Int J STTT (1998) 2: 68–88

Regular contribution

Verification of the Link layer protocol of the IEEE-1394 serial bus (FireWire): an experiment with E-LOTOS

Mihaela Sighireanu, Radu Mateescu

INRIA Rhône-Alpes and DYADE / VASY group, 655, avenue de l'Europe, F-38330 Montbonnot St. Martin, France;
`http://www.inrialpes.fr/vasy`, E-mail: {Mihaela.Sighireanu,Radu.Mateescu}@inria.fr

Abstract. The IEEE-1394 Standard ("FireWire") describes a high-speed serial bus for multimedia PCs intended to unify several serial buses such as VME, MULTIBUS II, and FUTURE BUS. This paper deals with the formal description of the LINK layer protocol of the IEEE-1394 Standard in the formal description techniques E-LOTOS and LOTOS, and its subsequent verification using model checking. The E-LOTOS descriptions are based on both the standard and the μCRL descriptions written by Luttik. The verifications are performed using the CADP (CÆSAR/ALDÉBARAN) toolbox. As a preliminary step, the E-LOTOS descriptions are translated, using the TRAIAN compiler, in LOTOS, which is the input language for the CADP toolbox. Five correctness criteria stated in natural language by Luttik are formalized in the action-based temporal logic ACTL. These logic formulas assess safety and liveness properties of the LINK layer protocol, such as deadlock freedom, appropriate sequencing of messages, etc. To contain the state space explosion, the verification is carried out only for a few scenarios involving a restricted number of nodes connected to the bus and a fixed number of transaction requests per node. Even under these restrictions, an undesirable deadlock situation in the protocol is detected. The error is caused by the ambiguous semantics of the state machines given in the standard, and it can be misleading for implementors of the IEEE-1394 protocol. A correction of this deadlock is proposed and validated.

Key words: Datagram protocol – E-LOTOS – Formal methods – Formal description techniques – IEEE-1394 – Labeled transition systems – LOTOS – Protocol engineering – Temporal logic – Verification

1 Introduction

The design and development of complex, critical applications such as distributed systems and communication protocols are difficult tasks requiring a careful methodology in order to avoid errors as far as possible.

A real-life example of such application is the "FireWire" high performance serial bus defined in the IEEE-1394 Standard [17]. The bus involves n nodes (addressable entities that run their own part of the protocol) connected by a serial line. On each node the IEEE-1394 protocol consists of three stacked layers: the transaction layer, the link layer, and the physical layer. The protocol implemented by the link layer is designed to transmit data packets over an unreliable medium to a specific node or to all nodes (broadcast), and it is similar to an "acknowledged datagram" protocol. The transmission can be performed synchronously or asynchronously. The desired functioning of the link layer protocol can be characterized by means of several correctness properties [22]: the protocol has to be free of deadlocks; at a given time, the protocol should allow only a single link layer to send a packet over the bus; a non-broadcast packet should always be acknowledged; a packet requiring an immediate response must give priority over the bus to its destination node, etc.

One approach that proved its usefulness in the design of critical applications is the use of formal methods throughout the design process, by means of specialized tools. For this purpose, the application must be described using an appropriate high-level language such as μCRL[1] [13], LOTOS[2] [18], or E-LOTOS[3] [28]. Such descriptions provide a formal, unambiguous basis upon which the verification of the desired correctness properties can be attempted.

A verification method that has been extensively studied over the last few years, and for which various algorithms and tools have been developed, is called *model checking*. In this approach, the correctness properties

[1] *micro* Common Representation Language
[2] Language Of Temporal Ordering Specification
[3] Extended LOTOS

are verified on a model automatically generated from the high-level description of the application under design. Although restricted to finite-state systems, model checking provides a simple, efficient way to detect errors in the early steps of the design process.

This paper aims to show the adequacy of the E-LOTOS language and the CADP toolbox for the verification of industrial applications. We illustrate this by analyzing the protocol of the asynchronous transmission mode of the link layer described in the IEEE-1394 Standard. The E-LOTOS [28] language is a new generation FDT[4] that extends the FDT LOTOS [18]. The CADP toolbox [7, 10, 11] is dedicated to the design and verification of communication protocols and distributed systems. Since CADP supports LOTOS as input language, we use the TRAIAN [32] compiler to translate E-LOTOS descriptions into LOTOS ones. To express and verify the correctness properties of the protocol, we use the ACTL [25] temporal logic, for which a model-checker is available within the XTL [23] tool of CADP.

Using this methodology, we were able to identify and correct an undesirable deadlock occurring in the protocol. These results are encouraging, since they show the effectiveness of the approach in the framework of real-life industrial applications.

The paper is organized as follows. Section 2 introduces briefly the LOTOS and E-LOTOS languages. Section 3 gives an informal presentation and an E-LOTOS description of the IEEE-1394 three-layered architecture. Sections 4, 5, 6, and 7 contain informal presentations and E-LOTOS descriptions of the data types, the TRANS layer, the LINK layer, and the BUS layer, respectively. Section 8 introduces the CADP protocol engineering toolbox. Section 9 presents the generation of the LTS models corresponding to the E-LOTOS descriptions. Sections 10 and 11 describe the correctness properties and their verification on the LTS models, respectively. Section 12 gives some concluding remarks. The complete E-LOTOS descriptions of data types, LINK layer, and BUS layer are given in Annexes A, B, and C, respectively.

2 The ISO language LOTOS and the E-LOTOS language

LOTOS [18] is a standardized Formal Description Technique intended for the specification of communication protocols and distributed systems. Several tutorials for LOTOS are available, e.g. [2, 31].

The design of LOTOS was motivated by the need for a language with a high abstraction level and a strong mathematical basis, which could be used for the description and analysis of complex systems. As a design choice, LOTOS consists of two "orthogonal" sublanguages:

The data part of LOTOS is dedicated to the description of data structures. It is based on the well-known theory of algebraic abstract data types [14], more specifically on the ACTONE specification language [6].

The control part of LOTOS is based on the process algebra approach for concurrency, and appears to combine the best features of CCS [24] and CSP [16].

LOTOS has been applied to describe complex systems formally, for example OSI TP[5] [19, Annex H] and FTAM[6] basic file protocol [21]. It has been mostly used to describe software systems, although there have been recent attempts to use it for asynchronous hardware description [4].

A number of tools have been developed for LOTOS, covering user needs in the areas of simulation, compilation, test generation, and formal verification.

Despite these positive features, a revision of the LOTOS standard has been undertaken within ISO since 1993, because feedback from users indicated that the usefulness of LOTOS is limited by certain characteristics related both to technical capabilities and user-friendliness of the language.

The ISO Committee Draft [28] appeared in May 1998 and proposes a revised version of LOTOS, named E-LOTOS. Compared to LOTOS, the language defined in [28] introduces new features, from which we mention only those used in this case study:

- Modularity: an E-LOTOS module is a collection of types, functions and/or process definitions, the visibility of which can be controlled by interface declaration; modules may be combined using importation and renaming.
- Data types: types are defined in a functional style; in addition, many useful types are predefined.
- Sequential composition operator: the action prefix, enabling, and 'accept' operators of LOTOS have been substituted with a new, simpler sequential composition operator.
- 'If-then-else' operator: to express conditional constructs, an explicit 'if-then-else' operator has been introduced instead of guarded commands combined with choice.
- Imperative features: to allow an imperative-like programming style, write-many variables as well as functions and processes with in/out parameters have been introduced. These features try to align E-LOTOS notations with standard programming languages.
- Gate typing: gates must be explicitly typed [9].

