Synchronous Link-level Protocols

Anneke A. Schoone

RUU–CS–91–2
February 1991

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Synchronous Link-level Protocols

Anneke A. Schoone

Technical Report RUU–CS–91–2
February 1991

Department of Computer Science
University of Utrecht
P.O. Box 80.089, 3508 TB Utrecht
the Netherlands.

ISSN:0924-3275

SYNCHRONOUS LINK-LEVEL PROTOCOLS*

Anneke A. Schoone

Department of Computer Science, University of Utrecht,
P.O.Box 80.089, 3508 TB Utrecht, the Netherlands.

Abstract. Aho et al. gave several protocols for sending a sequence of bits over an error-prone link an a synchronous environment. We investigate how the Krogdahl-Knuth technique of system-wide invariants can be used for proving the protocols correct.

1. Introduction. Aho et al. [AWYU] consider the problem of sending a sequence of bits over an error-prone link in a synchronous environment. They distinguish different classes of possible errors, and try to devise protocols of minimum complexity for the different classes that would ensure the full transmission of a sequence. The protocols that Aho et al. devise are stated in terms of finite-state automata. This is a way of precisely formulating a protocol, and has the advantage of providing a measure for the complexity of the protocol, namely the number of states of the automata. Aho et al. first prove that if all errors are allowed, no correct protocol is possible. Next they consider protocols for the case that the communication links only admit deletion errors. They begin by proving that one-state automata do not suffice for a correct protocol. We feel however that the proof hereof is not completely fair, as connection-management issues are used in a counter example. Halpern and Zuck [HaZu] give a knowledge-based protocol of which they claim that it contains all protocols defined by Aho et al., together with a correctness proof.

The disadvantage of a correctness proof in terms of finite-state automata is that all possible state transitions have to be checked. Considering the protocols of Aho et al., the question arose whether it is possible to give a correctness proof making use of system-wide invariants (and thus by assertional verification). These were introduced as a proof method for protocols in a distributed environment by Krogdahl [Kr] and Knuth [Kn], and strongly advocated by Lamport [La]. However, a main assumption in this proof method is that only one atomic action takes place at a time, while in the protocols proposed by Aho et al. synchronicity is assumed, i.e., operations of sender and receiver take place at the same time. Also, information is derived from things *not* happening at a certain time. Hence, to make the protocols amenable to assertional verification, we lessen the constraint of synchronicity slightly, while retaining the correctness of the protocols. Chandy and Misra [ChMi] use a different approach for an assertional verification of a synchronous program. They lump all atomic actions located at the different processors which are supposed to be executed at the same time together into one large action which is considered to be atomic.

*This research was supported by the ESPRIT Basic Research Actions of the EC under contract No. 3075 (Project ALCOM).

In the next section we give both the model of Aho et al. and the model which we use for the correctness proofs. Section 3 presents the actual protocols in our notation together with the correctness proofs. For one of the protocols only the ideas were sketched in [AWYU]. The (non-trivial) details are shown in section 3.3.

2. The models.

We first give a short synopsis of the model used by Aho et al. (details can be found in [AWYU]). After a short introduction to assertional verification we give the modifications which lead to our model, and state the necessary assumptions.

2.1. The original model.

Aho et al. considered the situation of two processors i and j, connected by unreliable communication links, where a sequence of bits must be transmitted from i to j. Actions to be taken by the processors are described in terms of finite automata. Processors i and j operate in a synchronous fashion. At each time step, they both do one move. A move of the sender i is based upon its current state, the symbol received over the communication link, and the symbol to transmit. It can consist of a change in state, sending a symbol over the link, and possibly advancing its input (reading the next bit of the input). A move of receiver j is based on its current state, and the symbol received over the link. It can consist of a change in state, sending a symbol over the link, and possibly writing a 0 or a 1 on the output (i.e., appending it to the bit sequence that it is accumulating).

The information that can be exchanged over the communication links, in each direction per ''time step'', are the bits 0 and 1, and ''nothing'', which is represented by the symbol λ. The following transmission errors are considered:

- *deletion errors*, in which a 0 or 1 is sent, but λ is received;
- *mutation errors*, in which a 0 or 1 sent is received as a 1 or 0, respectively; and
- *insertion errors*, where a λ sent is received as a 0 or 1.

2.2. Assertional verification.

A very useful concept if one wants to use system-wide invariants is the concept of a *protocol skeleton*. Protocol skeletons are generic descriptions for classes of protocols all of which have some underlying structure in common. A protocol skeleton consists of a number of *operations* which consist of a piece of program. Such an operation is viewed as an atomic action, i.e., it can not be interrupted. We do not specify anything about an assumed order in which the operations may take place, however an operation can contain a so-called *guard*: a boolean expression between braces { }, and the operation may only be executed if the guard is true, otherwise nothing happens. For example, a processor may only execute the code for receiving a message if there is indeed a message present to receive.

