Then Acked(j) held before the event (2.2.l), and also holds after the event.

- State(j):=Aborting and Parent(j):=i.
  3) j is Ready and receives an Abort from Queue(i,j). Then Release $\notin$ Queue(i,j) (2.2.c).

- j becomes not Acked.
  1)2)3) This is only possible if State(j)=Ready.

j) State(j)=Aborting and Parent(j)=i and not Acked(j) $\Longrightarrow$Abort $\notin$ Queue(i,j).

The dangerous actions are:

- State(j):=Aborting and Parent(j):=i and j becomes unacked.
  3) j is Ready and receives an Abort from Queue(i,j). There is no second Abort present in Queue(i,j) (2.2.d).
- an Abort is put into Queue(i,j).
  There is no Release in Queue(i,j)(2.2.i), so i is Aborting or (i,j) is down (1.7) so i cannot place an Abort in Queue(i,j).

k) State(j)=Aborting and Parent(j)=i and not Acked(j) $\Longrightarrow$AckPend$_i$(j)= true or a Down in Queue(i,j).

The dangerous actions are:

- State(j):=Aborting and Parent(j):=i and j becomes unacked.
  3) j is Ready and receives an Abort from Queue(i,j). Then AckPend$_i$(j)= true (2.2.a).
- AckPend$_i$(j):= false.
  1) i receives a Down from Queue(j,i). Then there is a Down in Queue(i,j) (1.5).
  4) i receives an Ack from Queue(j,i). Then not(j is Aborting and unacked and Parent(j)=i) (2.2.e)
- Down disappears from Queue(i,j).
  1) j receives a Down from Queue(i,j). Then Parent(j):=nil.

l) State(j)=Aborting and Parent(j)=i and AckPend$_i$(j)= false $\Longrightarrow$
Acked(j) or Down $\in$ Queue(i,j).

The dangerous actions are:

- State(j):=Aborting and Parent(j):=i.
  3) j is Ready and receives an Abort from Queue(i,j). Then AckPend$_i$(j)=true (2.2.a).
- AckPend$_i$(j):= false.
  1) i receives a Down from Queue(j,i). Then Down $\in$ Queue(i,j) (1.5). 4) i receives an Ack from Queue(j,i). Then Acked(j) holds (2.2.e).
- Down disappears from Queue(i,j).
  1) j receives a Down from Queue(i,j). Then parent(j):=nil.
- j becomes unacked.
  1)2)3) This is only possible if State(j)=Ready.

m) State(i)=Ready $\Longrightarrow$Acked(i).

Proof:
The dangerous actions are:

- State(i):=Ready.
  1)4) i is Aborting and receives a Down or Ack. Then i only calls procedure Release if i is now Acked.
  5) i is Aborting and receives a Release over its parent link. Then i is Acked (2.2.i).

- i becomes unacked.
  1)2)3) i is Ready and receives an Up, Down, or Abort. Then i becomes Aborting. □

Lemma 2.3 There are no nodes i with State(i)=Aborting and Acked(i) and Parent(i)=nil.

Proof:
The only dangerous action is:

- Such a node comes into existence.
  1)2) i is Ready, receives a Down or Up, and Used(i):= $\emptyset$. Then i becomes Aborted, Acked, and Ready again in one event.
  1) i is Aborting, receives a Down, becomes Acked, and Parent(i)=nil. Then i becomes Ready.
  4) i is Aborting, receives an Ack and becomes Acked. Then i becomes Ready. □

If a node is Acked, its children are also Acked:

Lemma 2.4 Acked(i) and State(j)=Aborting and Parent(j)=i and Down $\notin$ Queue(i,j) $\Longrightarrow$Acked(j).

Proof:
If State(j)=Aborting and Parent(j)=i and Down $\notin$ Queue(i,j) and Release $\notin$ Queue(i,j) then State(i)=Aborting and j $\in$ Used(i) (1.7). If State(j)=Aborting and Parent(j)=i and not Acked(j) and Down $\notin$ Queue(i,j), then AckPend$_i$(j)=true (2.2.k). Then i is unacked (2.1). So j must be Acked. □

Lemma 2.5 Down-a $\in$ Queue(i,j) or Up-a $\in$ Queue(i,j) $\Longrightarrow$
a is unique,
or Parent(j)=i and AbortId(j)=a.

Proof:
The dangerous events are:

- a Down-a or Up-a is put into Queue(i,j).
  8) Link (i,j) goes down. Then a is a new unique id. If j is Aborting and Parent(j)=i, then AbortId(j):=a. Otherwise a does not spread further in the network.
  9) Link (i,j) comes up. Then a is a new unique id, and there are no nodes renamed.

- a becomes "not unique".
  1)2) j is Ready, receives a Down-a or Up-a from Queue(i,j) and sends Abort-a messages. Then Down-a or Up-a, respectively, disappears from Queue(i,j).

- (Parent(j)=i and AbortId(j)=a) becomes false.
  1) j is Aborting-a, Parent(j)=i, and j receives a Down-a from Queue(i,j). Then Down-a disappears from Queue(i,j).
  5) j receives a Release from Queue(i,j). This is not possible while Queue(i,j) contains a Down or an Up (1.2, 1.4). □

Theorem 2 $\exists$i: AbortId(i)=a and not Acked(i)) $\Longrightarrow$
All nodes j with AbortId(j)=a plus all links (j,k) with Parent(j)=k and Down $\notin$ Queue(k,j) and Release $\notin$ Queue(k,j) form a directed tree.

Proof:
The dangerous actions are:

- A first unacked Aborting-a node is generated.
  1)2) i is Ready and receives a Down-a or Up-a and a is unique. Then the Aborting-a Group consists of 1 node: i, and no links.
  8) A link of an Aborting-b tree goes down and some Aborting-b nodes are changed to Aborting-a nodes. The new Aborting-a nodes plus their parent links form a tree.

- A node becomes Aborting-a.
  1) i is Ready and receives a Down-a. Then this a is unique (2.5).
  3) i is Ready and receives an Abort-a from Queue(j,i). Then j is Aborting-a and unacked (2.2.a), so the Aborting-a Graph was a tree before the event. The event adds node i plus link (i,j) to the tree. Down $\notin$ Queue(j,i) because Queue(i,j) contained an Abort message. Release $\notin$ Queue(i,j) because of 2.2.c. So after the event the Aborting-a Graph still is a tree.