This case study is based on a version of E-LOTOS described in [29, 30], which is slightly different from the Committee Draft one. The two main differences are: (a) instead of record subtyping and anonymous records, we use named records and overloading of functions and constructors; (b) although [28] introduces quantitative time, the fragment of E-LOTOS we consider here is untimed.

[4] Formal Description Technique

[5] Distributed Transaction Processing

[6] File Transfer, Access, and Management

To our knowledge, at the present time, there exists only one realistic experiment with E-LOTOS, namely the description of the ODP trader computational viewpoint [20] given in [28]. Thus, the case study presented here can be considered as a pioneering attempt at using E-LOTOS for the description and verification of a real application.

Since E-LOTOS is currently under balloting within ISO, there are no tools for E-LOTOS available yet. A straightforward approach is to translate E-LOTOS programs into LOTOS, and then use the existing tools dedicated to LOTOS. For this case study, we used the TRAIAN tool [32], a prototype translator from E-LOTOS to LOTOS, and the CADP toolbox [7, 10, 11], which provides state-of-the-art verification features.

3 The IEEE-1394 architecture

The IEEE's Microcomputer Standards Committee started to work in 1986 on the unification of several serial buses such as VME, MULTIBUS II, and FUTURE BUS. This effort led to a new serial bus protocol defined in the IEEE-1394 Standard [17].

We summarize below some important features of this protocol. An extended presentation is given in [22]. The IEEE-1394 architecture involves n nodes (addressable entities that run their own part of the protocol) connected by a serial line (referred to as the CABLE in the sequel). On each node the IEEE-1394 protocol consists of three stacked layers:

- The upper layer, or *transaction layer* (referred to as TRANS), implements the request-response protocol required to conform to the standard Control and Status Register Architecture for Microcomputer Buses [1]. It provides the applications running on the node with read, write, and lock transaction services.
- The middle layer, or *link layer* (referred to as LINK), provides an acknowledged datagram service to the transaction layer. It handles all packet transmission and reception, as well as cycle control for isochronous channels.
- The lower layer, or *physical layer* (referred to as PHY), provides the initialization and arbitration services necessary to ensure that only one node at a time is sending data. It also converts the serial bus data streams and electrical signals to those required by the LINK layer.

  In the sequel, we denote by BUS the PHY layer together with the CABLE.

According to [17], there is also a so-called node controller, which provides facilities for timeout control and reset procedures for all the three layers above. As in [22], we leave these facilities out of our E-LOTOS specification.

The architecture of the IEEE-1394 serial bus is depicted in Fig. 1 and described by the E-LOTOS specification module `IEEE_1394` below. This module includes (using the `import` clause) the modules defining the data types (`DATA`), the TRANS layer (`TRANS`), the LINK layer (`LINK`), and the BUS layer (`BUS`).

```
specification IEEE_1394
  import DATA, TRANS, LINK, BUS
is
  gates LDreq: LDreqType, LDcon: LDconType,
        LDind: LDindType, LDres: LDresType,
        PDind: PDindType, PDreq: PDreqType,
        PAcon: PAconType, PAreq: PAreqType,
        PCind: Nat, arbresgap, losesignal: none
  behaviour
    (    (Trans [...] (n, 0)
          |[LDreq, LDcon, LDind, LDres]|
          Link  [...] (n, 0))
     |||
     (* nodes 1 .. n-2 *)
     |||
         (Trans [...] (n, n-1)
          |[LDreq, LDcon, LDind, LDres]|
          Link  [...] (n, n-1))
    )
    |[PDind, PDreq, PAcon, PAreq, PCind]|
    Bus [...] (n)
endspec
```

The module declares the set of typed gates corresponding to the interfaces of the three layers, and to the actions of the BUS layer (`arbresgap` and `losesignal`). The entities depicted in Fig. 1 are represented by E-LOTOS processes evolving in parallel. Each node consists of a `Trans` and `Link` processes running in parallel and synchronizing by means of four gates: `LDreq`, `LDcon`, `LDind`, and `LDres`. The set of `n` nodes (indexed from `0` to `n-1`) is synchronized with the `Bus` process by means of five gates: `PDind`, `PDreq`, `PAcon`, `PAreq`, and `PCind`. Note the use of the shorthand notation '`...`' in the instantiation of the `Trans`, `Link`, and `Bus` processes. This notation allows the actual gate or value parameters to be elided when they

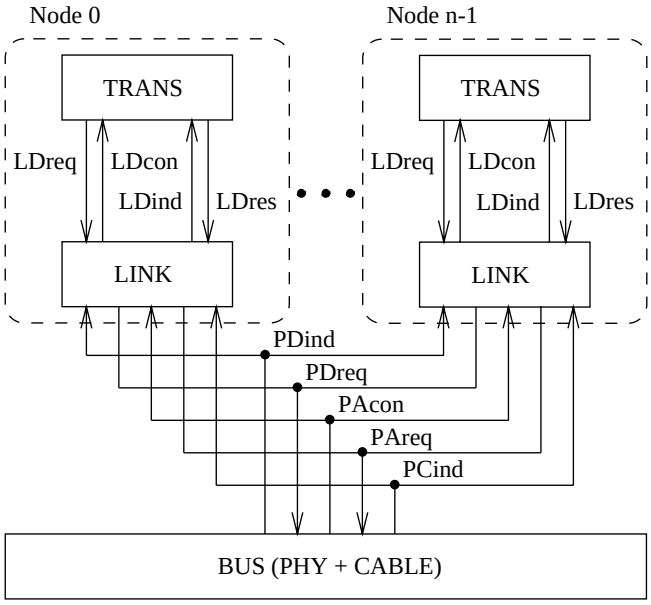

Fig. 1. The serial bus architecture

are identical to the formal ones defined in the corresponding process declaration.

The following sections refine this top-level architecture by describing the data types used in the protocol and the `Trans`, `Link`, and `Bus` processes.

4 The data types

The E-LOTOS data types used in the protocol are grouped into the `DATA` module whose detailed description is given in Annex A.

Here we give only an informal description of these data types:

- `Nat` and `Bool` are the E-LOTOS predefined types for natural numbers and booleans.
- `ACK`, `DATA`, and `HEADER` are enumerated types, representing the acknowledgement, data, and header part of the packets; `BOC` and `PHY_AREQ` are enumerated types, representing the bus occupancy control code (which may be `hold`, `release`, or `no_op`) and the physical acknowledge request code (which may be `fair` or `immediate`).
- `SIGNALS` is a union type modeling the data packet components[7] and the control signals traveling over the BUS.
- `SIG_TUPLE` is the type of the one place buffer used by the LINK of each node; it denotes either a quadruple of signals, or an empty buffer (value `void`).
- `LIN_DIND` is a union type used to attach an indication attribute to the data packets received by TRANS.
- `BoolTABLE` is a list of `n` natural-boolean pairs, implementing a boolean array indexed by the node identifiers.
- To each gate, we attach a tuple type whose fields correspond to the values exchanged during the rendezvous.

All the data types described above are finite, except for `Nat`. However, the integer numbers actually used in the protocol range between `0..n`, where `n` is the number of nodes connected to the BUS. Therefore, to obtain a finite state representation of the protocol that is suitable for analytic purposes, it is sufficient to fix the value of `n` (see also Sect. 9.2).