The most basic operations in a distributed protocol one can think of for processor i are (assuming (i,j) is a direct link) : send a message to j (S_i), receive a message from j (R_i), and do an internal (local) computation (I_i). Operations and variables are subscripted by the identity of the processor that performs and maintains them, respectively. This yields the following protocol skeleton.

Basic protocol skeleton :

```
$S_i$ :  begin send a message end

$R_i$ :  {a message has arrived}
         begin receive the message;  compute end

$I_i$ :  begin compute end
```

We will only state the corresponding operations S_j, R_j, and I_j for processor j explicitely if there are more differences than the interchanging of the symbols i and j.

We can use these operations as building blocks for bigger operations that we also consider to be atomic, thereby adding extra structure in the order of computation and/or the order of communication. We will call the resulting protocol skeleton a *refined protocol skeleton.*

In *assertional verification* the idea is that if an assertion (a relation between process variables for example) holds initially, and is kept invariant by all possible operations, then it will hold always in the distributed system, whatever order of operation takes place in an actual execution. Such an assertion is called a *system-wide invariant.*

The advantage of the use of protocol skeletons in assertional verification is the following. If one has a refined protocol skeleton or an algorithm which can be viewed as a special instance of a protocol skeleton, then any system-wide invariant which holds for the protocol skeleton, will hold also for the refined protocol skeleton or the algorithm. This is the case simply because the invariant was proven correct for any order of operations in the general case, and hence also for the special order of operations which will take place in the refined protocol skeleton or the algorithm.

System-wide invariants are very well suited to prove the *partial correctness* (or safety) of a protocol skeleton, e.g., that if a value is written in a variable, it is the correct value. In general they are less suited to prove *total correctness* (or liveness), e.g., that all variables are written in finite time. Although the freedom of deadlock can be stated in a system-wide invariant, this does not mean that the freedom of deadlock for a general protocol skeleton automatically carries over to the freedom of deadlock for a refinement of that protocol skeleton. As the order of operation in the latter might be more restricted because extra guards were introduced, freedom of deadlock will correspond to a *different* system-wide invariant. In this paper, we restrict ourselves to the proof of the partial correctness of the protocols of Aho et al.

2.3. Our model. In the model of Aho et al., the non-arrival of a physical bit in a time step was termed: the arrival of the symbol λ. Thus we assume that in our model, messages can have three different contents: 0, 1, and λ. We will denote a message with contents m as $<m>$. Hence we write $<\lambda>$ and mean: a physical message with empty contents. We need a physical message in our model to render the guard of the receive operation true: "a message has arrived".

We model the different classes of errors by operations acting on the "contents" of the communication links. Hence we assume that outside the error operations, messages sent are not lost, garbled, duplicated, or delayed infinitely long. Thus a message sent always arrives in finite time at its destination. We also assume that messages are not reordered: links have the FIFO property, i.e., a message sent first, arrives first. We do not really mean to say that the communication links have those properties, but that all the errors that do occur, can be described by the error operations.

Hence communication is modeled as follows. The link (i,j) is represented by two FIFO queues of messages $Q[i,j]$ and $Q[j,i]$: the messages from i to j and from j to i, respectively. We denote the fact that a message m is on its way from processor i to processor j over the direct link (i,j) as $<m> \in Q[i,j]$. Thus we get the following send and receive procedures.

```
proc send $<m>$ to $j$ by $i$ =
     begin append $<m>$ to $Q[i,j]$ end

proc receive $<m>$ from $j$ by $i$ =
     {$<m>$ is the first message in $Q[j,i]$}
     begin delete $<m>$ from $Q[j,i]$ end
```

The error operations we consider are:

```
$E^d_{ij}$ : { $\exists <x> \in Q[i,j]$ } begin $<x> := <\lambda>$ end                        (deletion)

$E^m_{ij}$ : { $\exists <x> \in Q[i,j]$ with $x \neq \lambda$ }                                 (mutation)
      begin if $x = 0$ then $<x> := <1>$ else $<x> := <0>$ fi end

$E^i_{ij}$ : { $\exists <\lambda> \in Q[i,j]$ } begin choose $x \in \{0,1\}$; $<\lambda> := <x>$ end   (insertion)
```

and the corresponding operations E^d_{ji}, E^m_{ji}, and E^i_{ji} for messages in $Q[j,i]$. With statements of the sort $<x> := <y>$ we mean: change only the contents and not the position or the existence of the message in the queue. Thus we have made the following extra assumption about the system of processors and communication links.

Assumption 2.1. Transmission delays are sufficiently bounded such that the arrival of an empty message (i.e., the non-arrival of a bit) can be inferred.

In synchronous computation it is usually assumed that messages are transmitted with a fixed delay, but assumption 2.2.1 is sufficient to make the transition from the synchronous model where the non-arrival of a message is used as information to an asynchronous model which is driven by the arrival of (possibly empty) messages. The assumption is not necessary if we allow messages with three possible contents, forgetting the original meaning. However, these three different contents cannot be represented by a single bit any more.