- A link disappears from an Aborting-a tree.
  8) Link (i,j) goes down, and i and j are Aborting-a and Parent(j)=i. Then all nodes in the subtree with j as root obtain a unique new AbortId, and a Down is put into Queue(i,j).
  1)4)5) A Release is put into Queue(i,j). Then node i disappears from the tree (see below).

- A node disappears from an Aborting-a tree.
  1)4) i receives an Ack or Down, Parent(i)=nil, and i calls procedure Release. Before this event i was unacked (2.3). So before the event the Aborting-a Graph was a tree with i as root. Assume j is child of i in this tree. Then link (i,j) is not down (1.5). Then j $\in$ Used(i) (1.7). So i places a Release into Queue(i,j). Then j is Acked (2.2.i). Then all Aborting-a nodes are Acked (2.4).
  5) i is Aborting-a and receives a Release from Queue(Parent(i),i). Then i is Acked (2.2.i). Suppose there is still an unacked Aborting-a node j. Then all Aborting-a nodes form a tree with i as root. But all nodes in this tree are Acked(2.4). Contradiction.
  8) Link (i,j) goes down, and i and j are Aborting-a and Parent(j)=i. Then all nodes in the subtree with j as root obtain a unique new AbortId, and a Down is put into Queue(i,j). Hence the Aborting-a Graph is still a tree. □

4.3 A node never participates twice in an Aborting Group

Lemma 3.1 j $\in$ UpLinks(i) $\Longrightarrow$LinkState((i,j))=Up or Down $\in$ Queue(j,i).

Proof:
Holds after initialization.
The dangerous actions are:

- j is put into UpLinks.
  2) i receives an Up from Queue(j,i). If LinkState((i,j))=Down, then Down $\in$ Queue(j,i) (1.2).

- LinkState((i,j)):=Down.
  8) Link (i,j) goes down. Then a Down is put into Queue(j,i).

- Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then j is removed from UpLinks(i). □

Lemma 3.2 πqueue$_i$(j)$\neq\emptyset$ and Down $\notin$ Queue(j,i) $\Longrightarrow$LinkState((i,j))=Up.

Proof:
The dangerous actions are:

- A π-messageis put into πqueue$_i$(j).
  6) i is Aborting and receives a π-message from Queue (j,i). Then LinkState((i,j))=Up (1.3).

- Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then πqueue$_i$(j):= $\emptyset$.

- LinkState((i,j)):=Down.
  8) Link(i,j) goes down. Then a Down is put into Queue(j,i). □

Lemma 3.3 j $\in$ Used(i) and a Down $\notin$ Queue(j,i) $\Longrightarrow$LinkState((i,j))=Up.

Proof:
The dangerous actions are:

- j is put into Used(i).
  1)4)5) i is Aborting, calls procedure Release, becomes Ready and πqueue$_i$(j)$\neq\emptyset$. If Down $\notin$ Queue(j,i) then LinkState((i,j))=Up (3.2).
  6) i receives a π-message from Queue(j,i). Then LinkState((i,j))=Up (1.3).
  7) i sends a π-message to j and j $\in$ UpLinks(i). Then LinkState((i,j))=Up or Down $\in$ Queue(j,i) (3.1).

- Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then j is removed from Used(i).

- LinkState((i,j):=Down.
  8) Link (i,j) goes down. A Down is put into Queue(j,i). □

Lemma 3.4 Down $\in$ Queue(i,j) $\Longrightarrow$Down $\in$ Queue(j,i) or
(j $\notin$ Used(i) and πqueue$_i$(j)=$\emptyset$).

Proof:
The dangerous actions are:

- A Down is put into Queue(i,j).
  8) Link (i,j) goes down. Then also a Down is put into Queue(j,i).

- A Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then j is removed from Used(i) and $\pi queue_i(j) := \emptyset$.

- j is put into Used(i).
  1)4)5) i is Aborting, calls procedure Release, and $\pi queue_i(j) \neq \emptyset$. If Down $\in$ Queue(i,j) then Link (i,j) is down (1.1). So Down $\in$ Queue(j,i) (3.2).
  6) i is Ready and receives a π-message from Queue(j,i). Then Down $\notin$ Queue(i,j) (1.3, 1.1).
  7) i is Ready and sends a π-message to j. Then j $\in$ UpLinks(i). So if Down $\in$ Queue(i,j), then Down $\in$ Queue(j,i) (1.1, 3.1).

- Message is put into $\pi queue_i(j)$.
  6) i is Aborting and receives a π-message from Queue(j,i). Then link (i,j) is Up (1.3) and Down $\notin$ Queue(i,j) (1.1). □

Lemma 3.5 Abort $\in$ Queue(i,j) $\Longrightarrow$no π-message behind the Abort in Queue(i,j).

Proof:
The only dangerous action is:

- π-message is put into Queue(i,j).
  If Abort $\in$ Queue(i,j) then State(i)=Aborting (2.2.a). Then i cannot place a π-message into Queue(i,j). □

The next lemmas describe the relation between π-messages on links and in πqueues, and the state of the node that sent the π-message.

Lemma 3.6 State(i)=Ready $\Longrightarrow \pi queue_i(j) = \emptyset$.

Proof:
The dangerous actions are:

- State(i):=Ready.
  1)4)5) i is Aborting and calls procedure Release. Then $\pi queue_i(j) := \emptyset$.

- Message is put into $\pi queue_i(j)$.
  6) This is only possible if State(i)=Aborting. □

Lemma 3.7 π-message $\in$ Queue(j,i) $\Longrightarrow$
(State(j)=Aborting-a and Abort-a behind the π-message)
or (State(j)=Ready and i $\in$ Used(j)).

Proof:
The dangerous actions are:

- π-message is put into Queue(j,i).
  6) j is Ready and receives a π-message and sends one to i. Then i is put into Used(j).
  7) j is Ready and receives a Start and sends a π-message to i. Then i is put into Used(j).

- State(j):=Ready.
  This is not possible if there is an Abort-a in Queue(j,i), for then AckPend$_j$(i)= true (2.2.a), and there is no Ack in Queue(i,j) (2.2.b), and the first message in Queue(j,i) is not a Down, and there is no Release on the Parent link of j (2.2.i).

- AbortId(j):=b, $b \neq a$.
  8) j is Aborting, AbortId(j)=a, some link goes down and AbortId(j):=b. Then also the id of the Abort message in Queue(j,i) becomes b.