We must underline here the conciseness of the E-LOTOS data language: the data types of the protocol are described in four pages of E-LOTOS instead of eight pages of algebraic data types in the μCRL description given in [22]. This conciseness is mainly due to the functional style of the E-LOTOS data types specification: since the types are specified using constructors, some functions may be automatically defined for each type and the pattern-matching can be used to specify user-defined functions. As an example, consider the E-LOTOS function below, which tests if its parameter is a destination signal:

```
function is_dest (x: SIGNALS) : Bool is
   case x is
     sig (any of DESTSIG) -> return true
   | any -> return false
   end case
end function
```

This function can be described in μCRL as follows:

```
func is_dest : SIGNALS -> Bool
var xdest: DESTSIG xsigh: HEADERSIG
    xsigd: DATASIG xsiga: ACKSIG
rew is_dest (sig (xdest)) = T
    is_dest (sig (xsigh)) = F
    is_dest (sig (xsigd)) = F
    is_dest (sig (xsiga)) = F
    is_dest (Start)       = F
    is_dest (EndS)        = F
    is_dest (Prefix)      = F
    is_dest (subactgap)   = F
    is_dest (Dummy)       = F
    is_dest (dhead)       = F
```

This more verbose definition (the corresponding LOTOS code is very similar) is due to the algebraic style of the μCRL and LOTOS data languages, which does not make any difference between constructor and non-constructor operations, and considers that all the equations have the same priority.

5 The TRANS layer behavior

In order to verify the LINK layer protocol, we had to specify the external behavior of TRANS with respect to LINK, although it was not formalized in [22]. Our description of the TRANS behavior is based on the state machine diagrams and informal explanation given in [17, Fig. 7-2, p. 184].

For each node, the TRANS layer provides read, write and lock transactions to the application running on the node. Transactions use four service primitives of the LINK layer, following to the OSI connection establishment diagram (shown in Fig. 2):

- *Request* (performed on the `LDreq` gate) is used by a TRANS responder to start an incoming transaction;
- *Indication* (performed on the `LDind` gate) is used to notify the TRANS requester of an incoming request;
- *Response* (performed on the `LDres` gate) is used by the TRANS requester to return status or data to the TRANS responder;
- *Confirmation* (performed on the `LDcon` gate) is used to notify the TRANS responder of the arrival of the corresponding response.

At any time, the TRANS entity of a node can process outgoing (request) and incoming (response) transactions. The TRANS behavior is defined by the E-LOTOS module below. The requester (`TransReq`) and the responder (`TransRes`) processes, which handle the outgoing and incoming transactions, respectively, evolve in parallel

[7] According to the IEEE-1394 Standard, the packets are transmitted over the BUS serially, one bit at a time. However, to obtain a more concise description, we model with signals the bit sequences corresponding to the same packet component.

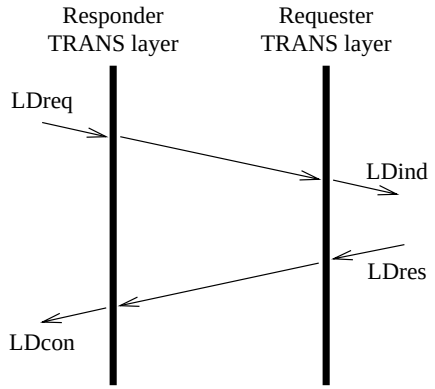

Fig. 2. The diagram of the TRANS layer

and synchronize on the hidden `TX0` gate. To improve readability, we use the `<<Trans gates>>` shorthand notation for the following list of typed gates: `LDreq: LDreqType`, `LDcon: LDconType`, `LDind: LDindType`, `LDres: LDresType`. The default type of the `TX0` gate is "`none`". Each process is parameterized by the total number of nodes `n` and the identification number of its node `id`.

```
module TRANS import DATA is
  process Trans [<<Trans gates>>] (n, id: Nat) is
    hide TDreq: LDreq, TX0 in
      TransRes [...] (n, id)
      |[TX0]|
      TransReq [...] (n, id)
    end hide
  end process
  (* processes TransRes and TransReq *)
end module
```

In the remainder of this section, we present the `TransRes` and `TransReq` processes, considering only the part of the TRANS behavior that is relevant to the LINK.

Request transaction. The requester awaits an indication of a transaction request on the `LDind` gate. If a broadcast is indicated, the requester sends a response to the LINK by means of an `LDres` action with parameter `no_op`. Otherwise, the TRANS may either respond immediately (*concatenated transaction*) or defer the response (*split transaction*). A concatenated transaction is possible only if the responder is in its initial state (this is ensured by the synchronization on the `TX0` gate), and a link data response with parameter `hold` is sent. For a split transaction, a link data response with parameter `release` is sent. After sending a link data response, the requester returns to its initial state.

```
process TransReq [LDreq:LDreqType,LDcon:LDconType,
                  TX0:none] (n, id: Nat) is
  loop
    var l: LIN_DIND in
      LDind (!id, ?l);
        if is_broadrec (l) then
          LDres (!id, ?any, !no_op)
        else
          (* concatenated transaction *)
          TX0;
            LDres (!id, ?any, !hold)
          []
          (* split transaction *)
          LDres (!id, ?any, !release)
        end if
    end var
  end loop
end process
```

For comparison, we give below the descriptions in LOTOS and μCRL of the `TransReq` process. Compared to the LOTOS process definition below, the E-LOTOS description above presents several advantages.

```
process TransReq [LDreq, LDcon, TX0]
                 (n, id: Nat) : noexit
:=
  LDind !id ?l: LIN_DIND;
    ([is_broadrec (l)] ->
      (choice a: ACK []
         LDres !id !a !no_op;
           TransReq [LDreq,LDcon,TX0](n,id))
     []
     [not (is_broadrec (l))] ->
       (choice a: ACK []
         ((* concatenated transaction *)
          TX0;
            LDres !id !a !hold;
             TransReq [LDreq,LDcon,TX0](n,id)
          []
          (* split transaction *)
          LDres !id !a !release;
            TransReq [LDreq,LDcon,TX0](n,id)
         ))
    )
endproc
```

The gate typing avoids synchronization errors caused by sending or receiving values of different types on a gate. The 'loop' operator and the symmetrical sequential composition operator allow the cyclical behavior of the responder to be described more concisely than using (tail) recursive process calls. The 'if-then-else' operator avoids the double evaluation of the predicate "`is_broadrec (l)`". The use of the '`...`' shorthand notation reduces the verbosity of process calls. The use of 'any' patterns to express reception offers of non-interesting values avoids the use of verbose 'choice' statements or the declaration of dummy variables holding these values (e.g., the variable `a` in the LOTOS process).

Compared to the μCRL description below, the E-LOTOS description of the `TransReq` process presents two other advantages. The iteration over sort domains using the '?' construct is more compact than the `Sum` operator.

```
TransReq (n, id: Nat) =
   sum (l:LIN_DIND,
        LDind (id,l) .
          (sum (a:ACK,
                LDres (id,a,no_op))
           <| is_broadrec (l) |>
           (sum (a:ACK,
                 TX0 . LDres (id,a,hold) .
                   TransReq (n, id))
            +
            sum (a:ACK,
                 LDres (id,a,release) .
                   TransReq (n, id)))
          )
       )
```

The re-usability of processes is stronger, due to the use of gate parameters in E-LOTOS (and LOTOS) instead of the use of the global gates in μCRL.

