Specification of the IEEE 802.5 Token Ring LAN in LOTOS

Michal IGLEWSKI Abdellatif OBAID
Université du Québec à Hull
Département d'informatique
C.P. 1250, succ."B", Hull, Que., CANADA
J8X 3X7
e-mail: iglewski@qucis.queensu.ca, obaid@UQHULL.bitnet

Abstract

LOTOS is a specification language that was developed within the ISO as a formal description technique for protocol specification and validation. Local area networks are growing in popularity and the design of communication protocols for these networks should be verified using formal description techniques. In this paper we present a specification, written in LOTOS, of an access protocol for a Token Ring Local Area Network. The low-level protocol and service described is a subset of the IEEE 802.5 standard.

1 Introduction

LOTOS ([ISO 87], [Bol 87] and [Log 88]) is a specification language that is intended for specifying distributed systems. It allows systems to be composed of concurrently running subsystems which can communicate both synchronously and asynchronously. LOTOS has been widely used to specify Open Systems Interconnection (OSI) protocols and services (cf. e.g. [Sco 89]). Though this language has features that allow describing system behaviors on different levels of abstraction, it was usually used only for fairly high-levels protocols. In this paper we show how LOTOS can be used for low-level protocols and services. We choose a subset of the IEEE 802.5 Token Ring LAN [IEEE 85] and specify its behavior in LOTOS. First, we describe a protocol for one station connected to this LAN. Then we specify the behavior of the whole network with several stations connected in a ring configuration. Several simulations of the specification have been run using a LOTOS interpreter [Haj 88].

We do not describe LOTOS language in this paper - a good tutorial can be found e.g. in [Bol 87].

2 Token Ring Network

A token ring network [Dix 87] is a collection of stations serially connected by a transmission medium. Each station is identified by a unique address and knows the address of its neighbors. Information is transferred sequentially, bit by bit, from one station to the next (always in one direction). A special bit pattern, called a *token frame* (in short: *token*), circulates around the ring whenever all stations are idle. When a station wishes to transfer some data, it must first capture the token and then modify it to a start-of-data-frame sequence and append control fields, and the data to be transferred. The so-formed *data frame* is transmitted to the downstream neighbor. This station possibly modifies one particular field and copies the data (if the frame was sent to this station) but always passes the frame to the downstream neighbor. Eventually the data frame reaches the original sender, who changes it back to the token. Because there is only one token, only one station can transmit data at a

given instant; because a sender must always return the token to the ring (once it converted the data frame back into the token frame), no station can monopolize the ring. If the token is passed around the ring without being used, then a station can once again seize it and transmit the next data frame.

Within the IEEE standard there is also a possibility to use priorities for the ring access control. To each station is assigned a priority ranking from zero to seven, what gives a station an opportunity to reserve the use of the ring for the next transmission round. This reservation scheme is not discussed in this paper.

2.1 Token Ring Protocol Architecture

The token ring protocol uses only the two lowest layers of the OSI Reference Model: the *Physical layer* and the *Data Link layer*, with the latter subdivided into *Medium Access Control layer* (*MAC*) and the *Logical Link Control layer* (*LLC*). The three layers communicate via *Service Access Points*, *MSAP* and *PSAP*, as shown in Figure 1.

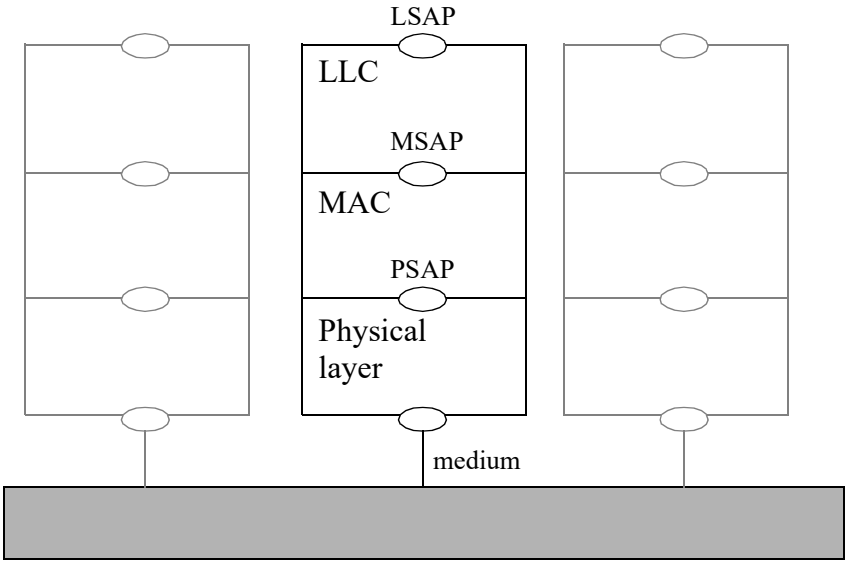

Figure 1: Token Ring Protocol architecture

2.1.1 LLC Layer Functions

The LLC supports media independent data link functions, and uses the services of the MAC layer to provide services to the network layer. It provides two forms of service for transmission of data frames between two neighbor stations: *unacknowledged connectionless service* and *connection-oriented service*. With the unacknowledged connectionless service each frame sent is individually acknowledged. This is achieved by the use of three service primitives: *L_DATA.request*, *L_DATA.indication* and *L_DATA*.confirmation. When a connection-oriented service is offered, the source and destination stations establish a connection before any data are transferred (the extra service primitive *L_DATA*.response is used). We will specify only the unacknowledged connectionless service in this paper.

2.1.2 MAC Layer Functions

The MAC layer provides mechanisms to control the access to the shared medium; it is therefore a

necessary part of the protocol ensuring that two stations do not send data at the same time. The MAC layer exchanges frames between data link layer entities. When there is no data traffic on the ring, the token circulates until a station sets a specific bit (called *token bit T*) to 0, thereby seizing the token. If the address in the frame's header indicates that the data is destined for the attached station, the ring interface copies the data and passes the information to this station.

Frame formats

In the IEEE 802.5 standard [IEEE 85] messages are structured as frames. There are two frame formats: token frames and data frames.

The token frame format (cf. Figure 2) consists of three fields only: the *Starting Delimiter* (*SD*), the *Access Control* (*AC*), and the *Ending Delimiter* (*ED*). The *AC* field contains a one-bit token indicator (*T*) that is set to 0 to indicate the token frame.