- State(j):=Aborting.
  1)2)3) j is Ready and receives a Down-a, Up-a, or Abort-a. i $\in$ Used(j) so j sends an Abort-a to i. And link (i,j) is up for Queue(j,i) contains a π-message. Moreover, j did not receive a Down from Queue(i,j), which would have caused i to be taken out of Used(j).

- i out of Used(j).
  1) j receives a Down from Queue(i,j). But Down $\notin$ Queue(i,j) (1.3, 1.1).
  1)2) j is Ready, receives an Up or Down and calls procedure Release. This is not possible if Used(j) $\notin \emptyset$. □

Lemma 3.8 πqueue$_i$(j)$\neq\emptyset$ and Down $\notin$ Queue(j,i) $\Longrightarrow$
(State(j)=Aborting-a and Abort-a $\in$ Queue(j,i))
or (State(j)=Ready and i $\in$ Used(j)) .

Proof:
The dangerous actions are:

- π-message is put into πqueue$_i$(j).
  6) i is Ready and receives a π-message from Queue(j,i). Then the conclusion holds (3.7).

- A Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then πqueue$_i$(j):= $\emptyset$.

- State(j):=Ready.
  If Abort $\in$ Queue(j,i), then AckPend$_j$(i)= true (2.2.a).
  Then Release $\notin$ Queue(Parent(j),j) (2.2.i, 2.1) and Ack $\notin$ Queue(i,j) (2.2.b), and Down $\notin$ Queue(i,j) (3.4). So no event is possible in which this action occurs.

- an Abort disappears from Queue(j,i).
  8) Link (j,i) goes down. Then a Down is put into Queue(j,i).
  3) i reads an Abort from Queue(j,i). Then πqueue$_i$(j):= $\emptyset$.

- State(j):=Aborting-a.
  1)2)3) j is Ready and receives a Down-a, Up-a, or Abort-a. Then State(j):=Aborting-a. If Down $\notin$ Queue(i,j) and πqueue$_i$(j)$\neq\emptyset$ then link (j,i) is up (3.2). Hence i $\in$ Used(j) and j sends an Abort-a to i.

- i is removed from Used(j).
  1) j is Ready and receives a Down from Queue(i,j). But if πqueue$_i$(j)$\neq\emptyset$ and Down $\notin$ Queue(j,i) then Down $\notin$ Queue(i,j) (3.4).

1)2) j is Ready, receives an Up or Down and calls procedure Release. This is not possible if Used(j) $\not\in \emptyset$. □

Lemma 3.9 State(i)=Ready and j $\in$ Used(i) and Down $\not\in$ Queue(j,i) $\Longrightarrow$
π-message $\in$ Queue(i,j)
or π-message $\in \pi\text{queue}_j(i)$
or (State(j)=Aborting-a and Abort-a $\in$ Queue(j,i))
or (State(j)=Ready and i $\in$ Used(j)).

Proof:
The dangerous actions are:

- State(i):= Ready and j is put into Used(i).
  1)4)5) i is Aborting, receives a Down, Ack, or Release and calls procedure Release while $\pi\text{queue}_i(j)\neq\emptyset$. Then the conclusion holds because of Lemma 3.8.

- j is put into Used(i).
  6) i is Ready and receives a π-message from Queue(j,i). Then the conclusion holds because of Lemma 3.7.
  6)7) i is Ready, receives a π-message or Start, and sends a π-message to j. This π-message is placed into Queue(i,j), for link (i,j) is up (3.3).

- Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then j is removed from Used(i).

- A message disappears because link (i,j) goes down.
  8) Then a Down is put into Queue(j,i).

- A π-message is read from Queue(i,j).
  6) j receives a π-message from Queue(i,j). If j is Ready then i is put into Used(j). If j is Aborting then the π-message is put into $\pi\text{queue}_j(i)$.

- A π-message disappears from $\pi\text{queue}_j(i)$.
  1) j receives a Down from Queue(i,j). But if j $\in$ Used(i) and Down $\not\in$ Queue(j,i), then Down $\not\in$ Queue(i,j) (1.1, 3.3).
  3) j receives an Abort from Queue(i,j). Then i is Aborting (2.2.a).
  1)4)5) j is Aborting, receives a Down, Ack, or Release, and calls procedure Release, while $\pi\text{queue}_j(i)\neq\emptyset$. Then j becomes Ready and i is put into Used(j).

- State(j):=Ready while Abort-a $\in$ Queue(j,i).
  If Abort $\in$ Queue(j,i), then $\text{AckPend}_j(i)$=true (2.2.a), and Ack $\not\in$ Queue(i,j) (2.2.b). Since i is unacked (2.1), Release $\not\in$ Queue(Parent(i),i) (2.2.i). So no event is possible that could make j Ready while Abort-a $\in$ Queue(j,i).

- AbortId(j):=b, $b \neq a$.
  8) j is Aborting-a, some link goes down and AbortId(j):=b. Then also the id of the Abort message in Queue(j,i) becomes b.

- An Abort read from Queue(j,i).
  3) i is Ready and receives an Abort from Queue(j,i). Then State(i):=Aborting.

- State(j):=Aborting-a.
  1)2)3) j is Ready and receives a Down-a, Up-a or Abort-a, and i $\in$ Used(j). Then State(j):=Aborting-a and j sends an Abort-a message to i. If j $\in$ Used(i) and Down $\notin$ Queue(i,j) then link (i,j) is up (3.3).

- i is removed from Used(j).
  1) i receives a Down from Queue(i,j). But Down $\notin$ Queue(i,j) (1.1, 3.3). □

We still need some extra lemmas to be able to prove that a node never participates twice in the same Abort Group. The next lemma says that if a Release message is on its way to an Aborting-a node on its parent link, all Aborting-a nodes are Acked.

Lemma 3.10 Release $\in$ Queue(i,j) and State(j)=Aborting-a and Parent(j)=i $\Longrightarrow$ $\forall$k(AbortId(k)=a $\Longrightarrow$Acked(k)).