In the model of Aho et al., the class of an error is defined by the relation between the symbol sent and the symbol received. In our model an error is an operation upon the contents of a message queue. These different viewpoints lead to a discrepancy for the case of deletion and insertion errors. This is discussed further in section 3.3.

The remaining assumptions in our model are the following.

Assumption 2.2. Messages are not lost, garbled otherwise than in the specified error operations, duplicated, or delayed infinitely long.

Assumption 2.3. Messages are not reordered.

As we want to model a synchronous system, we include one send and one receive action of one processor into one operation. Thus the operation will consist of the work to be done by one processor in "one time step". The operation will be guarded by the arrival of a message.

The assumption that links have the FIFO property (assumption 2.3) is not meaningful in the original synchronous model, but is necessary now. To see this, consider the case that $Q[i,j]$ and $Q[j,i]$ both contain one message, and that processor i receives one message from j. It then also sends a message to j. Hence $Q[i,j]$ now contains two messages. If these messages were reordered, j would receive the second message first. This clearly could never happen in the synchronous model of Aho et al., hence we have to exclude this possibility here.

To start the protocols, we assume that both $Q[i,j]$ and $Q[j,i]$ initially contain one empty message. This is done to remain close to the model of Aho et al., otherwise we would have to introduce a separate starting operation to send the first message. We denote the assertion "there is exactly one message in $Q[i,j]$ which is empty" as $<\lambda> \in !\ Q[i,j]$. Thus we get as a general synchronous protocol skeleton:

Protocol skeleton SP:

```
    Initially   $<\lambda> \in !\ Q[i,j]$ and $<\lambda> \in !\ Q[j,i]$.
    $R_i^{SP}$ : { $Q[j,i] \neq \emptyset$ }
        begin receive $<m>$ from $j$;  compute;  send $<x>$ to $j$ end
    $R_j^{SP}$ : defined similarly, and
    $E_{ij}^d$, $E_{ji}^d$, $E_{ij}^i$, $E_{ji}^i$, $E_{ij}^m$, and $E_{ji}^m$ : as defined above.
```

Given protocol skeleton SP, we are now ready to prove the first invariant.

Lemma 2.1. Using protocol skeleton SP, the total number of messages in $Q[i,j]$ and $Q[j,i]$ is 2.

Proof. Initially this is true. Recall that operations are considered to be atomic actions, hence the assertion only has to reflect the state of the queues after completion of an operation. The error operations do not change the number of messages in the queues, only their contents. Hence these operations do not falsify the assertion. In operation R_i^{SP} the number of messages in $Q[j,i]$ is decreased by one, while the number of messages in $Q[i,j]$ is increased by one. Hence the total number stays the same. For operation R_j^{SP} the same holds with i and j interchanged. Thus the assertion is kept invariant by all possible operations. ■

Let the *system state* be the entity consisting of the set of values of the local variables and the contents of message queues. We now define the concept of a *balanced state*, which corresponds to the states which can occur in the synchronous model of Aho et al.

Definition 2.1. A system state is *balanced* if the number of messages in $Q[i,j]$ is equal to the number of messages in $Q[j,i]$.

Lemma 2.2. Using protocol skeleton SP, the following assertions hold invariantly:

(1) in a balanced state both R_i^{SP} and R_j^{SP} are enabled,

(2) in an unbalanced state one of R_i^{SP} and R_j^{SP} is enabled, the other is disabled,

(3) operation R_i^{SP} transforms a balanced state in an unbalanced one and vice versa, as does operation R_j^{SP},

(4) starting from a balanced state, a sequence of two consecutive R^0-operations can only consist of R_i^{SP} and then R_j^{SP}, or R_j^{SP} and then R_i^{SP}, and in both cases this leads to a balanced state.

Proof. Obvious from the definition and the protocol skeleton. ■

Hence we can restrict ourselves to balanced states and need not worry whether it was R_i^{SP}; R_j^{SP} or R_j^{SP}; R_i^{SP}. (This is the case under the assumption that R_i^{SP} and R_j^{SP} are independent, i.e., the action that i takes does not depend on the value of a local variable of j. This should be the case in any protocol.)

Thus the artificial difference that we created in this model between "R_i^{SP} before R_j^{SP}" and "R_i^{SP} after R_j^{SP}" in contrast with "R_i^{SP} at the same time as R_j^{SP}" in the original model can be overlooked if we confine ourselves to balanced states.

3. Assertional proofs of the protocols of Aho et al.

In this section we specify refined skeletons for the protocols of Aho et al. for different (combinations of) errors. We discuss the protocol for the case of deletion errors only in section 3.1. In section 3.2 we consider deletion and mutation errors, while the combination of deletion and insertion errors is discussed in section 3.3.

First we give some general notation. The sequence of bits that i should transmit to j is in $string_i$, and j outputs it in $string_j$. For sake of notation and easy formulation of invariants, we consider the sequences as arrays which are subscripted by n_i and n_j, respectively, to give the current position. Hence a statement like "advance the input" is encoded as $n_i := n_i + 1$. However, we nowhere make use of any further properties of arrays, and we could as well use a one-way tape with one head.