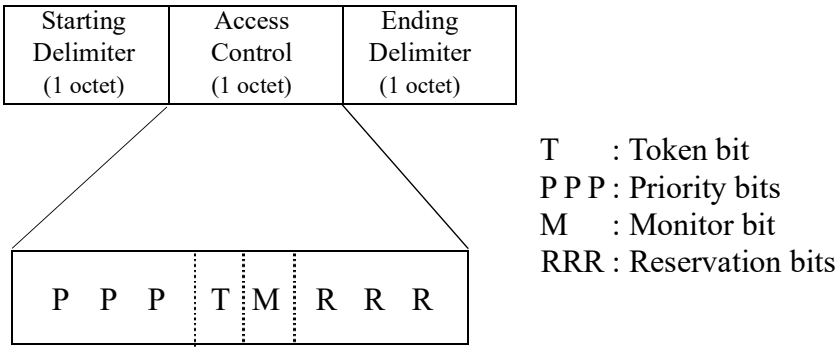

Figure 2: Token Frame Format

The data frame format consists of 9 fields (cf. Figure 3). Three of them (SD, AC, ED) are the same as before, with token indicator *T* set to 1. The *Frame Control* field (*FC*) defines the type of frame (i.e. MAC frame or LLC frame), the address fields identify the sending and receiving stations, the *Information* field contains user data, the *Frame-Check Sequence* (*FCS*) field is used for error checking, and the *Frame Status* (*FS*) field is used to indicate the conditions in which the receiving station has accepted or refused the frame. The latter field contains two special bits, *A* and *C*. They are transmitted as 0 by the station originating the frame. If another station recognizes the destination address as its own address, it will set the bit *A* to 1. If it copies the frame (into its receive buffer), it will also set the bit *C* to 1. Three combinations are possible:

a) *A=0*, *C=0*: Destination not present. The station will fail to accept the frame.

b) *A=1*, *C=0*: Destination present but frame not accepted (e.g. because there is not enough buffer space). The sending station may retry later.

c) *A=1*, *C=1*: Destination present and frame accepted. When the frame returns back to the original sender, this combination constitutes a positive acknowledgment.

Starting Delimiter (1 octet)	Access Control (1 octet)	Frame Control (1 octet)	Destination Address (2 or 6 octets)	Source Address (2 or 6 octets)	Information (0 or more oct.)	Frame-Check Sequence (4 octets)	Ending Delimiter (1 octet)	Frame Status (1 octet)

Figure 3: Data Frame Format

2.1.3 Physical Layer Functions

The physical layer is responsible for the physical transfer of bits between stations. The interfacing between a station and the ring is done via the *Ring Interface Unit* (*RIU*) which is an active device attached to each station that has the ability to recognize its own address in the received data in order to accept them. Each bit arriving within a specified time interval is copied into 1-bit buffer and then copied out onto the ring. While in the buffer, the bit can be inspected and possibly modified before being retransmitted.

The IEEE 802.5 standard physical layer provides for detailed description of the encoding techniques that are used. Since this layer is concerned with many electrical details, it is usually not specified formally. In our specification we model the physical layer at the byte level rather than at a bit one. This was necessary because otherwise the specification itself and the simulation time would have been very long.

3 Protocol Specification

Before presenting the specification let us explain how we model the communication. As shown in Figure 1, the ring is modelled by a single gate called *medium*. In order to model several stations connected to the same ring, the behavior expression describing the interaction will also contain the identity of the sending station together with some extra information. For example, if station *n* wants to perform a request for sending data *d* to its downstream neighbor, it writes m*edium!req!next(n)!d* to the communication channel medium.

As mentioned before, we assume that a byte (a value of type *Octet* in the specification) is the elementary unit of exchanged information. For this reason, in several parts of this specification, we analyze a frame octet by octet. This procedure allows us to analyze, for example, the *AC* field in order to recognize if the frame is the token or not. We also treat the source and destination address fields as a single octet.

Any octet that is transmitted by the MAC layer must be confirmed by the physical layer using a constant value called *conf*.

3.1 Specification of a Station

The behavior of a station with the three layers described above is modelled by the process *Station*:

```
process Station[lsap,medium] (Node:Octet) : noexit :=
    hide msap,psap in
    (   LLC[lsap,msap] (Node)
    |[msap]|
        MAC[msap,psap] (Node)
    |[psap]|
        PhyLayer[psap,medium] (Node)
    )
endproc
```

3.2 Specification of the LLC Layer

The LLC layer is modelled by the process *LLC*. This process has two gates that represent the two SAPs: *lsap* and *msap*. It can send and receive data concurrently:

```
process LLC [lsap,msap] (Node:Octet) : noexit :=
        Receive [lsap,msap]
    |||
        Send [lsap,msap]
endproc
```

Process *Receive* either receives new data from the MAC layer throughout the gate *msap* or refuses to receive because its buffer is full (this is modelled by the internal event *i*):

```
process Receive [lsap,msap] : noexit :=
        i;Receive [lsap,msap]      (* simulating a "buffer full" condition *)
    []
        msap!ind?Source:Octet?Data:OctetString;
        lsap!ind!Source!Data;
        Receive [lsap,msap]
endproc
```

Sending is done by the process *Send*. This process receives a request from the station to send data to another station identified by *Dest*. Then this request is sent to MAC layer by the process *RequestToSend*:

```
process Send [lsap,msap] : noexit :=
    lsap!req?Dest:Octet?Data:OctetString;
    RequestToSend [msap] (Dest,Data)
    >>
    Send [lsap,msap]
endproc
```

RequestToSend is the process that actually executes the request on behalf of an LLC layer entity and waits for confirmation by receiving the *FS* field. When this arrives, it checks the value of bits *A* and *C* of the received frame: if the bits *A* and *C* are set to 1 this means that the frame was correctly received and copied; if the bit *A* is set to 1 and the bit *C* is set to 0 this means that the receiving station did not accept the frame sent to it; if the bit *A* is set to 0 this means that the destination is unknown. In the first and the third case this process simply terminates and passes control to process *Send*. In the second case it keeps on trying by sending the same frame. Eventually, this process will terminate at either the first or the third condition.

```
process RequestToSend [msap] (Dest:Octet,Data:OctetString) : exit :=
    msap!req!Dest!Data;
    msap!conf?FS:Octet;
    (   [is_set_bit_A(FS) and is_set_bit_C(FS)]->
            exit                                    (* frame received and copied *)
    []
        [is_set_bit_A(FS) or not(is_set_bit_C(FS))]->
            RequestToSend [msap] (Dest,Data)        (* frame not copied *)
    []
        [not(is_set_bit_A(FS))]->
```

```
            exit                                    (* unknown address *)
    )
endproc
```

3.3 Specification of the MAC Layer

The MAC layer is modelled by the *MAC* process. In the specification that follows we assume that neither we have frame sequencing capability nor we perform error correction (i.e. we ignore the *FCS* information). The *MAC* process is specified as follows:

```
process MAC [msap,psap] (Node:Octet) : noexit :=
    Monitor[psap](Node)
    >>
    Manager[msap,psap](Node)
endproc
```

According to the IEEE standard, one particular station should play the role of the *monitor*. One of the functions of the monitor is to generate the token when it is lost or damaged. We use the monitor to decide which station initially puts the token into the ring (*station_1* in our case). Process *Monitor* sends to *station_1* a token frame (bit $T = 0$) with bits A and C equal to 0. This is done in several steps. First, the physical layer puts the station in *transmit mode*. Then, *station_1* sends the token frame octet by octet and for each octet it transmits it waits for a confirmation from the physical layer. The monitor behaves as follows:

```
process Monitor[psap](Node:Octet) : exit :=
        [Node eq station_1] ->
            psap!ind!transmit?OctetRing:Octet;
            psap!req!sd;
            psap!conf;
            psap!ind!transmit?OctetRing:Octet;
            psap!req!Octet(0,0,0,0,0,0,0,0);      (* AC: Token Bit = 0 *)
            psap!conf;
            psap!ind!transmit?OctetRing:Octet;
            psap!req!ed;
            psap!conf;
            exit
    []
        [Node ne station_1] ->
            exit
endproc
```

Process *Manager* is responsible for controlling the access to the ring. It behaves differently depending on whether the station is in the *repeat mode* or in the *transmit mode*. If it is in the repeat mode and gets the token frame (with bit $T = 0$) from the physical layer, it enters the transmit mode. In this mode it transmits data (after setting bit T to 1) if the station has something to send, by calling process *SendFrame*, or it simply retransmits the token, if there is nothing to send.