Proof:
Suppose $\exists$k: AbortId(k)=a and not Acked(k). Then the Aborting-a Graph is a tree (Thm 2) with j as root. But j is Acked (2.2.i), and so all nodes in the Aborting-a Graph are Acked (2.4). Contradiction. So all Aborting-a nodes are Acked. □

Lemma 3.11
State(i)=Aborting and AbortId(i)=a and AckPend$_i$(j)= true
and Down $\notin$ Queue(j,i) $\Longrightarrow$
Ack $\in$ Queue(j,i)
or State(j)=Aborting and Parent(j)=i
or Abort-a $\in$ Queue(i,j)
and (π-message $\in$ Queue(i,j) before the Abort-a
or π-message $\in$ πqueue$_j$(i)
or State(j)=Aborting and Abort $\in$ Queue(j,i)
or State(j)=Aborting and Ack $\in$ Queue(i,j) behind the Abort-a
or State(j)=Ready and i $\in$ Used(j).
)

Proof:
If AckPend$_i$(j)=true and Down $\notin$ Queue(j,i), then link (i,j) is Up (2.1, 3.3). Then Down $\notin$ Queue(i,j) (1.1).
The dangerous actions are:

- State(i):=Aborting-a and AckPend$_i$(j):= true.
  1)2)3) i is Ready and receives a Down-a, Up-a, or Abort-a, while j $\in$ Used(i). Then the conclusion holds because of 3.9.

- AbortId(i):=a, $a \neq b$.
  8) State(i)=Aborting-b and some link of i goes down Then AbortId(i):=a and ids of Abort messages on links (i,k) become a too.

- Down disappears from Queue(j,i).
  1) i receives a Down from Queue(j,i). Then AckPend$_i$(j):= false.

- A message disappears from Queue(i,j) or Queue(j,i) because link (i,j) goes down.
  8) Then a Down is put into Queue(j,i).

- An Ack disappears from Queue(j,i).
  4) i receives an Ack from Queue(j,i). Then $AckPend_i(j)$:= false.

- State(j):=Ready.
  If the premise holds, then Down $\notin$ Queue(i,j), and not Acked(i) so Release $\notin$ Queue(i,j) (3.10), so no event can make j Ready.

- An Abort-a disappears from Queue(i,j).
  3) j receives an Abort-a from Queue(i,j). If j is Ready then j becomes Aborting-a and Parent(j):=i. If j is Aborting then j sends an Ack to i, and link (j,i) is up if the premise holds.

- A π-message disappears from Queue(i,j).
  6) j receives a π-message from Queue(i,j). If State(j)=Ready, then i is put into Used(j). If State(j)=Aborting, then the π-message is put into $\pi queue_j(i)$.

- A π-message disappears from $\pi queue_j(i)$.
  1) j receives a Down from Queue(i,j). But Down $\notin$ Queue(i,j) (3.4).
  3) j is Aborting and receives an Abort from Queue(i,j). Then j sends an Ack to i. And Link (i,j) is up if the premise holds.

- State(j):=Ready while Abort-$a \in$ Queue(i,j).
  If Abort $\in$ Queue(j,i) then $AckPend_j(i)$= true (2.2.a) and Ack $\notin$ Queue(i,j) (2.2.b), and Release $\notin$ Queue(Parent(j),j) (2.2.i). If the premise holds then Down $\notin$ Queue(i,j), so no event can make j Ready while Abort-$a \in$ Queue(i,j).

- An Abort disappears from Queue(j,i).
  3) i receives an Abort from Queue(j,i). i is Aborting so i places an Ack behind the Abort-a in Queue(i,j).

- (j Aborting and an Ack behind the Abort-a in Queue(i,j)) becomes false.
  No event can make j Ready while Abort-$a \in$ Queue(i,j) (see above). The Ack cannot disappear from Queue(i,j) either as long as the Abort-a is in the queue.

- State(j):=Aborting.
  1)2)3) j is Ready and receives a Down, Up or Abort (Down $\notin$ Queue(i,j) if the premise holds), and i $\in$ Used(j). Then State(j):=Aborting and j places an Abort into Queue(j,i).

- i is removed from Used(j).
  This is not possible if Down $\notin$ Queue(i,j). □

Lemma 3.12 Down-$a \in$ Queue(i,j) and State(j)=Aborting-$a \Longrightarrow$
the Aborting-a Graph is a tree with j as root.

Proof:
The dangerous actions are:

- A Down-a is put into Queue(i,j) and State(j):=Aborting-a
  8) Link (i,j) goes down and State(j)=Aborting-b and Parent(j)=i. The Aborting-b Graph is a forest (Th.1). So the subgraph below j is a tree with j as root.

- A link disappears from the tree.
  8) Link (k,l) goes down and Parent(l)=k. Then all nodes in the subtree with l as root get a new AbortId, and hence the remaining Aborting-a Graph is still a tree.

- A new Aborting-a node is created.
  3) l is Ready and receives an Abort-a from k. Then Down $\notin$ Queue(k,l), and k is Aborting-a and unacked (2.2.a). Moreover, Release $\notin$ Queue(k,l) (2.2.c). So a node and a link are added to the tree.

- A node disappears from the tree.
  This is not possible because all nodes l in the tree have Parent(l) $\neq$nil, and if (k,l) is a link of the tree and Parent(l)=k, then Release $\notin$ Queue(k,l) (if Release $\in$ Queue(k,l) and Parent(l)=k, then (k,l) is not a link of the tree). □

If there is an unacked Aborting-a node, there is no node with a in its AbortSet:

Lemma 3.13 $\exists$i(State(i)=Aborting-a and not Acked(i)) $\Longrightarrow$
$\forall$j ($a \notin$ AbortSet(j)).

Proof:
The dangerous actions are:

- A first (unacked) Aborting-a node is created.
  1)2) i is Ready and receives Down-a or Up-a. If Down-a $\in$ Queue(k,i) or Up-a $\in$ Queue(k,i) and i is Ready, then a is unique (2.5).