The states of i and j are recorded in the variables $state_i$ and $state_j$, respectively. For the values of the variable *state* we use numbers, for 2-state automata we use numbers modulo 2, and for 3-state automata we use numbers modulo 3, respectively. As the actions of the protocol skeleton differ for i and j, we will denote their actions as S_i^D and R_j^D. As superscript we use a code for the class of errors allowed.

3.1. Deletion errors only.

The protocol Aho et al. give for the case of deletion errors only can be viewed as a special implementation of the alternating bit protocol, in which the control bit is not sent, but "signaled" by the sending of non-λ symbols at either odd or even time steps. The protocol takes the following form in our model and notation.

Protocol D:

Initially $state_i = state_j = 0,\ n_i = n_j = 1,\ <\lambda> \in\ !\ Q[i,j],\ <\lambda> \in\ !\ Q[j,i].$

```
$S_i^D$ : { $Q[j,i] \neq \varnothing$ }
      begin receive $<x>$ from $j$;
            if    $state_i = 0 \wedge x = \lambda$ then send $<string_i[n_i]>$ to $j$;  $state_i := 1$
            elif  $state_i = 0 \wedge x \neq \lambda$ then send $<\lambda>$ to $j$;  $n_i := n_i + 1$
            else  send $<\lambda>$ to $j$;  $state_i := 0$
            fi
      end

$R_j^D$ : { $Q[i,j] \neq \varnothing$ }
      begin receive $<x>$ from $i$;
            if $x = \lambda$
            then  send $<\lambda>$ to $i$;  $state_j := state_j + 1 \bmod 2$
            else  send <1> to $i$;
                  if $state_j = 1$
                  then $string_j[n_j] := x$;  $n_j := n_j + 1$
                  else $state_j := 1$
                  fi
            fi
      end
```

E_{ij}^d and E_{ji}^d as in section 2.3.

For this protocol we can derive the following invariants.

Lemma 3.1. Using protocol D, the following assertions hold invariantly:

(1) $state_i = 0 \Rightarrow <\lambda>$ tail of $Q[i,j] \vee Q[i,j] = \varnothing$,

(2) $state_j = 0 \Rightarrow <\lambda>$ tail of $Q[j,i] \vee Q[j,i] = \varnothing$,

(3) $<x>$ tail of $Q[i,j] \wedge x \neq \lambda \Rightarrow x = string_i[n_i] \wedge state_i = 1$,

(4) $<x>$ tail of $Q[j,i] \wedge x \neq \lambda \Rightarrow state_j = 1$.

Proof. (1). Initially $state_i = 0$ and $<\lambda> \in Q[i,j]$ is the last (and only) message, hence the assertion holds. Operation E_{ij}^d cannot change the content of an empty message. Operation R_j^D receives a message and deletes it from $Q[i,j]$. If it was the last one, then $Q[i,j] = \varnothing$ now holds; if it was not, then $<\lambda>$ remains the last message in $Q[i,j]$. Operation S_i^D can send a message such that the last message in $Q[i,j]$ does not contain λ any more; however, then $state_i$ is changed to 1. If S_i^D renders the premise true by setting $state_i = 0$, then i sends $<\lambda>$ to j. Hence the assertion is kept invariant by all possible operations.

(2). Initially the assertion holds. Likewise, in R_j^D, if $state_j$ is set to 0, $<\lambda>$ is sent to i. If j sends a non-λ symbol, $state_j$ is set to or remains 1. Operation S_i^D does not affect the last message of $Q[j,i]$ unless it empties the queue.

(3). Initially the premise is false, hence the assertion holds. In operation S_i^D, if i sends a non-λ message $<x>$, then $x = string_i[n_i]$ and i sets $state_i$ to 1. When n_i is increased, another message is sent to j and $<x>$ is no longer the last message. Operation E_{ij}^d on the last message in $Q[i,j]$ falsifies the premise, as does operation R_j^D if the last message from $Q[i,j]$ is received.

(4). Initially the premise is false. If in operation R_j^D <1> is sent to i, $state_j$ becomes or remains 1. If in R_j^D $state_j$ is set to 0, $<\lambda>$ is sent, hence <1> is no longer the last message in $Q[j,i]$. Operation E_{ji}^d may falsify the premise, and so does S_i^D if the last message is received. ■

Lemma 3.2. Using protocol D, the following assertions hold invariantly:
(1) the state is balanced and $state_i = state_j \Rightarrow n_i = n_j$,
(2) the state is balanced and $state_i \neq state_j \Rightarrow n_i = n_j - 1$.