If a station is in the transmit mode and gets the *AC* octet, then it enters the repeat mode and continues receiving the subsequent octets until it receives the Destination Address. This field is checked by the station to see if it is the proper destination. If so, it copies the field (by calling

CopyFrame).

In the repeat mode any received octet that is different from *SD* is simply passed forward as is.

Process *Manager* is described as follows:

```
process Manager[msap,psap](Node:Octet) : noexit :=
    (   psap!ind!repeat?Data:Octet[Data eq sd];
        (   psap!ind!transmit?AC:Octet[not(is_set_bit_T(AC))];
            (   (   exit(nothingToSend,Octet(0,0,0,0,0,0,0,0),
                            Octet(Octet(0,0,0,0,0,0,0,0)))
                [>
                    msap!req?Dest:Octet?UserData:OctetString;
                    exit(somethingToSend,Dest,UserData)
                )
                >> accept
                    LLCreq:LLCrequest,
                    Dest:Octet,
                    UserData:OctetString
                in
                (   [LLCreq eq nothingToSend] ->
                        psap!req!AC;
                        psap!conf;
                        exit
                []
                    [LLCreq eq somethingToSend] ->
                        psap!req!setBit_T(AC);
                        psap!conf;
                        SendFrame [msap,psap] (Node,Dest,UserData)
                )
            )
        []
            psap!ind!transmit?AC:Octet[is_set_bit_T(AC)];
            psap!req!AC;
            psap!conf;
            psap!ind!repeat?FC:Octet;
            psap!ind!repeat?Dest:Octet;
            (   [Dest eq Node] ->
                    CopyFrame[msap,psap]
            []
                [Dest ne Node] -> exit
            )
        )
    []
        psap!ind!repeat?Data:Octet[Data ne sd];
        exit
    )
    >> Manager[msap,psap] (Node)
endproc
```

Process *CopyFrame* assumes that a station is in the repeat mode. Upon receiving the Source Address, this process copies the subsequent octets. The number of copied octets equals to a constant value given in *UserDataLength*. *CopyFrame* concatenates these octets (by calling process *CopyOctets*) to form a frame (this is actually done in two steps: one for user data and one for *FCS* string). The last field expected in the repeat mode is the *ED* octet. Upon receiving this field the station changes its mode into the transmit one.

In the transmit mode either the condition *BufferFull* is generated or the received frame is delivered to the LLC layer through the *msap* gate. In the former case, the received frame is put back into the ring by setting the bit *A* to 1. In the latter case, the bits *A* and *C* are set to 1 to indicate the normal operation.

```
process CopyFrame[msap,psap] : exit :=
    psap!ind!repeat?Source:Octet;
    (   CopyOctets[psap] (userDataLength)
        >> accept Data:OctetString in
        (   CopyOctets[psap] (fcsLength)
            >> accept FCS:OctetString in
            (   psap!ind!repeat?ED:Octet;
                psap!ind!transmit?FS:Octet;
                (   (   exit(BufferFull)
                        [>
                        msap!ind!Source!Data;
                        exit(Copied)
                    )
                    >> accept BState:BufferState in
                    (   [BState eq BufferFull] ->
                            psap!req!setBit_A(FS);
                            psap!conf;
                            exit
                    []
                        [BState eq Copied] ->
                            psap!req!setBits_AC(FS);
                            psap!conf;
                            exit
                    )
                )
            )
        )
    )
endproc
```

CopyOctets is the process that forms frames out of the received octets. The number of these octets is passed as a parameter. This process is described as follows:

```
process CopyOctets[psap] (DataLength:Nat) : exit(OctetString) :=
    Copy[psap] (DataLength,<>)
    >>
    accept Data:OctetString in
    exit(Data)
```

```
where
    process Copy[psap] (Length:Nat,Data:OctetString) : exit(OctetString) :=
            [Length eq Succ(0)] ->
                psap!ind!repeat?X:Octet;
                exit(X + Data)
        []
            [Length gt Succ(0)] ->
                psap!ind!repeat?X:Octet;
                Copy[psap](dec(Length),X + Data)
    endproc
endproc
```

Sending frames is accomplished by process *SendFrame*. In the transmit mode, if a station has something to send, this process sends a frame given its user data (*UserData*), the identity of the sending station (*Node*) and the identity of the destination (*Dest*). This process sends frames (one field at-a-time). While sending, a station may receive the first octet of its own frame (that is the field *SD*) or it may happen that some idle time is required before this field is received. When this field is received, the token is reinserted (bit $T = 0$). Process *SendFrame* is decomposed into two concurrent processes (*TransmitFrame* and *RemoveFrame*) that synchronize by executing the action *sdIsBack*:

```
process SendFrame[msap,psap]
                (Node:Octet, Dest:Octet, UserData:OctetString) : exit :=
    hide sdIsBack in
        TransmitFrame[psap,sdIsBack] (Node,Dest,UserData)
        |[sdIsBack]|
        RemoveFrame[msap,psap,sdIsBack]
endproc
```

TransmitFrame is the process that transmits frames octet by octet onto the ring. It also waits for confirmation after each octet it sends. After receiving the frame that it sent, it puts the token back into the ring:

```
process TransmitFrame[psap,sdIsBack]
            (Node:Octet, Dest:Octet, UserData:OctetString) : exit :=
    psap!req!Octet(0,0,0,0,0,0,0,0);      (* FC: not used *)
    psap!conf;
    psap!req!Dest;
    psap!conf;
    psap!req!Node;                        (* local node *)
    psap!conf;
    TransmitOctets[psap](UserData,userDataLength) >>
    TransmitOctets[psap](fcs,fcsLength) >>
    psap!req!ed;
    psap!conf;
    psap!req!Octet(0,0,0,0,0,0,0,0);      (* FS: bits A and C equal 0 *)
    psap!conf;
    WaitForSD[psap,sdIsBack] >>
    PutToken[psap]
endproc
```

Process *TransmitOctets* is responsible for transmitting a number of octets given by the parameter *Length*. This process is called by *SendFrame* for sending user data as well as *FCS* strings. Its description is as follows:

```
process TransmitOctets[psap] (Data:OctetString, Length:Nat) : exit :=
        [Length eq Succ(0)] ->
            psap!req!first(Data);
            psap!conf;
            exit
    []
        [Length gt Succ(0)] ->
            psap!req!first(Data);
            psap!conf;
            TransmitOctets[psap](tail(Data),dec(Length))
endproc
```