- a is put into AbortSet(j).
  1)2) j is Ready, receives a Down-a or Up-a and Used(j)=$\emptyset$. If Down-a $\in$ Queue(k,j) or Up-a $\in$ Queue(k,j) and j is Ready, then a is unique (2.5).
  3) j is Ready and receives an Abort-a from Queue(k,j). Then k is Aborting-a and AckPend_k(j)=true (2.2.a). And there is no π-message in Queue(k,j) (3.5). Moreover, πqueue_j(k)= $\emptyset$ (3.6), and Ack $\notin$ Queue(j,k) (2.2.b). Then it follows with Lemma 3.11 that k $\in$ Used(j). So Used(j) $\neq\emptyset$, and j does not call procedure Release.
  1)4) j is Aborting and Parent(j)=nil, j receives a Down or Ack, and j calls procedure Release. Then j was unacked before the event (2.3), so the Aborting-a Graph was a tree (Th 2) with j as root. After the event j is Acked, and with 2.4 the whole Aborting-a tree is Acked. Contradiction.
  1) j is Aborting-a and parent(j)=k and j receives a Down-a from Queue(k,j). Then the Aborting-a Graph is a tree with j as root (3.12). If j becomes Acked after the event, all Aborting-a nodes are acked (2.4). Contradiction.
  5) j is Aborting-a and receives a Release from Queue(k,j). Then all Aborting-a nodes are Acked(2.4). □

There is no Abort-a is on its way to a node with a in its AbortSet:

Lemma 3.14 $a \in$ AbortSet(j) $\Longrightarrow$Abort-$a \notin$ Queue(i,j).

Proof:
Suppose Abort-a $\in$ Queue(i,j). Then i is Aborting-a and not Acked (2.2.a), so $a \notin$ AbortSet(j) (3.13). □

Lemma 3.15 State(i)=Aborting $\Longrightarrow$AbortId(i)$\neq$Undefined.

Proof:
The dangerous actions are:

- State(i):=Aborting
  1)2)3) i is Ready and receives an Up-a, Down-a, or Abort-a. Then AbortId(i):=a.
- AbortId(i):=Undefined.
  1)4)5) i is Aborting, receives a Down, Ack, or Release, and calls procedure Release. Then State(i):=Ready. □

The AbortId of a node i differs from all ids in AbortSet(i).

Theorem 3 $\forall x \in$ AbortSet(i)(x $\neq$AbortId(i)).

Proof:
The dangerous actions are:

- A value is assigned to AbortId(i).
  1)2) i is Ready and receives an Up-a or Down-a. Then a is unique (2.5).
  3) i is Ready and receives an Abort-a. Then $a \notin$ AbortSet(i) (3.14).
  1)4)5) i is Aborting, receives a Down, Ack, or Release, and calls procedure Release. Then AbortId(i):=Undefined, and Undefined $\notin$ AbortSet(i) (3.15).
- A value is added to AbortSet(i) without assignment of a value to AbortId(i).
  1)2) i is Ready, receives an Up-a or Down-a, while Used(i)=$\emptyset$. But $a \neq$Undefined.
  □

4.4 The number of events of a node in one Abort Group is finite

We assume that the number of nodes in the network is finite. We have now proved that each node participates at most once in each Aborting Group. After the last topological change there is a finite number of Aborting Groups, because the number of topological changes is finite. We prove that the number of reset events occurring in a node while it participates in an Aborting Group is finite. Reset events are events 1), 2), 3), 4), and 5).

Events 1) and 2) occur only finitely many times, for there are finitely many Up and Down messages in the network. A node is added no more than once to an Aborting Group with AbortId a. In this and in no other event it sends Abort-a messages (a finite number). So a finite number of Abort-a messages is produced, and only a finite number of occurrences of event 3) is possible. We assume that the network is finite. Then Aborting Groups are finite too. Also a node leaves an Aborting Group only once, and this is the only event while the node belongs to the Aborting Group in which it sends Release messages (a finite number). So a finite number of Release messages is produced.

To prove that event 4) (Reception of an Ack) happens only finitely many times, we need some lemmas:

Lemma 4.1 State(i)=Ready $\Longrightarrow$AckPend$_i$(j)= false.

Proof:
Holds after initialization.
The dangerous actions are:

- State(i):=Ready.
  1)4)5) i is Aborting and receives an Abort, Ack, or Release and calls procedure Release. Then $AckPend_i(j)$= false for all j $\in$ Used (i). And $AckPend_i(j)$= true $\Longrightarrow$j $\in$ Used(i) (2.1). So $AckPend_i(j)$= false for all j.

- $AckPend_i(j)$:= true.
  1)2)3) i is Ready and receives a Down, Up, or Abort. Then State(i):=Aborting. □

Lemma 4.2 Count(i)= #{ j | $AckPend_i(j)$}.

Proof:
Holds after initialization.
The dangerous actions are:

- Value of Count(i) changes.
  1) i is Aborting and $AckPend_i(j)$= true and i receives a Down from Queue(j,i). Then $AckPend_i(j)$:= false and Count(i):=Count(i)-1.
  4) i receives an Ack from Queue(j,i). Then i is Aborting and $AckPend_i(j)$= true (2.2.f). $AckPend_i(j)$:= false and Count(i):=Count(i)-1.
  1)2)3) i is Ready and receives a Down, Up or Abort. Then Count(i):= #{j | j $\in$ Used(i)} and j $\in$ Used(i) $\Longleftrightarrow$$AckPend_i(j)$:= true, and $AckPend_i(j)$= false if i is Ready (4.1).

- $AckPend_i(j)$:=false. This case is already treated.

- $AckPend_i(j)$:=true. This case is already treated. □

Lemma 4.3 Event 4) occurs only finitely many times in a node participating in an Aborting Group.

Proof:
When a node i enters an Aborting Group, Count(i) is set to some finite integer value $\geq$ 0. Every time event 4) happens, Count(i) is decreased by one. Count(i) cannot become negative (4.2). □

Theorem 4 The number of reset events of a node participating in an Aborting Group is finite.

Proof:
All reset events of a node while participating in an Aborting Group occur only finitely many times (4.3). □

4.5 The Reset algorithm does not cause deadlock

Lemma 5.1 If all Aborting-a nodes are Acked, a Down or Release message is on its way to each root of an Aborting tree.

Proof:
The Aborting-a nodes form a forest (Th. 1). A root is a node i with Parent(i)=nil, or Parent(i)=j and a Down or Release in Queue(j,i).There are no Acked nodes i with Parent(i)=nil (2.3). So for every root i there is a j such that a Down or Release $\in$ Queue(j,i). □

If node i is Aborting-a and $AckPend_i(j)$=true there is an Abort in Queue(i,j) or an Ack or a Down in Queue(j,i), or there are nodes k and l in the subtree of the Aborting-a tree with i as root that have these properties.

Lemma 5.2 State(i)=Aborting-a and $AckPend_i(j)$=true $\Longrightarrow$
$\exists$k,l(k in Aborting-a subtree with root i and $AckPend_k(l)$= true
and (Abort-a $\in$ Queue(k,l)
or Ack $\in$ Queue(l,k)
or Down $\in$ Queue(l,k))).