Proof. We prove the assertions by simultaneous induction. By lemma 2.2 we know that we reach a balanced state from another balanced state by one operation S_i^D and one operation R_j^D in an arbitrary order, as the code of the operations is indeed independent. Initially $state_i = state_j = 0$ and $n_i = n_j = 1$. If the state is balanced and $state_i = state_j = 0$, we know by lemmas 3.1 and 2.1 that the first message in both queues is $<\lambda>$, hence the next balanced state will be $state_i = state_j = 1$. As n_i nor n_j is changed upon receipt of $<\lambda>$, $n_i = n_j$ still holds. If $state_i = 1$, then the next $state_i = 0$ and n_i is not changed. If $state_j = 1$, then either n_j is increased by one and $state_j$ remains 1, or n_j is not changed but $state_j$ is set to 0. Hence the next balanced state has $state_i = state_j = 0$ and $n_i = n_j$ or $state_i = 0$, $state_j = 1$, and $n_i = n_j - 1$. From this last case, we get in the next state $state_j = 0$ and n_j unchanged because $<\lambda>$ was received. Operation S_i^D either sets $state_i$ to 1 and lets n_i unaltered, or increases n_i by one and leaves $state_i = 1$. Hence the next balanced state has either $state_i = 1$, $state_j = 0$, and $n_i = n_j - 1$; or $state_i = state_j = 1$ and $n_i = n_j$. From the first one we get $state_i = 0$ and n_i unchanged because we had $<\lambda> \in Q[j,i]$, and from $state_j = 0$ we get $state_j = 1$ and n_j is unchanged, hence we get $state_i = 0$, $state_j = 1$, and $n_i = n_j - 1$. This exhausts all possibilities. ■

Theorem 3.3. Using protocol D, $string_i[1:n_j - 1] = string_j[1:n_j - 1]$.

Proof. Initially n_j is 1, and the strings are both empty. The only operation which affects the assertion is R_j^D, when n_j is increased. This is only done if $state_j$ was 1 and $<x> \in Q[i,j]$ with $x \neq \lambda$ held. Thus the last balanced state was $state_i = state_j = 1$ and hence $n_i = n_j$. By lemma 3.1 we have that for this received message $<x>$, $x = string_i[n_i]$ and hence $x = string_i[n_j]$. As x is written in $string_j[n_j]$ and n_j is increased afterwards in R_j^D, we now have $string_i[n_j - 1] = string_j[n_j - 1]$. As the assertion held for the old value of n_j, we now have $string_i[1:n_j - 1] = string_j[1:n_j - 1]$ for the new value. ■

This concludes the assertional proof of the partial correctness of protocol D for the case of deletion errors only.

3.2. Deletion and mutation errors.

We now assume that deletion and mutation errors can occur, but no insertion errors. As the difference between the bits 0 and 1 now can be obscured by mutation errors, another way is needed to distinguish them. In protocol D an alternation between odd and even time steps marked the difference between retransmissions and new values sent, while now an "alternation" between time steps that are 0, 1, and 2 modulo 3 distinguishes between retransmissions, a new 0, and a new 1. The protocol proposed by Aho et al. is as follows.

Protocol *DM*:

```
    Initially   $state_i = state_j = 0,\ n_i = n_j = 1,\ <\lambda> \in !\ Q[i,j],\ <\lambda> \in !\ Q[j,i]$.
    $S_i^{DM}$ : { $Q[j,i] \neq \varnothing$ }
        begin receive $<x>$ from $j$;
            if $state_i = 0$
            then send <1> to $j$; $state_i := 1$
            else send $<\lambda>$ to $j$;
                 if $state_i = 2 \wedge x \neq \lambda$
                 then if $string_i[n_i] = 1$ then $state_i := 1$ fi; $n_i := n_i + 1$
                 else $state_i := state_i + 1 \bmod 3$
                 fi
            fi
        end
    $R_j^{DM}$ : { $Q[i,j] \neq \varnothing$ }
        begin receive $<x>$ from $i$;
            if $x = \lambda$
            then send $<\lambda>$ to $i$; $state_j := state_j + 1 \bmod 3$
            else send <1> to $i$;
                 if   $state_j = 0$ then $string_j[n_j] := 1$; $n_j := n_j + 1$
                 elif $state_j = 2$ then $string_j[n_j] := 0$; $n_j := n_j + 1$
                 fi; $state_j := 2$
            fi
        end
```

E_{ij}^d, E_{ji}^d, E_{ij}^m, and E_{ji}^m as in section 2.3.

For protocol *DM* we can derive the following invariants.

Lemma 3.4. Using protocol *DM*, the following assertions hold invariantly:

(1) $state_i = 0 \vee state_i = 2 \Rightarrow <\lambda>$ tail of $Q[i,j] \vee Q[i,j] = \varnothing$,

(2) $state_j = 0 \vee state_j = 1 \Rightarrow <\lambda>$ tail of $Q[j,i] \vee Q[j,i] = \varnothing$,

(3) $<x>$ tail of $Q[i,j] \wedge x \neq \lambda \Rightarrow state_i = 1$,

(4) $<x>$ tail of $Q[j,i] \wedge x \neq \lambda \Rightarrow state_j = 2$.