Response transaction. Incoming transactions arrive on the `TDreq` gate. The request consists of a destination node identifier (`dest`), a header (`h`), and data (`d`). The responder may either signal to the requester (on the `TX0` gate) that it is in the initial state, or may continue (by means of the 'null' construct). After this, it sends the data request to the LINK on the `LDreq` gate and waits for a confirmation on the `LDcon` gate. The confirmation can indicate either a broadcast completion (`broadsent`), an acknowledgement of the request (`ackrec`), or a negative acknowledgement (`ackmiss`). In all cases, the responder returns to its initial state afterwards.

```
process TransRes [TDreq,LDreq:LDreqType,
                  LDcon:LDconType,TX0:none]
                 (n, id: Nat) is
  loop
    var dest:Nat, h:HEADER, d:DATA in
      TDreq (!id, ?dest, ?h, ?d) [dest <= n];
      (TX0 [] null); (* initial state *)
      LDreq (!id, !dest, !h, !d);
        (if (dest == n) then
           LDcon (!id, !broadsent)
         else
           LDcon (!id, ?ackrec (any of ACK))
         end if
         []
         LDcon (!id, !ackmiss))
    end var
  end loop
end process
```

6 The LINK layer behavior

The LINK layer protocol is designed to transmit data packets over an unreliable medium, by splitting them into *signals* that are sent sequentially, asynchronously or isochronously. In this case study we consider only the asynchronous part of the LINK protocol.

The asynchronous LINK protocol provides transmission of a data packet to a precise node or to all nodes (by broadcast). The protocol is similar to an "acknowledged datagram" protocol, since each transmission is one-way and needs a confirmation.

According to [22], we represent each state of the LINK protocol by an E-LOTOS process having (at least) three value parameters: the total number of nodes connected to the BUS, the identification number of its node, and a buffer that may contain one asynchronous packet.

The LINK protocol has three main modes: a *send* mode, a *receive* mode, and a *send acknowledge* mode. From its initial state, LINK can evolve into send or receive modes. In the send mode, after transmitting a packet, LINK waits for an acknowledgement (if the packet is not a broadcast), then returns to its initial state. In the receive mode, LINK indicates to TRANS the receipt of a packet. If the packet is not a broadcast, LINK waits for a confirmation from TRANS containing the type of the acknowledgement to be sent and a control code. After sending the acknowledgement, if the control code indicates that TRANS asks to send an immediate response (*concatenated response* mode of TRANS), LINK evolves into the send mode; if TRANS asks to defer the response (*split response* mode of TRANS), LINK returns to its initial state.

The dependences between the LINK modes, together with the collection of processes implementing the states of each mode, are shown in Fig. 3.

In the remainder of this section, we describe informally the initial state and the three modes of LINK. The LINK behavior is defined by the E-LOTOS module `LINK`, whose detailed description is given in Annex B. To improve readability, we use the shorthand notation `<<Link gates>>` for the following list of typed gates: `LDreq: LDreqType`, `LDcon: LDconType`, `LDind: LDindType`, `LDres: LDresType`, `PDreq: PDreqType`, `PDind: PDindType`, `PAreq: PAreqType`, `PAcon: PAconType`, `PCind: Nat`.

```
module LINK import DATA is
  process Link [<<Link gates>>] (n, id: Nat) is
    Link0 [...] (n, id, void)
  end process
  (* ... other Link processes *)
endmod
```

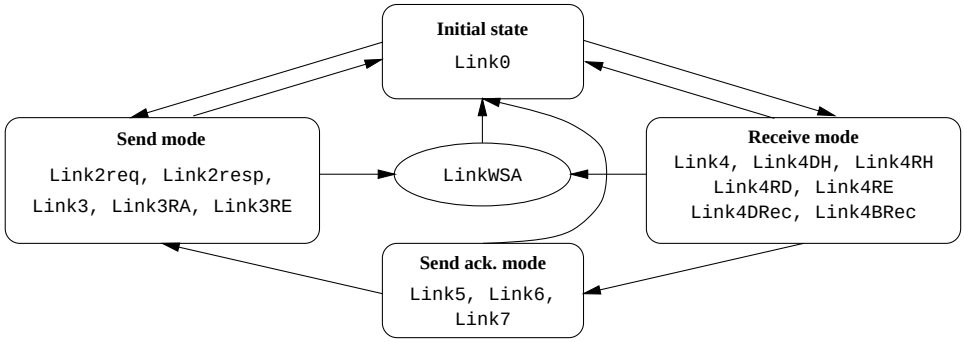

Fig. 3. The LINK behavior

This modeling of the LINK behavior is based on the μCRL description given in [22], the state machine depicted in [17, Fig. 6-19, p. 166], and the explanatory text of the IEEE-1394 Standard.

Initial State. Initially, LINK is in the state `Link0` and has an empty buffer represented by the `void` value; it can receive either a packet from TRANS on the `LDreq` gate, or an indication of a packet arrival from BUS on the `PDind` gate.

In the former case, LINK constructs the packet from the parameters received and puts it in its buffer. The buffer being no longer empty, LINK tries to gain access to BUS by sending a fair arbitration request (`PAreq` action with parameter `fair`) and waits for BUS arbitration response. If BUS responds positively (`PAcon` action with parameter `won`), LINK evolves into the send mode (state `Link2req`). If a negative response is received (`PAcon` action with parameter `lost`), LINK returns to its initial state `Link0`.[8] This behavior induces a livelock (i.e., a circuit made from actions `PAreq` with parameter `fair` and `PAcon` with parameter `lost`) in the LTS model. Notice that this livelock occurs only in our untimed description of the protocol, and not in a real implementation, where the "arbitration reset gap" signal is guaranteed to occur in a finite amount of time.

In the latter case, if the signal received is `Start`, LINK enters into the receive mode (state `Link4`); otherwise LINK ignores the signal and returns to its initial state.

Send Mode. Being granted the access to BUS, LINK responds to every clock indication received on the `PCind` gate by sending a signal on the `PDreq` gate. The packet transmission begins with a `Start` signal, followed by the data packet – split up into four signals: destination header signal, destination signal, header signal, and data signal – and the termination signal `EndS`. Depending on whether the packet sent was a broadcast packet or an asynchronous packet (this can be determined from the destination field of the packet), LINK either confirms to TRANS (on the `LDcon` gate) that a broadcast packet was sent properly and returns to its initial state, or goes to state `Link3` and waits for an acknowledgement packet.

The acknowledgement packet must arrive within some specific amount of time: if a "subaction gap" signal occurs before an acknowledgement with a valid checksum has been entirely received, then LINK will act as if the acknowledgement was missing. The acknowledgement packet begins with a `Start` signal, possibly preceded by any number of `Prefix` signals. When the `Start` signal arrives, LINK evolves into the `Link3RA` ("Receive Acknowledge") state.

In the state `Link3RA`, upon receipt of a data signal (i.e., not a control one), LINK goes into the `Link3RE` ("Receive End") state, where it awaits the terminating signal `EndS`, checks its validity and sends an "acknowledgement received" confirmation (`LDcon` action with parameter `ackrec`) to TRANS. However, if anything goes wrong, LINK sends an "acknowledgement missing" confirmation (`LDcon` action with parameter `ackmiss`) to TRANS. In case of either failure or success, LINK must wait for a "subaction gap" indication from BUS, before returning to its initial state. This is done in the `LinkWSA` state.