Proof:
The dangerous actions are:

- i becomes Aborting-a and $AckPend_i(j)$:=true.
  1)2)3) i is Ready and receives a Down-a, Up-a or Abort-a. $AckPend_i(j)$:=true iff i sends an Abort-a to j. If link (i,j) is down then Down $\in$ Queue(j,i) (2.1, 3.3).
  8) i is Aborting-b and becomes Aborting-a because some link goes down. If an Abort-b message was present in Queue(i,j), its id becomes a too.

- An Abort-a disappears from Queue(k,l).
  8) Link (k,l) goes down. Then a Down is put into Queue(l,k).
  3) l reads an Abort-a.
  If l is Aborting it places an Ack into Queue(l,k) and $AckPend_l(k)$:= true. If l is Ready then π-message $\notin$ Queue(k,l) (3.5) and $\pi queue_l(k)=\emptyset$ (3.6). So k $\in$ Used(l) (3.11). So l sends an Abort-a to k and $AckPend_l(k)$:=true.

- Down or Ack disappears from Queue(l,k).
  8) Link (k,l) goes down and the Ack disappears. Then a Down is put into Queue(l,k).
  1)4) k receives a Down or Ack from Queue(l,k). Then $AckPend_k(l)$:=false.
  If k is Acked after this action it sends an Ack to its parent m. Then $AckPend_m(k)$= true (2.2.f). Thus the lemma holds for m and k, or if i=k the premise does not hold anymore.
  If k is not acked there is an n such that $AckPend_k(n)$=true. Then the lemma holds for k and n. □

Theorem 5 During execution of the reset algorithm no deadlock occurs.

Proof:
As long as a link queue contains a message an event is possible, because reception of a

message by a node is unconditional. If there is still an Aborting node present in the network, there is also a message present in a link queue(4.1, 4.2, 5.1). □

To prove termination of the reset Algorithm we must assume that each message that is sent and not discarded is received within finite time. Then the finite number of reset messages, produced after the last topological change (Th.4), is received within finite time, since no deadlock can occur (Th.5).

4.6 Correctness

Here we have to prove that after termination of the reset algorithm there are no more obsolete nodes or obsolete π-messages in the system.

Lemma 6.1 (M $\in$ Queue(i,j) or M $\in$ πqueue$_j$(i)) and M is an Obsolete π-message $\Longrightarrow$
πState(i) =Obsolete
or Abort $\in$ Queue(i,j) and (M $\in$ Queue(i,j) $\Longrightarrow$Abort behind M)
or Down $\in$ Queue(i,j).

Proof:
The dangerous actions are:

- obsolete π-message is put into Queue(i,j).
  7) i is obsolete π and receives a π-message, or i receives an obsolete π-message and becomes obsolete-π and i places an obsolete π-message into Queue(i,j).

- A π-message in Queue(i,j) becomes obsolete.
  8)9) A link comes up or down and causes i to become obsolete.

- i is Ready, πState(i)=Obsolete and i becomes Aborting.
  1)2)3) i is Ready and Obsolete and receives a Down, Up or Abort. If Down $\in$ Queue(j,i), then Down $\in$ Queue(i,j) (1.1, 2.1, 3.3). If a π-message $\in$ Queue(i,j) or a π-message $\in$ πqueue$_j$(i) and Down $\notin$ Queue(i,j), then j $\in$ Used(i) (3.7, 3.8). So i places an Abort into Queue(i,j).

- An Abort is removed from Queue(i,j).
  8) link (i,j) goes down. Then a Down is placed into Queue(i,j).
  3) j receives an Abort from i. Then M $\notin$ Queue(i,j) and πqueue$_j$(i):= $\emptyset$.

- Down is removed from Queue(i,j).
  1) j receives a Down from Queue(i,j). Then πqueue$_j$(i):=$\emptyset$, and there is no message in Queue(i,j). □

Lemma 6.2 Link (i,j) is π-marked $\Longrightarrow$i $\in$ Used(j) and j $\in$ Used(i).

Proof:
The dangerous actions are:

- Link (i,j) becomes π-marked.
  6) i is Ready and receives a π-message from Queue(j,i). Then i $\in$ Used(j) (3.7), and j is put into Used(i).
  1)4)5) i is Aborting, receives a Down, Ack, or Release, and calls procedure Release. If $\pi queue_i(j) \neq \emptyset$, then i $\in$ Used(j) (3.8, 2.2.a, 2.1), and j is put into Used(i).

- i is put into Used(j). This case is already treated.

- i is removed from Used(j). This is not possible if i is Ready and Down $\notin$ Queue(j,i). □

Lemma 6.3 Every connected group of obsolete π nodes contains a node i such that Abort, Down or Up $\in$ Queue(j,i) for some j.

Proof:
The dangerous actions are:

- A new connected group of obsolete π nodes comes into existence.
  8)9) A link (i,j) adjacent to a Ready π node j changes its status. Then an Up or Down is placed into Queue(i,j).

- A node i is removed from a connected group of obsolete π nodes.
  1)2)3) i is Ready, πState(i) =Obsolete, i receives a Down, Up or Abort, and becomes Aborting. If link (i,j) was π-marked, then j $\in$ Used(i) (6.2), so i sends Aborts over all π-marked links (i,j).

- A link is removed from the connected group of π nodes.
  8) Link (i,j) goes down. Then a Down is placed into Queue(i,j) and Queue(j,i).

- An Abort, Up or Down message disappears.
  1)2)3)8) These cases are already treated. □

Theorem 6 After the termination of the reset algorithm there are no more obsolete π nodes or obsolete π-messages present.