Proof. Obvious from the protocol and the allowed error operations. ∎

Lemma 3.5. Using protocol *DM*, the following assertions hold invariantly:

(1) the state is balanced and $state_i = state_j \Rightarrow$
$$n_i = n_j \wedge string_i[n_j - 1] = string_j[n_j - 1],$$

(2) the state is balanced and $state_i = state_j - 1 \bmod 3 \Rightarrow$
$$n_i = n_j + 1 \wedge string_i[n_j] = 0,$$

(3) the state is balanced and $state_i = state_j + 1 \bmod 3 \Rightarrow$
$$n_i = n_j + 1 \wedge string_i[n_j] = 1.$$

Proof. We will prove the three assertions by simultaneous induction. Again we can use lemma 2.2 to observe that in going from one balanced state to the next we need to consider one S_i^{DM} and one R_j^{DM} operation in arbitrary order. Initially $state_i = state_j = 0$ and $n_i = n_j = 1$, hence the strings are equal, as they are both empty. By lemma 3.4 we have that both queues contain $<\lambda>$, hence in the next balanced state $state_i = state_j = 1$ and n_i and n_j are unchanged. From this state we get $state_i = state_j = 2$ and n_i and n_j remain unaltered. As $state_i = 2$ implies that $<\lambda> \in Q[i,j]$, we have $state_j := 0$ and n_j unchanged. In operation S_i^{DM} three things can happen. First, $state_i := 0$ and n_i remains unchanged; secondly, $state_i := 1$, n_i is increased by one, and $string_i[n_i - 1] = string_i[n_j] = 1$, rendering assertion (3) true; and thirdly, $state_i$ remains 2, n_i is increased by one and $string_i[n_i - 1] = string_i[n_j] = 0$, rendering assertion (2) true. Starting from the balanced state $state_i = 2$, $state_j = 0$, $n_i = n_j + 1$, and $string_i[n_j] = 0$, we have $<\lambda> \in Q[i,j]$, hence the next $state_j = 1$, next $state_i = 0$, and n_i and n_j unchanged. Again, $<\lambda>$ is in both queues, hence the next balanced state is $state_i = 1$ and $state_j = 2$ with n_i and n_j unchanged. As in operation R_j^{DM} the message $<\lambda>$ was received, $<\lambda>$ was sent to i, also. Hence $state_i$ becomes 2 with n_i unchanged. From $state_j = 2$ now two things can happen: either $<\lambda>$ is received, $state_j := 0$, and n_j is unchanged, leaving assertion (2) true, or a non-λ symbol is received causing a 0 to be written in $string_i[n_j]$ and n_j is increased, thus rendering assertion (1) true. Starting from the balanced state $state_i = 1$, $state_j = 0$, $n_i = n_j + 1$, and $string_i[n_j] = 1$. As $<\lambda> \in Q[j,i]$, next $state_i = 2$ with n_i unchanged. In R_j^{DM} two things can happen: $string_i[n_j] := 1$, n_j is increased by one, and $state_j := 2$, rendering assertion (1) true, or $state_j := 1$ and n_j is unchanged leaving assertion (3) true. As $<\lambda>$ is in both queues, the next balanced state is $state_i = 0$ and $state_j = 2$ with n_i and n_j unchanged. The next state is $state_i = 1$, $state_j = 0$, with n_i and n_j unchanged, which completes all possibilities. ■

Theorem 3.6. Using protocol DM, $string_i[1:n_j - 1] = string_j[1:n_j - 1]$.

Proof. Follows from lemma 3.5. ■

This concludes the assertional verification of the safety of protocol DM.

3.3. Deletion and insertion errors. In case deletion and insertion errors (only) can arise, we encounter the problem that modeling errors by operations acting on the contents of the message queues, i.e., the contents of the links, leads to a difference with the model of Aho et al. In our model, operations can be executed in any order and as often as wished, as long as their guards are true. Thus, for a message $<0> \in Q[i,j]$ we can have a deletion error, changing this message to $<\lambda> \in Q[i,j]$. But nothing now prevents an insertion error, changing it to $<1> \in Q[i,j]$. Thus we have produced a mutation error, in the terminology of Aho et al., as in their case the kind of error is determined by the relation between the symbol sent and the symbol received. One way to restore the correspondence between the model of Aho et al. and ours is to restrict errors: to demand that a message in a queue is subject of an error operation only once. Another, easier, way is to use the fact that the protocols of Aho et al. for deletion and insertion errors avoid insertion errors by never sending λ: as λ is never sent, insertion errors in fact do not occur. Thus by leaving out the insertion error operations in the protocol skeletons for the case of deletion and insertion errors and not sending λ, we model the same situation as Aho et al.