Receive Mode. When receiving a `Start` signal, LINK expects an asynchronous packet to be transmitted by another node via BUS. As mentioned already for the send mode, the asynchronous packet consists of exactly four signals, and LINK must receive two signals (on the `PDind` gate) before determining whether the packet is addressed to itself or to another LINK. If anything goes wrong, it goes in the `LinkWSA` state, where it waits for the next "subaction gap" signal and returns to its initial state afterwards.

If the second signal received on the `PDind` gate is a destination signal, then LINK must check whether the incoming packet is either (a) a packet addressed to itself, (b) a broadcast packet, or (c) a packet addressed to another node. In the case (a), it must notify the BUS (by means of a `PAreq` action with parameter `immediate`) that it wants access as soon as the packet has been entirely received, in anticipation of sending the acknowledgement. Broadcast packets should not be acknowledged, so in the case (b) no such request is needed. In both cases, LINK evolves into the state `Link4RH` ("Receive Header"), keeping as parameter the destination of the packet. In the case (c), the packet is not addressed to this LINK, so it is ignored: the LINK will return to its initial state after waiting for a "subaction gap" signal (the `LinkWSA` state).

The third signal on the `PDind` gate is expected to be a header signal (received in the `Link4RH` state), and the fourth signal should be a data signal (received in the `Link4RD` state).

If the packet is correctly terminated by an `EndS` or a `Prefix` signal, then it is indicated to TRANS either as a broadcast packet (state `Link4BRec`) or as a packet addressed to this node (state `Link4DRec`). In both cases, the data checksum is verified. In the second case, the packet has to be acknowledged, so when BUS becomes free (signaled by a `PAcon` action with parameter `won`), LINK evolves into its "send acknowledge mode" (state `Link5`).

Any deviation from the above behavior will cause LINK to ignore the entire packet; it goes into the `LinkWSA` state where it waits for a "subaction gap" signal.

When a broadcast is received by LINK, a link data indication (`LDind` action) is signaled to TRANS, and LINK returns to its initial state. We will see later that this behavior (which follows strictly the state machine given in [17, p. 166] and the μCRL description of [22]) is erroneous.

Send acknowledge mode. While waiting for TRANS to respond to a packet indication, LINK keeps BUS into the

[8] Here, unlike the μCRL description, we merge `Link0` and `Link1` into the same process `Link0`.

"busy" state by sending a `Prefix` signal on every clock indication. Upon receipt on the `LDres` gate of a proper acknowledgement from TRANS (together with an extra `release` or `hold` parameter), LINK evolves into the state `Link6`.

In both `release` and `hold` cases, the acknowledgement is sent. If TRANS indicates that no concatenated response is requested (`release`), LINK releases the BUS and go to its initial state `Link0`. If a concatenated response is requested, LINK evolves in the `Link7` state and holds the BUS by responding to clock indications with a `Prefix` signal.

Since LINK already has control over BUS, upon receipt of a packet from the TRANS via `LDreq`, it may evolve into the send mode (state `Link2resp`) immediately.

The state `Link2resp` differs from the state `Link2req` only by the presence of a non-void buffer, which buffer has to be transmitted into the next fairness interval.

In the `LinkWSA` state, LINK awaits either a "subaction gap" signal from BUS and then evolves into its initial state, or a BUS indication of access granted. This access is due to the immediate arbitration request of LINK. Therefore, if the destination signal indicates that the packet was meant for this LINK, the arbitration confirmation must be received and BUS control must be terminated immediately by sending an `EndS` signal.

7 The BUS layer behavior

In order to model the interactions between the LINK layers of several nodes, we describe in E-LOTOS the external behavior of the BUS using both the μCRL descriptions given in [22] and the IEEE-1394 Standard [17].

The BUS layer has two primary functions: arbitration of LINKs accesses to the CABLE (by means of `PAreq` actions) and transmission/receipt of signals (by means of `PDreq` and `PDind` actions).

The arbitration protocol implemented by the BUS is based on the concept of *fairness interval*, illustrated in Fig. 4.

During a fairness interval, each LINK may send at most one asynchronous data packet over the BUS, but it can send several acknowledgement packets. The time needed for the transmission of a data packet followed by a (possibly empty) sequence of acknowledgement packets is called a *subaction*. A fairness interval may contain one or more subactions delimited by "subaction gap" (`subactgap`) signals sent by the BUS. The fairness intervals are delimited by "arbitration reset gap" (`arbresgap`) signals sent by the BUS. An `arbresgap` signal is emitted when, after some subactions, the BUS has been idle for a specific amount of time.

The transmission protocol we describe below considers an unreliable communication medium (i.e., the signals may be corrupted or lost).

The BUS behavior is defined by the E-LOTOS module `BUS`. The complete E-LOTOS description of this module is given in Annex C. According to [22], we represent each state of the BUS protocol by an E-LOTOS process parameterized (at least) by the total number of nodes `n` connected to the BUS. To improve readability, we use the `<<Bus gates>>` shorthand notation for the following list of typed gates: `PDind: PDindType`, `PDreq: PDreqType`, `PAcon: PAconType`, `PAreq: PAreqType`, `PCind: Nat`, `arbresgap: none`, `losesignal: none`.

```
module BUS import DATA is
  process Bus [<<Bus gates>>] (n: Nat) is
    BusIdle [...] (n, init (n))
  end process
  (* ... other Bus processes *)
end module
```

Idle state. Initially, BUS is idle (state `BusIdle`). The parameter `t` of process `BusIdle` is a boolean table recording the accesses of LINKs during a fairness interval. For each LINK `j`, the corresponding entry `t[j]` is true if the LINK `j` accessed the BUS during the current fairness interval and false otherwise. At the beginning of each fairness interval, all the entries of the table `t` are set to false using the function `init`.

In this state, every time an arbitration request (`PAreq`) with parameter `fair` is received from some LINK, BUS checks whether the LINK accessed it during the present fairness interval;[9] if not, BUS grants access to the requesting LINK by means of the `PAcon` action with parameter `won` and evolves into the `BusBusy` state.

If BUS has been idle for a specific amount of time (longer than the maximal delay for a LINK arbitration

[9] Here, unlike the μCRL description, we merge `BusIdle` and `DecideIdle` into the same process `BusIdle`.

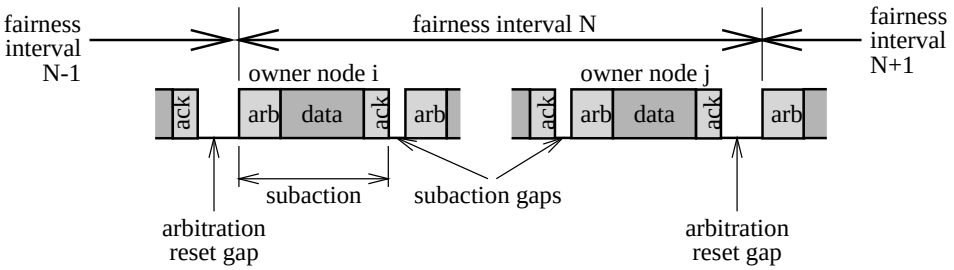

Fig. 4. The structure of the fairness interval

request) and at least one LINK has had access to the BUS (`zero (xbt)` is false), BUS sends an `arbresgap` signal and begins a new fairness interval. Since we use an untimed version of E-LOTOS, the quantitative time constraint above cannot be considered here. The `arbresgap` is an alternative to the `PAreq` action.