Proof:
If there is an obsolete π-node or an obsolete π-message, there is also a message of the reset algorithm or an Up or Down message (6.1, 6.3). So the reset algorithm has not terminated yet. □

4.7 Alternative model for links

We assumed that if a link went down, all messages sent but not yet received were lost. Another possible assumption is that messages sent before the link went down are still available and only the messages sent when the link was down are lost. This gives a slightly different version of event 8:

```
if LinkState((i,j))=Up
then LinkState((i,j)):=Down;
     fetch new unique ids a and b;
     RenameIds(a,i,j);
     RenameIds(b,j,i);
     MakeObsolete(i);
     MakeObsolete(j);
     enqueue(Down, Queue(i,j));
     enqueue(Down, Queue(j,i))
fi
```

The actions of the Global Observer must also be adapted. Procedure RenameIds becomes:

```
Procedure RenameIds(a,i,j):
  if Abort ∈ Queue(i,j)
       and Down ∉ Queue(i,j)
  then the id of this Abort becomes a
  fi;
  if State(j)=Aborting
       and Parent(j)=i
       and Down ∉ Queue(i,j)
       and Release ∉ Queue(i,j)
  then AbortId(j):= a;
       for all nodes k adjacent to j
       do RenameIds(a,j,k)
       od
  fi;
```

This assumption alters the proof slightly. We can no longer assume that no messages are received if a link is down.
Messages do not disappear any more because a link goes down. If a message is placed into Queue(i,j), link(i,j) is up. If Queue(i,j) contains a Down, this is the last message of the queue. After j reads this Down, Queue(i,j) is empty until the link comes up again. Where in the old proof we used "if a linkqueue contains a message other than Up or Down the link is not down" we now use "if a link contains a message and is down, it also contains a Down behind the message", which we already formulated as lemma 1.2.

Some lemmas must be slightly modified:
Lemma 1.3 becomes:

Lemma 1.3 M ∈ Queue(i,j) and Down ∉ Queue(i,j) ⟹
LinkState((i,j))=Up.

And to the proof of Lemma 1.3 is added:

- Down is removed from Queue(i,j).
  1) j receives a Down from Queue(i,j). Then Queue(i,j):=$\emptyset$(1.2).

In lemma 1.6:
"or Release $\in$ Queue(i,j)" becomes "or Release or Down $\in$ Queue(i,j)".

For the proof of Theorem 1 we need a new definition and a new lemma:

Definition Id a **is unique for Queue(i,j)** if there are no nodes with a as AbortId, and there are no messages in other queues than Queue(i,j) with id a.

Lemma 1.7 Abort-a $\in$ Queue(i,j) and a is unique for Queue(i,j) $\Longrightarrow$
Down-a $\in$ Queue(i,j).

Proof:
The dangerous actions are:

- An Abort-a is put into Queue(i,j).
  1)2)3) i is Ready and receives an Up-a, Down-a, or Abort-a. Then i becomes Aborting-a and a is not unique for Queue(i,j).
  8) Link (i,j) goes down, Abort-b $\in$ Queue(i,j), the id of this Abort becomes a, and a does not spread further. Then a Down-a is placed into Queue(i,j).
  8) If another link goes down and Abort-b:= Abort-$a$in Queue(i,j) then a is not unique.
- A Down-a is removed from Queue(i,j).
  This is not possible while Abort-a $\in$ Queue(i,j). □

Changes to the proof of Theorem 1 are:

- A new Aborting-a Graph comes into existence.
  1)2) .....
  3) Node j is Ready and receives an Abort-a from Queue(i,j) and a is a new unique id. Then Parent(j):=i, but Down $\in$ Queue(i,j), so link (i,j) does not belong to the graph.
  .....

Lemma 2.2.a becomes:

Lemma 2.2.a Abort-a $\in$ Queue(i,j) $\Longrightarrow$
(State(i)=Aborting and AbortId(i)=a and AckPend$_i$(j)=true)
or Down $\in$ Queue(i,j).

To prove lemma 2.5 we now need an extra lemma:

Lemma 2.5* Down-a $\in$ Queue(i,j) and Release $\in$ Queue(i,j) $\Longrightarrow$
a is unique for Queue(i,j).

Proof: The dangerous actions are:

- Down-a is put into Queue(i,j).
  8) Link (i,j) goes down. If Release $\in$ Queue(i,j), then a does not spread out of Queue(i,j).

- a becomes not unique for Queue(i,j).
  1) j is Ready and receives Down-a from Queue(i,j). Then Queue(i,j) := $\emptyset$. (1.2)
  3) j is Ready and receives an Abort-a from Queue(i,j). This is not possible if Release $\in$ Queue(i,j), for then the Release comes before the Abort (2.2.d). □

Lemma 2.5 becomes:

Lemma 2.5 Down-a $\in$ Queue(i,j) or Up-a $\in$ Queue(i,j) $\Longrightarrow$
a is unique for Queue(i,j)
or Parent(j)=i and AbortId(j)=a.

And its proof is changed to: .....

- a becomes not unique for Queue(i,j).
  1)2) .....
  3) j is Ready and receives an Abort-a from Queue(i,j). Then j becomes Aborting and Parent(j):=i.

- (Parent(j)=i and AbortId(j)=a) becomes false.
  1) .....
  5) j receives Release from Queue(i,j). Then a is unique for Queue(i,j) (2.5*). □

For the proof of Theorem 2 we need an extra lemma:

Lemma 2.6 Abort-a $\in$ Queue(i,j) and Down-x $\in$ Queue(i,j) $\Longrightarrow$x=a.

Proof:
The dangerous actions are:

- An Abort-a is put into Queue(i,j).
  This is not possible if Down $\in$ Queue(i,j), for then link (i,j) is down (1.1).

- Down-x is put into Queue(i,j).
  8) Link (i,j) goes down. Then a Down-x is placed into Queue(i,j), and the Abort message in Queue(i,j) gets the same id as the Down. □

Changes in the proof of Theorem 2 are:

- a first unacked Aborting-a node is generated.
  1)2) .....
  3) i is Ready and receives an Abort-a and a is unique. Then Down $\in$ Queue(j,i) (1.7), so the Aborting-a Graph consists of one node and no links.
  8) .....

- A node becomes Aborting-a.
  .....
  3) i is Ready and receives an Abort-a from Queue(j,i). If a is not unique for Queue(j,i), then Down $\in$ Queue(j,i) (1.7, 2.5, 2.6). So j is Aborting-a and .....

Changes in the proof of Lemma 3.4 are:
.....

- i is Ready and receives a π-message from Queue(j,i).
  If Down $\in$ Queue(i,j), then Down $\in$ Queue(j,i) (1.1, 1.2).

- Message is put into πqueue$_i$(j).
  6) i is Aborting and receives a π-message from Queue(j,i). If Down $\in$ Queue(i,j), then Down $\in$ Queue(j,i) (1.1, 1.2).