Process *WaitForSD* models idle time for a station. If a station finishes transmitting the entire frame prior to receiving its own starting delimiter (*SD*), it continues to transmit idle characters (contiguous 0-bits) on the *psap* gate until it receives the first field of its own frame. This is signaled to the process *RemoveFrame* which waits for a synchronization on action *sdIsBack*.

```
process WaitForSD[psap,sdIsBack] : exit :=
    (   exit(sdStillComing)
        [>
        sdIsBack; exit(sdArrived)
    )
    >> accept SDst:SDstate in
    (   [SDst eq sdStillComing] ->
            psap!req!idle;
            psap!conf;
            WaitForSD[psap,sdIsBack]
    []
        [SDst eq sdArrived] ->
            exit
    )
endproc
```

Process *PutToken* simply generates and inserts a token into the ring:

```
process PutToken[psap] : exit :=
    psap!req!sd;
    psap!conf;
    psap!req!Octet(0,0,0,0,0,0,0,0);(* AC, token bit = 0 *)
    psap!conf;
    psap!req!ed;
    psap!conf;
    exit
endproc
```

By executing the action *sdIsBack* (after receiving the field *SD* of its frame), *RemoveFrame* causes that *TransmitFrame* puts the token back into the ring. At the same time this process continues to accept the rest of the frame until it receives the field ED.

```
process RemoveFrame[msap,psap,sdIsBack] : exit :=
        psap!ind!transmit?Data:Octet[Data eq sd];
        (   sdIsBack;  (* tell to TransmitFrame that it can *)
            exit       (* put back the token on the ring    *)
        |||
            psap!ind!transmit?AC:Octet;
            psap!ind!transmit?FC:Octet;
            psap!ind!transmit?Dest:Octet;
            psap!ind!transmit?Source:Octet;
            WaitForED[msap,psap]
        )
    []
        psap!ind!transmit?Data:Octet[Data ne sd];
        RemoveFrame[msap,psap,sdIsBack]
endproc
```

Waiting for the *ED* field is done by the process *WaitForED*:

```
process WaitForED[msap,psap] : exit :=
        psap!ind!transmit?Data:Octet[Data eq ed];
        psap!ind!transmit?FS:Octet;
        msap!conf!FS;
        exit
    []
        psap!ind!transmit?Data:Octet[Data ne ed];
        WaitForED[msap,psap]
endproc
```

3.4 Specification of the Physical Layer

Depending on the mode, the physical layer repeats or transmits the octets it receives. In our specification this layer describes the behavior of the Ring Interface Unit (*RIU*). This interface communicates with the process *Medium* via the gate *medium* using the communication pattern described at the beginning of the Section 3.

```
process PhyLayer [psap,medium] (Node:Octet) : noexit :=
    RIU[psap,medium](Node)
endproc
```

RIU is the process that models the interface between a station and the ring. In the repeat mode, the received octets are simply copied onto the gate *psap*. In the transmit mode, it sends each received octet and waits for confirmation. In both modes data is sent to the next station (with identifier *next(Node)*). This process is described as follows:

```
process RIU[psap,medium](Node:Octet) : noexit :=
    medium!ind!Node?Data:Octet;
    (   psap!ind!repeat!Data;      (* repeat state *)
        exit(Data)
    []
        psap!ind!transmit!Data;   (* transmit state *)
        psap!req?Data:Octet;
        psap!conf;
        exit(Data)
    ) >> accept Data:Octet in
    medium!req!next(Node)!Data;
    RIU[psap,medium](Node)
endproc
```

The process Medium is specified below. It simply represents connections between the Ring Interface Units:

```
process Medium[medium](Node:Octet) : noexit :=
    medium!req!Node?Data:Octet;
    medium!ind!Node!Data;
    Medium[medium](Node)
endproc
```

3.5 Protocol Specification

It is essential that each station knows its number, so that it can identify its downstream neighbor. We model the protocol by linking several stations to a network according to the diagram given in Figure 4.

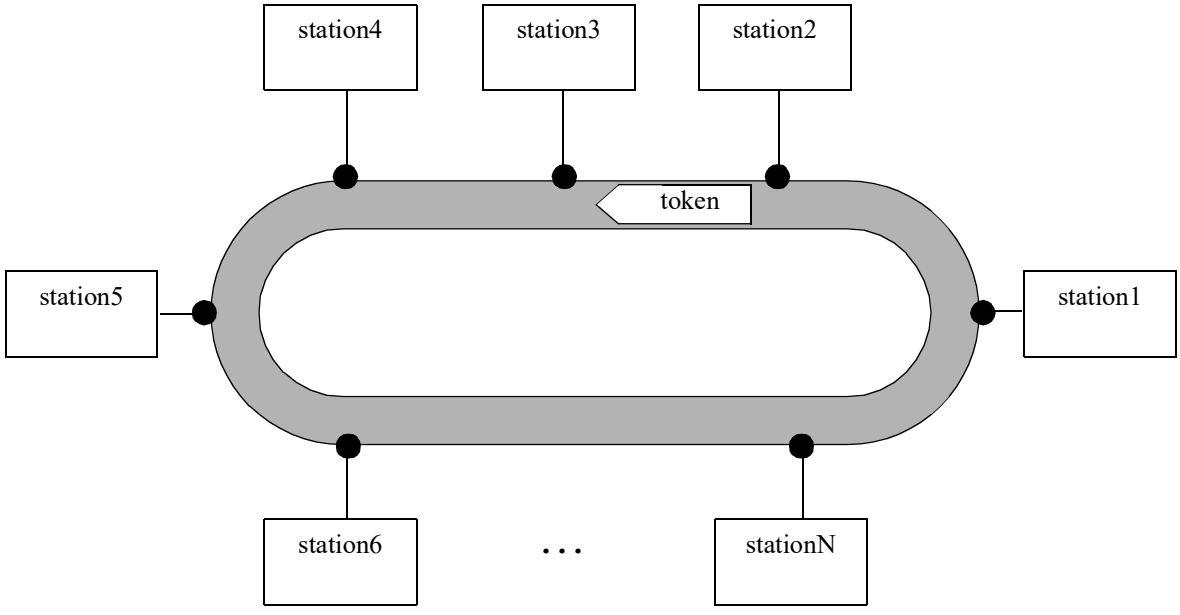

Figure 4: The Ring Architecture and the Token

The overall structure of this protocol can be presented as shown below. For simplicity, we chose to model a collection of three stations only.

```
hide medium in
    (InitRing[medium] >>
    (Station[lsap,medium](station_1)
    |||
    Station[lsap,medium](station_2)
    |||
    Station[lsap,medium](station_3)
    ))
|[medium]|
    (Medium[medium](station_1)
    |||
    Medium[medium](station_2)
    |||
    Medium[medium](station_3)
    )
```

InitRing simply allows to activate each interface by sending idle octets. It behaves according to the following process:

```
process InitRing [medium] : exit :=
    medium!req!station_1!idle;
    medium!req!station_2!idle;
    medium!req!station_3!idle;
    exit
endproc
```

The generalization of this scheme could also be done as follows:

```
hide medium in
    (InitRing[medium] >> Stations[lsap,medium](NbOfStations))
    |[medium]|
    Media[medium](NumberOfConnections)
```

where *Stations* is the description of several parallel instantiations of one station. Process *Station* can be defined as follows:

```
process Stations[lsap,medium](N: Integer) : noexit :=
    [N eq 0] -> Station[lsap,medium](N)
    []
    [N gt 0] -> (Station[lsap,medium](N)
                |||
                Stations[lsap,medium](N-1))
endproc
```

and *Media* could be defined in a similar way, as follows:

```
process Media[medium](N: Integer) : noexit :=
    [N eq 0] -> Medium[medium](N)
    []
    [N gt 0] -> (Medium[medium](N)
                |||
                Media[medium](N-1))
endproc
```

The text of the protocol specification is given in the Appendix 1. Data types are specified in the ACT ONE specification language that is part of LOTOS. We used the LOTOS data type library [ISO 87]. Part of an execution session script is given in the Appendix 2.