Busy state. In the `BusBusy` state, BUS is accessed by a LINK whose identifier is given by the `busy` parameter. Beside the boolean table `t`, which records the LINKs accesses, there are two other boolean table parameters: `next` and `destfault`, with all entries initialized to false. The `next` table records the LINKs having issued an immediate arbitration request (`PAreq` action with parameter `immediate`). The `destfault` table records the LINKs having received a corrupted destination signal; for these LINKs, the checksum of the header signal (which follows the destination signal) has to be invalidated by the BUS (see the `Distribute` state).

In the `BusBusy` state, the BUS may still accept fair arbitration requests from LINKs (`PAreq` actions with parameter `fair`), but sends negative responses to requesters (`PAcon` action with parameter `lost`).

If a LINK asks for immediate arbitration, the BUS records its request in the `next` table and will send the confirmation as soon as the `busy` node will release the BUS.

The `busy` LINK transmits its data packet by splitting it into six signals (i.e., `Start`, destination header, destination, header, data, and `EndS` signals) and sending them sequentially to the BUS upon clock indications (`PCind` actions). Then, the BUS distributes these signals to all the other LINKs (state `Distribute`).

As soon as the `busy` LINK terminates the transmission of its data packet (modeled by setting the `busy` parameter to `n`), the BUS checks the `next` table in order to send confirmations to all the LINKs having issued an immediate arbitration request (state `Resolve`). If the `next` table has all its entries set to false, the BUS indicates the end of the current subaction to all the LINKs by sending a subaction gap signal (state `SubactionGap`) and returns to state `BusIdle` afterwards.

Distribute state. In the `Distribute` state, BUS iterates over all LINKs except the `busy` one. To each LINK (identified by the parameter `j`), BUS delivers the signal (parameter `p`) previously sent by the `busy` LINK.

The signal `p` may be distributed correctly or, due to the unreliable communication medium, it may be corrupted or lost. However, the signal `p` must be corrupted (or lost) if it is a header signal and the current LINK `j` is recorded in the `destfault` table (meaning that the LINK has previously received a corrupted destination signal). The unreliable communication medium is modeled in the following way:

- The corruption of destination signals is modeled by changing their values; in this case, the current LINK `j` is recorded into the `destfault` table.
- The corruption of header, data, and acknowledgement signals is modeled by setting the `crc` field of these signals to `bottom`.
- The loss of header, data, and acknowledgement signals is modeled by a `losesignal` action.
- The corruption of data signals by modification of their length is modeled by immediately sending a `Dummy` signal after the data signal.

At any moment of the distribution, BUS may accept an immediate arbitration request of the current LINK `j`, which is recorded into the `next` table.

After the current signal `p` is distributed to all LINKs (`j` becomes equal to `n`), BUS evolves into the `BusBusy` state, where it awaits another signal to be distributed. If the current signal `p` is `EndS`, which indicates the termination of the asynchronous packet, the parameter `busy` is set to `n`.

Resolve states. The BUS sends a winning arbitration confirmation (`PAcon` action with parameter `won`) and a clock indication to all LINKs that issued an immediate arbitration request.

Then, it evolves into the `Resolve2` state, which is intended to avoid conflicting situations in which several LINKs would have control over the BUS: as long as there are more than one LINKrecorded in the `next` table, BUS accepts only `EndS` signals from these LINKs, and eliminates them from the `next` table. If the remaining LINK sends an `EndS` signal, BUS evolves into the `SubactionGap` state; otherwise BUS delivers the signal to the other LINKs by moving into the `Distribute` state.

In the remainder of the paper, we present the CADP toolbox we used for this case study and we discuss the verification results obtained.

8 The CADP verification toolbox

The CADP[10] toolbox [7, 10, 11] is dedicated to the design and verification of communication protocols and distributed systems. Initiated in 1986, its development has been guided by several motivations:

- This toolbox aims to offer an integrated set of functionalities ranging from interactive simulation to exhaustive, model-based verification methods. In particular, both logical and behavioral specifications can be verified.
- A major objective of the toolbox is to deal with large case studies. Therefore, in addition to enumerative verification methods, it also includes more sophisticated approaches, such as symbolic verification, on-the-fly verification, and compositional model generation.
- Finally, this toolbox can be viewed as an open software platform: in addition to LOTOS, it also supports

[10] CÆSAR/ALDÉBARAN Development Package

lower-level formalisms such as finite state machines and networks of communicating automata.

In the sequel, we only present the CADP tools used throughout this case study:

- CÆSAR [12] and CÆSAR.ADT [8] are compilers that translate a LOTOS program into a Labeled Transition System (LTS for short) exhaustively describing its behavior. This LTS can be represented either *explicitly*, as a set of states and transitions, or *implicitly*, as a library of C functions allowing the program behavior to be executed in a controlled way.
- ALDÉBARAN [3] is a verification tool for comparing or minimizing LTSs with respect to (bi)simulation relations [24, 26]. Initially designed to deal with explicit LTSs produced by CÆSAR, it has been extended to also handle networks of communicating automata (for on-the-fly and symbolic verification). Several simulation and bisimulation relations are implemented within ALDÉBARAN, which offers a wide spectrum for expressing such behavioral specifications.
- XTL (*eXecutable Temporal Language*) [23] is a functional language allowing a compact description of various temporal logic operators to be evaluated over an LTS. The XTL language gives access to all the information contained in the states and labels of an LTS and offers primitives for exploring the transition relation. Temporal logic operators are implemented as recursively defined functions operating on sets of states. A prototype compiler for XTL has been developed, and several temporal logics like CTL [5] and ACTL [25] have already been implemented in XTL.

We also used a new tool (not yet integrated in CADP), called TRAIAN [32]. TRAIAN is a prototype translator from E-LOTOS to LOTOS, which is currently under development. The current version supports a subset of E-LOTOS [30] sufficient for this case study. It translates into LOTOS all the constructs used in this paper, namely declarations (of types, functions, and processes), simple behavior expressions (parallel composition, choice, 'if-then-else', sequential composition, 'var' declarations, actions, process call), and simple data expressions ('if-then-else', values, and operations calls).

9 Model generation

In order to perform verification by model-checking, we generated, using the CADP tools, various LTSs corresponding to the IEEE-1394 protocol.

First, we give the formal definition of the LTS model. Then, we present our experimental results concerning IEEE-1394 model generation.

9.1 The LTS model

According to the operational semantics of LOTOS and untimed E-LOTOS, both LOTOS and E-LOTOS programs can be translated into (possibly infinite) LTSs, which encode all their possible execution sequences. An LTS is formally defined as a quadruple $M = \langle Q, A, T, q_{\text{init}} \rangle$ where:

- Q is the set of *states* of the program;
- A is a set of *actions* performed by the program. An action $a \in A$ is a tuple $G\ V_1, ..., V_n$ where G is a *gate* and V_1, ..., V_n ($n \geq 0$) are the values exchanged (i.e., sent or received) during the rendezvous at G. For the *silent* action τ, the value list must be empty ($n = 0$);
- $T \subseteq Q \times A \times Q$ is the *transition relation*. A transition $\langle q_1, a, q_2 \rangle \in T$ (written also "$q_1 \xrightarrow{a} q_2$") means that the program can move from state q_1 to state q_2 by performing action a;
- $q_{\text{init}} \in Q$ is the *initial state* of the program.