Aho et al. gave two protocols for the case of deletion and insertion errors, but only in words. Although the idea upon which the first protocol ($DI\,1$) is based is simple, its actual code is not as simple as that of the previous protocols, at least if "simple" is defined as: consisting of only a few states. Our translation of the idea for the protocol needs four states for the sender and even six states for the receiver, although we are not sure this is the minimum number necessary with the idea of this protocol. The problem can be solved by a protocol with fewer states, as the second protocol ($DI\,2$) in this section demonstrates. The idea used in protocol $DI\,1$ is to distinguish new bits being sent and retransmissions of old bits of the string by a parity bit, like in the alternating bit protocol. However, the need to remember the current parity increases the number of states by a factor of two. In stating the protocol we will not do this, but write n_i mod 2 to denote the parity, since we feel that it is more clear in this way. However, if one wants to state the protocol as a finite state automaton in the model of Aho et al., each state has to be replaced by two states, one where the parity is odd, and one for an even parity. The protocol sends the bit to transmit in odd time steps and the control bit in even time steps. The straightforward transcription of this idea leads to the following protocol.

Protocol $DI\,1$:

```
    Initially   $state_i = state_j = 0,\ n_i = n_j = 1,\ <\lambda> \in !\ Q[i,j],\ <\lambda> \in !\ Q[j,i].$
    $S_i^{DI1} : \{ Q[j,i] \neq \emptyset \}$
        begin receive $<x>$ from $j$;
            if $x \neq \lambda \wedge x \neq n_i$ mod 2 then $n_i := n_i + 1$ fi;
            if $state_i = 0$
            then send $<n_i$ mod $2>$ to $j$
            else send $<string_i[n_i]>$ to $j$
            fi; $state_i := state_i + 1$ mod 2
        end
    $R_j^{DI1} : \{ Q[i,j] \neq \emptyset \}$
        begin receive $<x>$ from $i$;
            if $x \neq \lambda \wedge state_j = 0$ then $string_j[n_j] := x$;  $n_j := n_j + 1$ fi;
            if $state_j = 2 \vee state_j = 1 \wedge x \neq \lambda \wedge x = n_j$ mod 2
            then  $state_j := state_j - 1$
            else  $state_j := state_j + 1$
            fi; send $<n_j$ mod $2>$ to $i$
        end
    $E_{ij}^d$, and $E_{ji}^d$ as in section 2.3.
```

Processor j needs three states apart from the parity, because two states are used for remembering whether to expect a parity bit or a bit with string information, while the third is used for the case "a string bit is coming but it is probably a retransmission". The trick to repeatedly write retransmitted values in the same position of $string_j$ does not work, as it might also be a new value that is sent, but the receiver has no way of "knowing" that, and the old value would be lost.

Lemma 3.7. Using protocol $DI1$, the following assertions hold invariantly:
(1) $state_i = 0 \wedge <x>$ tail of $Q[i,j] \wedge x \neq \lambda \Rightarrow x = string_i[n_i]$,
(2) $state_i = 1 \wedge <x>$ tail of $Q[i,j] \wedge x \neq \lambda \Rightarrow x = n_i$ mod 2,
(3) $<x>$ tail of $Q[j,i] \wedge x \neq \lambda \Rightarrow x = n_j$ mod 2.

Proof. Obvious from the protocol skeleton and the allowed error operations. ■

Lemma 3.8. Using protocol $DI1$, the following assertions hold invariantly:
(1) the state is balanced $\Rightarrow state_i = state_j \vee state_i = 0 \wedge state_j = 2$,
(2) the state is balanced and $state_j = 0 \Rightarrow$
$$n_i = n_j \wedge string_i[n_j - 1] = string_j[n_j - 1],$$
(3) the state is balanced and $(state_j = 1 \vee state_j = 2) \Rightarrow$
$$(n_i = n_j \vee n_i = n_j - 1) \wedge string_i[n_j - 1] = string_j[n_j - 1].$$

Proof. (1). Obvious from the protocol skeleton. We prove assertions (2) and (3) by simultaneous induction. Initially (2) holds. As in a balanced state $state_i$ is determined by $state_j$ according to (1), we will only state the value of $state_j$ in the sequel. From assertion (2), the next balanced state has $state_j = 1$. As before this transition we had $n_i = n_j$, we have by lemma 3.7 (3) that n_i is unchanged. In R_j^{DI1} two things can happen. First, if the symbol received was unequal λ, we have by lemma 3.7 (1) that n_j is increased such that $n_i = n_j - 1$ and the new value is $string_j[n_j - 1] = string_i[n_j - 1]$. Secondly, if the symbol received was λ, then n_j remains unchanged. Hence assertion (3) is rendered true. From this balanced state with $n_i = n_j - 1$, if a non-λ symbol is received in S_i^{DI1}, then it was n_j mod 2 by lemma 3.7 (3), hence n_i is increased such that $n_i = n_j$ holds again. Otherwise, n_i remains $n_j - 1$. In operation R_j^{DI1} we have by lemma 3.7 (2) that the next $state_j = 2$ with n_j unchanged. From the balanced state with $state_j = 1$ and $n_i = n_j$, we get n_j unchanged and next $state_j$ is either 0 or 2. In S_i^{DI1} n_i remains unchanged because of lemma 3.7 (3). Thus assertion (3) remains true or (2) is rendered true. From the balanced state with $state_j = 2$ and $n_i = n_j - 1$ we get that the next $state_j = 1$, n_j unchanged, and in S_i^{DI1} n_i can remain unchanged or is increased such that $n_i = n_j$. Hence assertion (3) remains true. From the balanced state with $state_j = 2$ and $n_i = n_j$ we get next $state_j = 1$, n_j unchanged, and n_i unchanged, too. Thus assertion (3) remains true. This completes all possible transitions. ■