4 Conclusions

We used LOTOS to specify a simplified version of the IEEE 802.5 protocol. This language is suitable for expressing the modularity of different layers (LLC, MAC, Physical). We found out that it is also relatively easy to extend the existing specification. For example, when we replaced first version of the *Monitor* process by a new, elaborate one, it did not cause any global changes.

LOTOS has other powerful constructs for specifying more complicated systems. For example, we could use the full synchronization operator, ||, in order to apply the constraint oriented style. It would allow for more constructive specification.

However, we have noticed some shortcomings in the capabilities of the language. Consider a station that has some data to send. It must seize the token and append the data octets to the beginning of the frame. When a station does not have any data to send, it simply passes the token frame to the next station. In order to specify such a situation we used the LOTOS disabling operator, [>, as illustrated (in a simplified form) below:

```
(   exit(nothingToSend,anyOctet,anyOctet)
[>
    msap!req?Dest:Octet?UserData:Octet;
    exit(somethingToSend,Dest,UserData)
)
>> accept Condition:ConditionType,Octet1:Octet,Octet2:Octet
in
(   [Condition eq nothingToSend] -> do_this
[]
    [Condition eq somethingToSend] -> do_that
)
```

This solution is a bit artificial. We used the disabling operator because we thought that the transmission of data could be viewed as an interruption of the token circulation. According to the semantics of the language, the behavior expressed above is equivalent to:

```
i; do_this []  msap!req?Dest:Octet?UserData:Octet; i; do_that
```

However, this expression models a behavior with the “priority” assigned to the first condition.

What we actually want to model is the possibility to have both alternatives somehow exclusive: if there is something to send then the first alternative should not be offered. This capability is not possible in LOTOS.

A fundamental problem that we encountered was a lack of tools for a quick exploration of the behavior of the system. The execution of the specification with ISLA [Haj 88] was done manually in a step-by-step manner. This way of proceeding is rather a heavy burden. Automatic generation of execution paths is time consuming and the execution itself is hard to follow.

Acknowledgments

Martin Poirier helped to verify the specification and develop a version handling priorities. We are also grateful to Jan Madey and A. John van Schouwen for helpful comments on earlier drafts of this paper.

References

[Bol 87] Bolognesi, T., Brinksma, E., “Introduction to the ISO Specification Language LOTOS”, Computer Networks and ISDN systems, Vol. 14, No. 1, 1987, pp. 25-59.

[Dix 87] Dixon, R.C., “Lore of the Token Ring”, IEEE Network Magazine, vol. 1, Jan. 1987, pp. 11-18.

[Haj 88] Haj-Hussein, M., “ISLA: An Interactive System for LOTOS Applications”, M.Sc thesis, University of Ottawa, 1988.

[IEEE 85] IEEE Standards for Local Area Networks: Token Ring Access Method and Physical Layer Specifications, ANSI/IEEE Std 802.5, New York, The Institute of Electrical and Electronics Engineering Inc., April 1985.

[ISO 87] International Organization for Standardization, Information Processing Systems, Open Systems Interconnection, LOTOS - A Formal Description Technique Based on Observable Behavior (ISO IS 8807), 1987.

[Log 88] Logrippo, L., Obaid, A., Fehri, M., Briand, J.P., “LOTOS: An Executable Specification Language for Distributed Systems”, Software - Practice and Experience, Vol. 122, No. 4, 1988, pp. 365-385.

[Sco 89] Scollo, G., “Formal Description of the OSI Session Layer: Transport Service”, The Formal Description Technique LOTOS, North-Holland, 1989, pp. 97-116.