For each state $q \in Q$, we note $Path(q)$ the set of all paths $q(= q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 ...$ issued from q.

9.2 LTS generation for IEEE-1394

In order to generate the LTSs corresponding to the IEEE-1394 E-LOTOS description, we used the following methodology: (1) selection of appropriate abstractions allowing finite LTSs of tractable size to be generated; (2) translation of the E-LOTOS descriptions in LOTOS using the TRAIAN tool; (3) translation of the resulting LOTOS descriptions into LTSs using the CÆSAR and CÆSAR.ADT compilers.

To generate finite LTSs, we restricted to finite sets the domains of all protocol parameters. Therefore, in each experiment we gave a fixed value for the number of nodes n.

To keep the state space tractable, we made additional restrictions. Firstly, we restricted the domain of the sorts `HEADER`, `DATA`, and `ACK` to a single value. Secondly, we required that each TRANS process performs a finite number of request transactions. This can be elegantly modeled by adding to each node an `Application` process, which performs a finite number k of requests on the `TDreq` gate. The behavior of the `Trans` process presented in Sect. 5 becomes:

```
hide TDreq: LDreq, TX0:none in
  par
    [TX0,TDreq] -> TransReq [...] (n, id)
    [TX0]       -> TransRes [...] (n, id)
    [TDreq]     -> Application [TDreq]
  end par
end hide
```

where the 'par' operator of E-LOTOS indicates for each process running in parallel the gates on which it shall synchronize.

Finally, we considered three particular scenarios for the applications connected to the TRANS level:

S1. All the applications are passive (behavior `stop`), except one (e.g., node 0) that performs a single request (behavior `TDreq (!id, !dest, !h, !d); stop`).

S2. All the applications perform a single request (behavior `TDreq (!id, !dest, !h, !d); stop`).

S3. All the applications are passive (behavior `stop`), except one (e.g., node 0) that performs only k broadcast requests (behavior `TDreq (!id, !n, !h, !d)` repeated k times).

The experiments on scenarios **S1** and **S2** were performed for a number of requests k fixed to 1, and for a number of nodes n varying between 1 and 3. The experiments on scenario **S3** were performed for n fixed to 2, and for k ranging between 2 and 4. The results of LTSs generation are given in Table 1. For each experiment, the table gives the size (in number of states and transitions) of the LTS and the time (in hours, minutes, and seconds) required for its generation.

Table 1. Results of LTSs generation for IEEE-1394

sc.	n	k	*protocol LTS* *states*	*trans.*	*time* $(h:m's'')$
	1	1	42	42	$0:00'53''$
S1	2	1	1 552	1 836	$0:01'00''$
	3	1	85 780	109 775	$0:15'41''$
	1	1	42	42	$0:00'52''$
S2	2	1	658 468	822 453	$0:55'45''$
	3	1	>6 942 719	>17 226 055	$9:34'31''$
	2	2	4 894	6 716	$0:01'07''$
S3	2	3	26 136	37 975	$0:01'22''$
	2	4	76 660	115 770	$0:03'30''$

All the experiments but one were performed on a Sun Ultra Sparc-1 machine (143 MHz) with 256 Mbytes of memory. The experiment on scenario **S2** with $n = 3$ and $k = 1$ was performed on a Sun Enterprise-4000 machine with 2 Gbytes of memory: the corresponding results are partial, because the generation was interrupted due to memory limitations.

The state explosion problem prevented us from studying more complex scenarios with a greater number of nodes and/or requests. We identified several reasons for this:

- A rough estimation of the state space for $n = 2$ (based on the sizes of state variable domains) gives six million states approximately.
- The presence of an unreliable medium induces a high degree of non-determinism: a signal can disappear, change its size or its destination, or be corrupted.
- This non-determinism is propagated to the LINK layer, which uses a "line listening" protocol, and therefore must take into account all possible incoming signals.
- The splitting of each data packet into four signals causes a "fine granularity" of the protocol behavior.

10 Correctness requirements

The CADP toolbox offers two different verification approaches: bisimulations (using the ALDÉBARAN tool) and temporal logic properties (using the XTL tool). In this case study we chose the second approach, because the desired correctness properties for the IEEE-1394 protocol expressed in natural language (see Sect. 10.2) are easier to translate into temporal logic formulas rather than bisimulations between LTSs and lead to shorter specifications.

As the dynamic semantics of LOTOS and E-LOTOS are action-based, it is natural to express the properties of programs in a temporal logic interpreted over the actions of LTSs. We used here a simplified fragment of the ACTL (Action CTL) temporal logic defined in [25], which is sufficiently powerful to express safety and liveness properties.

First, we briefly present the syntax and semantics of the ACTL fragment we used. Then, we express in ACTL the required properties of the IEEE-1394 protocol.

10.1 The ACTL temporal logic

In order to express predicates over the program actions (the so-called *basic predicates*), a small auxiliary logic of actions is needed. The action formulas α of this logic have the following syntax:

$$\alpha ::= \mathbf{true} \mid \{G\ V_1, ..., V_n\} \mid \neg\alpha \mid \alpha \wedge \alpha'$$

The $\{G\ V_1, ..., V_n\}$ construct denotes an *action pattern*, where G is a gate name and the values V_i ($1 \leq i \leq n$, $n \geq 0$) match the corresponding values exchanged (i.e., sent or received) when the action is performed. For simplicity, and unlike the original ACTL logic, we also allow action patterns (of the form $\{\tau\}$) matching τ-actions.

The usual derived boolean operators are also allowed: we write **false** for $\neg\mathbf{true}$, $\alpha \vee \alpha'$ for $\neg(\neg\alpha \wedge \neg\alpha')$, and $\alpha \Rightarrow \alpha'$ for $\neg\alpha \vee \alpha'$.

The action formulas α are interpreted over the actions $a \in A$ of the model $M = \langle Q, A, T, q_{\text{init}}\rangle$ corresponding to a LOTOS program. The satisfaction of an action formula α by an action $a \in A$, written $a \models_M \alpha$ (or simply $a \models \alpha$ if the model M is understood), is defined inductively by:

$a \models \mathbf{true}$	always;
$a \models \{G\ V_1, ..., V_n\}$	iff $a = G\ V_1, ..., V_n$;
$a \models \neg\alpha$	iff $a \not\models \alpha$;
$a \models \alpha \wedge \alpha'$	iff $a \models \alpha$ and $a \models \alpha'$.

The formulas ϕ of the ACTL fragment we used are defined by the following syntax:

$$\begin{aligned}\phi ::= {} & \mathbf{true} \mid \neg\phi \mid \phi \wedge \phi' \mid \mathbf{init} \\ & \mid \mathbf{EX}_\alpha\phi \mid \mathbf{E}\,[\phi_\alpha \mathbf{U} \phi'] \mid \mathbf{E}\,[\phi_\alpha \mathbf{U}_{\alpha'} \phi'] \\ & \mid \mathbf{AX}_\alpha\phi \mid \mathbf{A}\,[\phi_\alpha \mathbf{U} \phi'] \mid \mathbf{A}\,[\phi_\alpha \mathbf{U}_{\alpha'} \phi']\end{aligned}$$

The **init** formula (which is not part of the original ACTL definition) characterizes the initial state of an LTS. We added it in order to express more naturally certain properties.