Lemma 3.7 is changed to:

Lemma 3.7 π-message$\in$ Queue(j,i) $\Longrightarrow$
(State(j)=Aborting-a and an Abort-a behind the π-message
or State(j)=Ready and i $\in$ Usedj)
or Down $\in$ Queue(j,i).

The proof of lemma 3.7 becomes:

- A π-messageis put into Queue(j,i).
  .....

- State(j):=Ready.
  If a Down $\in$ Queue(i,j), the conclusion still holds. If Down $\notin$ Queue(i,j) and Abort-a $\in$ Queue(i,j), then j cannot become Ready, for then .....

- AbortId(j):=b, $b \neq a$.
  8) j is Aborting-a, some link goes down, and AbortId(j):=b. If Down $\in$ Queue(j,i), then the conclusion still holds. If Down $\notin$ Queue(j,i), then also the id of the Abort message becomes b.

- State(j):=Aborting.
  1)2)3) j is Ready and receives a Dowd-a, Up-a, or Abort-a. i $\in$ Used(j) so i sends an Abort-a to i. If i received a Down from Queue(j,i) Queue(j,i) is now empty (1.2), so the premise does not hold any more. Else if link (i,j) is down then Down $\in$ Queue(j,i) (1.2). Otherwise the Abort-a message is placed into Queue(j,i).

- i is removed from Used(j).
  1) j receives a Down from Queue(i,j). Then Down $\in$ Queue(i,j) (1.1,1.3).
  1)2) ..... □

The proof of lemma 3.12 is changed slightly:

- A new Aborting-a node is created.
  3) l is Ready and receives an Abort-a from Queue(k,l). If Down $\in$ Queue(k,l), then a is unique for Queue(k,l) (2.5,2.6). If Down $\notin$ Queue(k,l), then k is Aborting-a.....

An addition to the proof of Lemma 3.13 is:

- a is put into AbortSet(i).
  .....
  3) i is Ready and receives an Abort-a from Queue(j,i). If Down $\in$ Queue(j,i) then this a is a new unique id (2.6,2.7). If Down $\notin$ Queue(j,i) then j is Aborting-a and AckPend$_j$(i)=true (2.2.a).

Addition to the proof of Lemma 5.2:

- An Abort-a disappears from Queue(i,j).
  ..... So j sends an Abort-a to i and $AckPend_j$(i)=true. If link (i,j) is down then Down $\in$ Queue(j,i) (2.1, 3.3).

With these changes the original proof of termination and correctness of the reset Algorithm is transformed to a proof for this algorithm if we assume that the links behave according to our second model. So also in a network with links behaving according to our second model the reset Algorithm works correctly.

Chapter 5

Conclusions and Discussion

The reset algorithm of [AAG87] is proved correct.
Essential assumptions are:

- The links are FIFO.
- Message transport is error-free.
- If a link status changes, the adjacent nodes are eventually notified of this.
- A link does not come up before both adjacent nodes are notified that the link was down.
- The network contains a finite number of nodes.
- Messages present in a link are received within finite time unless the link goes down.

We used two models for the behavior of links. In the first model all messages that were not yet received are discarded if a link goes down. In the second model only messages sent when the link was down are discarded. The reset algorithm works correctly with both models.

The π-algorithm of [AAG87] is started by one or more nodes that receive a Start signal from the outside. These nodes are then initiator nodes. If their π-computation becomes obsolete and is reset, an initiator node starts the π-computation anew. This way of restarting the π-computation is not needed for a correct operation of the reset algorithm. Other ways are also possible, e.g. reset nodes test if a π-computation is already in progress, and if not, start a leader election.

A node is only allowed to end its execution of the π-algorithm if termination is detected in all nodes of the connected π-group. Otherwise a link connected to another node of this group still executing the π-algorithm could change its state, and so make the computation obsolete.

According to Afek et al. [AAG87] a group of connected π-nodes becomes obsolete if an obsolete π-message is received by a node of the group. This is not the case in the Version Number method [Fin79]. Here an obsolete π-message is recognizable as obsolete by a node that is executing a π-computation with a higher version number and is discarded.This is an advantage of the Version Number method.

An implicit assumption of [AAG87] is that a node may only participate in one π-computation at a time. The reset algorithm is not well-defined if a node should participate in more than one π-computation.

The basis of the reset algorithm of [AAG87] is a succession of PIF (Propagation of Information with Feedback) and PI (Propagation of Information) protocol of [Seg83].

Assertional verification of a distributed algorithm consists of a number of steps:

1. Describing the actions of the network as a system with state variables and events.
2. Finding invariants: predicates over the state variables of the system that are not violated by the events and from which follow the properties we want to prove.
3. Proving that the invariants are indeed invariant.

In step 1 everything that happens in the network and that concerns the algorithm we want to prove correct must be described as a system with components that have state variables and perform events. Components of such a system could be nodes and links. Events happen at one component of the system. Events may only change state variables of another component, if these changes always happen together. An example is an event of a node in which the node sends a message, and that message is placed into a state variable of a link.

The distributed algorithm that is to be proved correct should be transformed to a version with state variables and events. If other actions of the nodes are are also of importance, like the π-algorithm in the proof of the reset algorithm, they should also be translated into events.

In a fault-free network with synchronous message passing messages are sent and received without errors at the same moment, or with a constant delay. Then there is no need to consider links as separate components of the system, because they do not perform any actions. In a network with asynchronous FIFO links these links may have queues as state variables, if links are non-FIFO the state variables could be sets(if all messages are different) or bags (if duplicate messages are allowed).

For the verification of a fault-tolerant distributed algorithm errors that may occur in the network must also be incorporated in the model of the network. Links that are not error-free or lose messages may be modelled by adding link events that transform a message to an error value or discard a message from the link state variable that contains the messages. If links may fail, a state variable is needed that reflects the state of the link: up or down. Failure of the link may be modelled by a link event that changes the state of the link and discards all messages present in link state variables. Here the state variables containing messages correspond to messages present in the link itself. If failure of a link is not modelled as discarding of all messages, the state variables correspond to queues present in the communication software of the network.

Nodes may be notified of a link failure or link restoration by placing a message in the state variables of the link. Node failures could be modelled as events of the nodes.

Events are atomic. They should not be too small, for that makes the proof very lengthy [Lam82], but also not too big, for that makes it difficult to see the results of an event, which also complicates the proof.

It may be necessary for the proof to record part of the history of an execution of the algorithm. We modelled this as a Global Observer, who has its own state variables, and