Appendix 1

Specification of a Subset of the IEEE 802.5 Standard of Token Ring in LOTOS

```
(*************************************************************)
(*          Subset of the IEEE Standard 802.5 of TokenRing                 *)
(*************************************************************)

specification TokenRingProtocol[lsap0,lsap1,lsap2]: noexit

library
    Bit,Octet,OctetString,Boolean,NaturalNumber
endlib

 type DataType is Octet,OctetString
    opns
        idle,
        sd,                 (* starting delimiter *)
        ed: -> Octet        (* ending delimiter *)

        fcs: -> OctetString

        is_set_bit_A,
        is_set_bit_C,
        is_set_bit_T: Octet -> Bool

        setBit_T,
        setBit_A,
        setBits_AC: Octet -> Octet

        first: OctetString -> Octet
        tail: OctetString -> OctetString
    eqns
        forall  X: Octet,
                b1,b2,b3,b4,b5,b6,b7,b8: Bit,
                S: OctetString
        ofsort Octet
            idle = Octet(0,0,0,0,0,0,0,0);
            sd = Octet(1,1,0,1,1,0,0,0);
            ed = Octet(1,1,1,1,1,1,0,0);

            setBit_T(Octet(b1,b2,b3,b4,b5,b6,b7,b8)) = Octet(b1,b2,b3,1,b5,b6,b7,b8);
            setBit_A(Octet(b1,b2,b3,b4,b5,b6,b7,b8)) = Octet(1,b2,b3,b4,1,b6,b7,b8);
            setBits_AC(Octet(b1,b2,b3,b4,b5,b6,b7,b8)) = Octet(1,1,b3,b4,1,1,b7,b8);
            first(X + S) = X
        ofsort Bool
            is_set_bit_T(X) = Bit4(X) eq 1 of Bit;
            is_set_bit_A(X) = Bit1(X) eq 1 of Bit;
            is_set_bit_C(X) = Bit2(X) eq 1 of Bit
        ofsort OctetString
            fcs = Octet(0,0,0,0,0,0,0,0) + <>;
            tail(X + S) = S
    endtype

 type StationIdType is Octet
    opns
```

```
53              station_1,
54              station_2,
55              station_3:-> Octet
56              next : Octet-> Octet
57          eqns
58              forall X : Octet
59              ofsort Octet
60                  station_1 = Octet(0,0,0,0,0,0,0,1);
61                  station_2 = Octet(0,0,0,0,0,0,1,0);
62                  station_3 = Octet(0,0,0,0,0,0,1,1);
63                  X eq station_1 => next(X) = station_2;
64                  X eq station_2 => next(X) = station_3;
65                  X eq station_3 => next(X) = station_1;
66  endtype
67
68  type NaturalNumberPlus is NaturalNumber
69      opns
70          userDataLength,
71          fcsLength:    -> Nat
72          dec         : Nat-> Nat
73      eqns
74          forall X : Nat
75          ofsort Nat
76              userDataLength = Succ(0);
77              fcsLength = Succ(0);
78
79              dec(Succ(X)) = X
80  endtype
81
82  type TwoValuesEnum is Boolean
83  sorts Values
84      opns
85          val1,val2: -> Values
86          _eq_: Values,Values -> Bool
87      eqns
88          forall X: Values
89          ofsort Bool
90              X eq X = true
91  endtype
92
93  type ConnectionType is
94      sorts Connection
95      opns
96          req,ind,conf: -> Connection
97  endtype
98
99  behaviour
100     hide medium in
101     (   InitRing[medium]
102         >>
103         (   Station[lsap0,medium](station_1)
104         |||
105             Station[lsap1,medium](station_2)
106         |||
107             Station[lsap2,medium](station_3)
108         )
```

```
109     )
110     |[medium]|
111     (     Medium[medium](station_1)
112     |||
113           Medium[medium](station_2)
114     |||
115           Medium[medium](station_3)
116     )
117
118  where
119     process InitRing [medium] : exit :=
120           medium!req!station_1!idle;
121           medium!req!station_2!idle;
122           medium!req!station_3!idle;
123           exit
124     endproc
125
126     process Station[lsap,medium] (Node :Octet) : noexit :=
127           hide msap,psap in
128           (     LLC[lsap,msap] (Node)
129           |[msap]|
130                 MAC[msap,psap] (Node)
131           |[psap]|
132                 PhyLayer[psap,medium] (Node)
133           )
134
135     where
136           type RIUstateType is TwoValuesEnum renamedby
137                 sortnames RIUstate for Values
138                 opnnames
139                       repeat for val1
140                       transmit for val2
141           endtype
142
143           (*************************************************************)
144           (*                        Logical Link Control               *)
145           (*************************************************************)
146           (* - connectivity: one station per node                      *)
147           (* - queue: one frame                                        *)
148           (* - service: acknowledged connectionless (Type 3)           *)
149           (*************************************************************)
150           process LLC [lsap,msap] (Node :Octet) : noexit :=
151                       Receive [lsap,msap]
152                 |||
153                       Send [lsap,msap]
154
155           where
156                 process Receive [lsap,msap] : noexit :=
157                             i;Receive [lsap,msap]       (* simulating a “buffer full” condition *)
158                       []
159                             msap!ind?Source:Octet?Data:OctetString;
160                             lsap!ind!Source!Data;
161                             Receive [lsap,msap]
162                 endproc (* Receive *)
163
164                 process Send [lsap,msap] : noexit :=
```

```
165                 lsap!req?Dest:Octet?Data:OctetString;          (* “Data” must be different from “sd” *)
166                 RequestToSend [msap] (Dest,Data)               (* and “ed” constants to avoid conflicts *)
167                 >>
168                 Send [lsap,msap]
169
170             where
171                 process RequestToSend [msap] (Dest:Octet,Data:OctetString) : exit :=
172                     msap!req!Dest!Data;
173                     msap!conf?FS:Octet;
174                     (     [is_set_bit_A(FS) and is_set_bit_C(FS)]->
175                             exit (* frame received and copied *)
176                     []
177                             [is_set_bit_A(FS) or not(is_set_bit_C(FS))]->
178                             RequestToSend [msap] (Dest,Data)          (* frame not copied *)
179                     []
180                             [not(is_set_bit_A(FS))]->
181                             exit (* unknown address *)
182                     )
183                 endproc (* RequestToSend *)
184             endproc (* Send *)
185         endproc (* LLC *)
186
187         (*************************************************************)
188         (*                      Medium Access Control                *)
189         (************************************************************* *)
190         (*   Frame Sequencing        : none                           *)
191         (*   Reservation             : not used                       *)
192         (*   Frame Check Sequence    : not used                       *)
193         (*   Monitor service         : puts the token on the ring at the beginning  *)
194         (*   Error Recovery          : not used                       *)
195         (*************************************************************)
196         process MAC [msap,psap] (Node:Octet) : noexit :=
197             Monitor[psap](Node)
198             >>
199             Manager[msap,psap](Node)
200
201         where
202             (*********************************************************)
203             (*                      Monitor                          *)
204             (*********************************************************)
205             (*          puts initially the token on the ring         *)
206             (*********************************************************)
207             process Monitor[psap](Node:Octet) : exit :=
208                     [Node eq station_1] ->
209                         (* the station station_1 puts the token on the ring *)
210                         psap!ind!transmit?OctetRing:Octet;
211                         psap!req!sd;
212                         psap!conf;
213                         psap!ind!transmit?OctetRing:Octet;
214                         psap!req!Octet(0,0,0,0,0,0,0,0);          (* AC: Token Bit = 0 *)
215                         psap!conf;
216                         psap!ind!transmit?OctetRing:Octet;
217                         psap!req!ed;
218                         psap!conf;
219                         exit
220                 []
```

```
                [Node ne station_1] ->
                    exit
endproc (* Monitor *)

(*********************************************************)
(*                      Manager                          *)
(*********************************************************)
(*            sends and receives frames                  *)
(*********************************************************)
process Manager[msap,psap](Node:Octet) : noexit :=
    (   psap!ind!repeat?Data:Octet[Data eq sd];
        (   psap!ind!transmit?AC:Octet[not(is_set_bit_T(AC))];
            (   (   exit(nothingToSend,Octet(0,0,0,0,0,0,0,0),
                                Octet(Octet(0,0,0,0,0,0,0,0)))
                [>
                    msap!req?Dest:Octet?UserData:OctetString;
                    exit(somethingToSend,Dest,UserData)
                )
                >> accept
                    LLCreq:LLCrequest,
                    Dest:Octet,
                    UserData:OctetString
                in
                (   [LLCreq eq nothingToSend] ->
                        psap!req!AC;
                        psap!conf;
                        exit
                []
                    [LLCreq eq somethingToSend] ->
                        psap!req!setBit_T(AC);
                        psap!conf;
                        SendFrame [msap,psap] (Node,Dest,UserData)
                )
            )
        []
            psap!ind!transmit?AC:Octet[is_set_bit_T(AC)];
            psap!req!AC;
            psap!conf;
            psap!ind!repeat?FC:Octet;
            psap!ind!repeat?Dest:Octet;
            (   [Dest eq Node] ->
                    CopyFrame[msap,psap]
            []
                [Dest ne Node] -> exit
            )
        )
    []
        psap!ind!repeat?Data:Octet[Data ne sd];
        exit
    )
    >> Manager[msap,psap] (Node)

where
    type LLCrequestType is TwoValuesEnum renamedby
        sortnames LLCrequest for Values
        opnames
```

```
277                     nothingToSend for val1
278                     somethingToSend for val2
279                 endtype
280
281             (*************************************************************)
282             (*                          CopyFrame                        *)
283             (*************************************************************)
284             (* A frame has been recognized and is copied, while it passes through, to be  *)
285             (* eventually transferred to the LLC (if there is enough place in LLC buffer) *)
286             (*************************************************************)