Theorem 3.9. Using protocol $DI1$, we have
$$string_i[1 : n_j - 1] = string_j[1 : n_j - 1].$$

Proof. Initially this is true because the strings are empty. By lemma 3.8 the assertion remains true as n_j is increased. ■

The second idea for a protocol for the case of deletion and insertion errors is based on protocol DM for the case of deletion and mutation errors in section 3.2. This latter protocol is based on the fact that the symbol λ cannot be changed by the error operations to a symbol $\neq \lambda$. If we allow only deletion and insertion errors, we know that the symbol 0 is never changed into the symbol 1. Thus we can use the same protocol in this case, if replace all symbols appropriately: "λ" by "0", and "$\neq \lambda$" by "1". Hence we get the following protocol.

Protocol *DI*2:

```
Initially   $state_i = state_j = 0,\ n_i = n_j = 1,\ <0> \in !\ Q[i,j],\ <0> \in !\ Q[j,i]$.
$S_i^{DI2}$ : { $Q[j,i] \neq \varnothing$ }
      begin receive $<x>$ from $j$;
            if $state_i = 0$
            then  send <1> to $j$;  $state_i := 1$
            else  send <0> to $j$;
                  if $state_i = 2 \wedge x = 1$
                  then  if $string_i[n_i] = 1$ then $state_i := 1$ fi;  $n_i := n_i + 1$
                  else  $state_i := state_i + 1 \bmod 3$
                  fi
            fi
      end
$R_j^{DI2}$ : { $Q[i,j] \neq \varnothing$ }
      begin receive $<x>$ from $i$;
            if $x \neq 1$
            then  send <0> to $i$;  $state_j := state_j + 1 \bmod 3$
            else  send <1> to $i$;
                  if     $state_j = 0$ then $string_j[n_j] := 1$;  $n_j := n_j + 1$
                  elif   $state_j = 2$ then $string_j[n_j] := 0$;  $n_j := n_j + 1$
                  fi;  $state_j := 2$
            fi
      end
```

E_{ij}^d, and E_{ji}^d as in section 2.3.

Now we can derive the following invariants.

Lemma 3.10. Using protocol *DI*2, the following assertions hold invariantly:

(1) $state_i = 0 \vee state_i = 2 \Rightarrow <x>$ tail of $Q[i,j] \wedge x \neq 1 \vee Q[i,j] = \varnothing$,

(2) $state_j = 0 \vee state_j = 1 \Rightarrow <x>$ tail of $Q[j,i] \wedge x \neq 1 \vee Q[j,i] = \varnothing$,

(3) <1> tail of $Q[i,j] \Rightarrow state_i = 1$,

(4) <1> tail of $Q[j,i] \Rightarrow state_j = 2$.

Proof. Obvious from the protocol skeleton and the allowed error operations. ∎

Corollary 3.11. Using protocol *DI*2, we have

$$string_i[1:n_j-1] = string_j[1:n_j-1].$$

Proof. This follows from lemma 3.10, the correspondence with operations S_i^{DM} and R_j^{DM}, and theorem 3.6. ∎

Similarly, this protocol can be used for the remaining possibilities of classes of errors.

4. References.

[AWYU] Aho, A.V., A.D. Wyner, M. Yannakakis, and J.D. Ullman, *Bounds on the size and transmission rate of communications protocols*, Comp. & Maths. with Appls. vol 8 (1982), 205-214.

[ChMi] Chandy, K.M., and J. Misra, *Parallel Program Design, A Foundation*, Addison-Wesley Publ. Comp., Reading, Mass., 1988.

[HaZu] Halpern, J.Y., and L.D. Zuck, *A little knowledge goes a long way: Simple knowledge-based derivations and correctness proofs for a family of protocols*, Proc. 6th PoDC, Vancouver, Canada, 1987, pp. 269-280; extended version available as: Report RJ5857, IBM Almaden Research Center, San Jose, CA, 1987 (revised 1989).

[Knu] Knuth, D.E., *Verification of link-level protocols*, BIT 21 (1981), 31-36.

[Kro] Krogdahl, S., *Verification of a class of link-level protocols*, BIT 18 (1978), 436-448.

[Lam] Lamport, L., *An assertional correctness proof of a distributed algorithm*, Science of Computer Programming 2 (1982), 175-206.