287             process CopyFrame[msap,psap] : exit :=
288                 psap!ind!repeat?Source:Octet;
289                 (   CopyOctets[psap] (userDataLength)
290                     >> accept Data:OctetString in
291                     (   CopyOctets[psap] (fcsLength)
292                         >> accept FCS:OctetString in
293                         (   psap!ind!repeat?ED:Octet;
294                             psap!ind!transmit?FS:Octet;
295                             (   (   exit(BufferFull)
296                                     [>
297                                     msap!ind!Source!Data;
298                                     exit(Copied)
299                                 )
300                                 >> accept BState:BufferState in
301                                 (   [BState eq BufferFull] ->
302                                         psap!req!setBit_A(FS);
303                                         psap!conf;
304                                         exit
305                                 []
306                                     [BState eq Copied] ->
307                                         psap!req!setBits_AC(FS);
308                                         psap!conf;
309                                         exit
310                                 )
311                             )
312                         )
313                     )
314                 )
315
316             where
317                 type BufferStateType is TwoValuesEnum renamedby
318                     sortnames BufferState for Values
319                     opnnames
320                         BufferFull for val1
321                         Copied for val2
322                 endtype
323             endproc (* CopyFrame *)
324
325             process CopyOctets[psap] (DataLength:Nat) : exit(OctetString) :=
326                 Copy[psap] (DataLength,<>)
327                 >>
328                 accept Data:OctetString in
329                 exit(Data)
330
331             where
332                 process Copy[psap] (Length:Nat,Data:OctetString) : exit(OctetString) :=
```

```
333                         [Length eq Succ(0)] ->
334                             psap!ind!repeat?X:Octet;
335                             exit(X + Data)
336                     []
337                         [Length gt Succ(0)] ->
338                             psap!ind!repeat?X:Octet;
339                             Copy[psap](dec(Length),X + Data)
340             endproc (* Copy *)
341     endproc (* CopyOctets *)
342
343     process SendFrame[msap,psap]
344                         (   Node:Octet,
345                             Dest:Octet,
346                             UserData:OctetString) : exit :=
347         hide sdIsBack in
348             TransmitFrame[psap,sdIsBack] (Node,Dest,UserData)
349         |[sdIsBack]|
350             RemoveFrame[msap,psap,sdIsBack]
351
352     where
353         (*************************************************)
354         (*                   TransmitFrame               *)
355         (*************************************************)
356         process TransmitFrame[psap,sdIsBack]
357                         (   Node:Octet,
358                             Dest:Octet,
359                             UserData:OctetString) : exit :=
360             psap!req!Octet(0,0,0,0,0,0,0,0); (* FC: not used *)
361             psap!conf;
362             psap!req!Dest;
363             psap!conf;
364             psap!req!Node;          (* local node *)
365             psap!conf;
366             TransmitOctets[psap](UserData,userDataLength) >>
367             TransmitOctets[psap](fcs,fcsLength) >>
368             psap!req!ed;
369             psap!conf;
370             psap!req!Octet(0,0,0,0,0,0,0,0); (* FS, bits A and C equal 0 *)
371             psap!conf;
372             WaitForSD[psap,sdIsBack] >>
373             PutToken[psap]
374
375         where
376             process TransmitOctets[psap] (Data:OctetString, Length:Nat) : exit :=
377                         [Length eq Succ(0)] ->
378                             psap!req!first(Data);
379                             psap!conf;
380                             exit
381                     []
382                         [Length gt Succ(0)] ->
383                             psap!req!first(Data);
384                             psap!conf;
385                             TransmitOctets[psap](tail(Data),dec(Length))
386             endproc (* TransmitOctets *)
387
388             (*****************************************************)
```

```
389                 (*                          WaitForSD                                  *)
390                 (*******************************************************)
391                 (*  If a node finishes transmitting the entire frame prior to          *)
392                 (*  receiving its own starting delimiter, it continues to              *)
393                 (*  transmit idle characters (contiguous 0-bits) on psap until         *)
394                 (*  the header is recognized by Remove (action sdIsBack)               *)
395                 (*******************************************************)
396                 process WaitForSD[psap,sdIsBack] : exit :=
397                     (   exit(sdStillComing)
398                         [>
399                         sdIsBack; exit(sdArrived)
400                     )
401                     >> accept SDst:SDstate in
402                     (   [SDst eq sdStillComing] ->
403                             psap!req!idle;
404                             psap!conf;
405                             WaitForSD[psap,sdIsBack]
406                     []
407                         [SDst eq sdArrived] ->
408                             exit
409                     )
410
411                 where
412                     type SDstateType is TwoValuesEnum renamedby
413                         sortnames SDstate for Values
414                         opnnames
415                             sdStillComing for val1
416                             sdArrived for val2
417                     endtype
418                 endproc (* WaitForSD *)
419
420                 process PutToken[psap] : exit :=
421                     psap!req!sd;
422                     psap!conf;
423                     psap!req!Octet(0,0,0,0,0,0,0,0);          (* AC: token bit = 0 *)
424                     psap!conf;
425                     psap!req!ed;
426                     psap!conf;
427                     exit
428                 endproc (* PutToken *)
429             endproc (*TransmitFrame*)
430
431             (*************************************************)
432             (*                      RemoveFrame                                  *)
433             (*************************************************)
434             process RemoveFrame[msap,psap,sdIsBack] : exit :=
435                     psap!ind!transmit?Data:Octet[Data eq sd];
436                     (   sdIsBack;    (* tell to TransmitFrame that it can *)
437                         exit         (* put back the token on the ring    *)
438                     |||
439                         psap!ind!transmit?AC:Octet;
440                         psap!ind!transmit?FC:Octet;
441                         psap!ind!transmit?Dest:Octet;
442                         psap!ind!transmit?Source:Octet;
443                         WaitForED[msap,psap]
444                     )
```

```
445                             []
446                                 psap!ind!transmit?Data:Octet[Data ne sd];
447                                 RemoveFrame[msap,psap,sdIsBack]
448
449                         where
450                             process WaitForED[msap,psap] : exit :=
451                                     psap!ind!transmit?Data:Octet[Data eq ed];
452                                     psap!ind!transmit?FS:Octet;
453                                     msap!conf!FS;
454                                     exit
455                                 []
456                                     psap!ind!transmit?Data:Octet[Data ne ed];
457                                     WaitForED[msap,psap]
458                             endproc (* WaitForED *)
459                         endproc (* RemoveFrame *)
460                     endproc (* SendFrame *)
461                 endproc (* Manager *)
462             endproc (* MAC *)
463
464             (*************************************************************)
465             (*                        Physical Layer                     *)
466             (*************************************************************)
467             (*  Depending on the mode, the Physical Layer repeats or     *)
468             (*  transmits an octet.                                      *)
469             (*************************************************************)
470             process PhyLayer [psap,medium] (Node :Octet) : noexit :=
471                 RIU[psap,medium](Node)
472
473             where
474                 (*********************************************************)
475                 (*                          RIU                          *)
476                 (*********************************************************)
477                 (*  Ring interfaces have two operating modes, repeat and transmit.  *)
478                 (*  In repeat mode, the input octets are simply copied onto psap    *)
479                 (*  and repeated on medium.                                         *)
480                 (*********************************************************)
481                 process RIU[psap,medium](Node: Octet) : noexit :=
482                     medium!ind!Node?Data:Octet;
483                     (   psap!ind!repeat!Data;       (* repeat state *)
484                         exit(Data)
485                     []
486                         psap!ind!transmit!Data;     (* transmit state *)
487                         psap!req?Data:Octet;
488                         psap!conf;
489                         exit(Data)
490                     ) >> accept Data:Octet in
491                     medium!req!next(Node)!Data;
492                     RIU[psap,medium](Node)
493                 endproc (* RIU *)
494             endproc (* PhyLayer *)
495         endproc (* Station *)
496
497         (*****************************************************************)
498         (*                            Medium                             *)
499         (*****************************************************************)
500         (*  Connections between the RIUs                                 *)
```

```
501     (*    Delay: one octet                                                    *)
502     (*************************************************************)
503     process Medium[medium](Node:Octet) : noexit :=
504         medium!req!Node?Data:Octet;
505         medium!ind!Node!Data;
506         Medium[medium](Node)
507     endproc (* Medium *)
508
509 endspec (* TokenRingProtocol *)
```

Appendix 2. A (Part of) Simulation Session Using ISLA

In this annotated script the actions that were chosen are indicated by *. Annotations are related to the chosen actions. The notation bh[i] refers to the behavior given in the i-th line of the specification text.

1. Station_1 gets the token first

```
==========================================================================================
Level/28
   <3>- i (specified explicitly) ---> bh3 [157]
*  <4>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh4 [165]
   <5>- i (hiding: psap !ind !transmit:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet
              [not(is_set_bit_T(Octet(0,0,0,0,0,0,0,0)))] ) ---> bh5 [232,486]
                                                        (* tell LLC that there is something to send *)
 Passed evaluated value ==> req
 Enter a value for Dest:Octet => station_2
 Enter a value for Data:OctetString => Octet(Octet(1,1,1,1,1,1,1,1))
==========================================================================================
Level/29
   <3>- i (specified explicitly) ---> bh3 [157]
*  <4>- i (hiding: psap !ind !transmit:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet
              [not(is_set_bit_T(Octet(0,0,0,0,0,0,0,0)))] ) ---> bh4 [232,486]
                                                        (* token is coming *)
==========================================================================================
Level/30
   <3>- i (specified explicitly) ---> bh3 [157]
   <4>- i (hiding: psap !req !Octet(0,0,0,0,0,0,0,0):Octet) ---> bh4 [245,487]
*  <5>- i (hiding: msap !req:Connection !Octet(0,0,0,0,0,0,1,0):Octet !Octet(1,1,1,1,1,1,1,1)+ <>:OctetString)
               ---> bh5 [172,236]
                                        (* MAC has something to send. Actions at <4> and <5> are exclusive *)
==========================================================================================
Level/31
   <3>- i (specified explicitly) ---> bh3 [157]
*  <4>- i (hiding: psap !req !Octet(0,0,0,1,0,0,0,0):Octet) ---> bh4 [250,487]
                                        (* Set bit T to 1 to indicate the beginning of a frame *)
```

2. Entering Transmit Mode

```
==========================================================================================
Level/6
   <1>- i (specified explicitly) ---> bh1 [157]
   <2>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh2 [165]
*  <3>- i (hiding: psap !ind !transmit:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet) ---> bh3 [210,486]
                                                        (* Octet sent up to MAC *)
==========================================================================================
Level/7
   <1>- i (specified explicitly) ---> bh1 [157]
```

<2>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh2 [165]
* <3>- i (hiding: psap !req!Octet(1,1,0,1,1,0,0,0):Octet) ---> bh3 [211,487]
(* Transmit mode requires one octet from MAC to be sent on medium *)

==========

Level/8
<1>- i (specified explicitly) ---> bh1 [157]
<2>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh2 [165]
* <3>- i (hiding: psap !conf ---> bh3[212]

3. Entering Repeat Mode

==========

Level/8
<3>- i (specified explicitly) ---> bh3 [157]
<4>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh4 [165]
* <6>- i (hiding: psap !ind !repeat:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet [ne(idle,sd)]) ---> bh6 [268,483]
(* The octet is sent up to MAC and rewritten onto medium *)

4. Release of Token

==========

Level/112
<3>- i (specified explicitly) ---> bh3 [157]
* <5>- i (hiding: psap !ind !transmit:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet [ne(first(fcs),ed)]) --->
bh5 [456,486]

==========

Level/117
<3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !req !Octet(1,1,0,1,1,0,0,0):Octet) ---> bh4 [421,487]

==========

Level/118
<3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !conf ---> bh4[422]

==========

Level/129
<3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !ind !transmit:RIUstate !Octet(1,1,1,1,1,1,0,0):Octet [eq(ed,ed)]) ---> bh4 [451,486]

==========

Level/130
<3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !req !Octet(0,0,0,0,0,0,0,0):Octet) ---> bh4 [423,487]

==========

Level/131
<3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !conf ---> bh4[424]

==========

Level/139
<3>- i (specified explicitly) ---> bh3 [157]

```
* <4>- i (hiding: psap !ind !transmit:RIUstate !Octet(1,1,0,0,1,1,0,0):Octet) ---> bh4 [442,486]
==========================================================================================
Level/140
   <3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !req !Octet(1,1,1,1,1,1,0,0):Octet) ---> bh4 [425,487]
==========================================================================================
Level/141
   <3>- i (specified explicitly) ---> bh3 [157]
* <4>- i (hiding: psap !conf ---> bh4[426]
==========================================================================================
```

5. Message Received at Gate lsap

```
==========================================================================================
Level/28
   <3>- i (specified explicitly) ---> bh3 [157]
* <4>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh4 [165]
   <5>- i (hiding: psap !transmit:RIUstate !Octet(0,0,0,0,0,0,0,0):Octet
              [not(is_set_bit_T(Octet(0,0,0,0,0,0,0,0)))] ) ---> bh5 [232,486]
 Passed evaluated value ==> req
 Enter a value for Dest:Octet => station_2
 Enter a value for Data:OctetString => Octet(Octet(1,1,1,1,1,1,1,1))
                                                    (* A request to send is issued from station_1 *)

==========================================================================================
Level/120
   <3>- i (specified explicitly) ---> bh3 [157]
* <4>- lsap !ind:Connection !Octet(0,0,0,0,0,0,0,1):Octet !Octet(1,1,1,1,1,1,1,1)+ <>:OctetString ---> bh4 [160]
                                                    (* An indication that a frame has arrived from station_1 *)
```

6. Communication between LLC and MAC

```
==========================================================================================
Level/30
   <3>- i (specified explicitly) ---> bh3 [157]
   <4>- i (hiding: psap !Octet(0,0,0,0,0,0,0,0):Octet) ---> bh4 [245,487]
* <5>- i (hiding: msap !req:Connection !Octet(0,0,0,0,0,0,1,0):Octet !Octet(1,1,1,1,1,1,1,1)+ <>:OctetString)
              ---> bh5 [172,236]
                                    (* A request to send from LLC to MAC of station_1. Destination: station_2 *)
==========================================================================================
Level/119
   <4>- i (specified explicitly) ---> bh4 [157]
   <5>- lsap !req:Connection ?Dest:Octet ?Data:OctetString ---> bh5 [165]
* <7>- i (hiding: msap !ind:Connection !Octet(0,0,0,0,0,0,0,1):Octet !Octet(1,1,1,1,1,1,1,1)+ <>:OctetString)
              ---> bh7 [159,297]
                                    (* An indication on msap that frame has arrived from station_1 *)
                                    (* Action at <7> indicates that LLC is ready to buffer octets to form a frame *)
```