The satisfaction of an ACTL formula ϕ by a state $q \in Q$ of an LTS $M = \langle Q, A, T, q_{\text{init}}\rangle$, written $q \models_M \phi$ (or simply $q \models \phi$ if the model M is understood), is defined

inductively by:

$q \models \mathbf{true}$ always;
$q \models \neg\phi$ iff $q \not\models \phi$;
$q \models \phi \wedge \phi'$ iff $q \models \phi$ and $q \models \phi'$;
$q \models \mathbf{init}$ iff $q = q_{\text{init}}$;
$q \models \mathbf{EX}_\alpha \phi$ iff $\exists q \xrightarrow{a} q' \in T$ such that $a \models \alpha$ and $q' \models \phi$;
$q \models \mathbf{E}\,[\phi_\alpha \mathbf{U} \phi']$ iff $\exists q(=q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \in Path(q)$, $\exists k \geq 0$ such that $q_k \models \phi'$ and $\forall i \in [0; k-1], q_i \models \phi$ and $a_i \models \alpha$;
$q \models \mathbf{E}\,[\phi_\alpha \mathbf{U}_{\alpha'} \phi']$ iff $\exists q(=q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \in Path(q)$, $\exists k > 0$ such that $q_k \models \phi'$ and $\forall i \in [0; k-1], q_i \models \phi$ and $\forall j \in [0; k-2], a_j \models \alpha$ and $a_{k-1} \models \alpha'$;
$q \models \mathbf{AX}_\alpha \phi$ iff $\forall q \xrightarrow{a} q' \in T$, $a \models \alpha$ and $q' \models \phi$;
$q \models \mathbf{A}\,[\phi_\alpha \mathbf{U} \phi']$ iff $\forall q(=q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \in Path(q)$, $\exists k \geq 0$ such that $q_k \models \phi'$ and $\forall i \in [0; k-1], q_i \models \phi$ and $a_i \models \alpha$;
$q \models \mathbf{A}\,[\phi_\alpha \mathbf{U}_{\alpha'} \phi']$ iff $\forall q(=q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \in Path(q)$, $\exists k > 0$ such that $q_k \models \phi'$ and $\forall i \in [0; k-1], q_i \models \phi$ and $\forall j \in [0; k-2], a_j \models \alpha$ and $a_{k-1} \models \alpha'$.

A model $M = \langle Q, A, T, q_{\text{init}} \rangle$ satisfies a formula ϕ, noted $M \models \phi$ (or simply ϕ if the model M is understood), if and only if $q \models \phi$ for all $q \in Q$.

Besides the usual derived boolean operators **false**, $\vee$, and $\Rightarrow$, we also define the following useful derived modalities and temporal operators:

$\langle \alpha \rangle \phi = \mathbf{EX}_\alpha \phi$
$[\alpha] \phi = \neg \langle \alpha \rangle \neg \phi$
$\mathbf{EF}_\alpha \phi = \mathbf{E}\,[\mathbf{true}_\alpha \mathbf{U} \phi]$
$\mathbf{AF}_\alpha \phi = \mathbf{A}\,[\mathbf{true}_\alpha \mathbf{U} \phi]$
$\mathbf{EG}_\alpha \phi = \neg \mathbf{AF}_\alpha \neg \phi$
$\mathbf{AG}_\alpha \phi = \neg \mathbf{EF}_\alpha \neg \phi$

The $\langle \alpha \rangle \phi$ and $[\alpha] \phi$ operators are the well-known Hennessy–Milner modalities [15]. A state q satisfies $\langle \alpha \rangle \phi$ (*resp.* $[\alpha] \phi$) iff some (*resp.* all) of its direct successors reached after an action satisfying α satisfies (*resp.* satisfy) ϕ. A state q satisfies $\mathbf{EF}_\alpha \phi$ (*resp.* $\mathbf{AF}_\alpha \phi$) iff some path (*resp.* all paths) issued from q leads (*resp.* lead) via actions satisfying α to a state satisfying ϕ. A state q satisfies $\mathbf{EG}_\alpha \phi$ (*resp.* $\mathbf{AG}_\alpha \phi$) iff for some path (*resp.* all paths) issued from q, every prefix consisting of actions that satisfy α leads to a state satisfying ϕ.

10.2 The correctness properties of IEEE-1394

The expected functioning of the IEEE-1394 LINK layer protocol was informally characterized by Luttik [22] in the form of five correctness requirements stated in natural language. Together, these requirements specify the essential safety and liveness properties concerning the transmission of data packets and signals during the fairness intervals; in particular, they precise the responses of the protocol to the immediate and fair arbitration requests issued by the LINKs.

In this section, we formally express these properties as ACTL temporal logic formulas. For conciseness, when writing the ACTL formulas, we will use suggestive names for the action patterns rather than their precise syntax defined in Sect. 10.1. For example, instead of writing "{ `PAreq id immediate` }" for the action pattern denoting the emission of a request by the node `id` on gate `PAreq` with parameter `immediate`, we will simply write "`PAREQ_id_immediate`." Also, we introduce the following shorthand notations:

- $\mathbf{inev}(\alpha_1, \alpha_2) = \mathbf{A}\,[\mathbf{true}_{\alpha_1} \mathbf{U}_{\alpha_2} \mathbf{true}]$, meaning that the program eventually performs an action satisfying α_2, possibly preceded only by actions satisfying α_1. Informally, $\mathbf{inev}(\alpha_1, \alpha_2)$ ensures the reachability of α_2 independently of the (fair or unfair) scheduling policy of actions.
- $\mathbf{fair}(\alpha_1, \alpha_2) = \mathbf{AG}_{\neg\alpha_2 \wedge \alpha_1} \mathbf{EF}_{\alpha_1} \langle \alpha_2 \rangle \mathbf{true}$, meaning that every sequence of actions which do not satisfy α_2, but satisfy α_1, leads to a state from which it is possible to reach an action satisfying α_2. Informally, $\mathbf{fair}(\alpha_1, \alpha_2)$ means that, assuming a fair scheduling of actions, the program will eventually reach an action satisfying α_2.

These operators are the action-based translations of the corresponding state-based operators defined in the LTAC temporal logic [27].

The five correctness properties can be formulated in ACTL as follows.

Property 1.

The protocol is deadlock free.

We must express this property in the context of the finite behaviors we considered for TRANS: when all TRANS entities have reached their quota (in terms of transaction requests), the protocol will eventually reach a "terminating state", since no more request transaction can be done. The problem is to make a distinction between these terminating states (artifacts of our machine limitations) and real deadlocks. A careful examination of the LINK and TRANS behaviors allowed us to identify these "correct" terminating states: they can occur only after an "arbitration reset gap" signal (action pattern `arbresgap`) followed by 0 or more confirmations that are sent back to the TRANS layer (action pattern `LDcon_any`). Thus, a deadlock occurring in a state different from the aforementioned terminating states is a real one. The formula below expresses that no such deadlock can be reached from the initial state of the program.

$$\mathbf{init} \Longrightarrow \neg \mathbf{EF}_{\mathbf{true}}\ \langle \neg(\texttt{arbresgap} \vee \texttt{LDcon\_any}) \rangle\ \mathbf{EF}_{\texttt{LDcon\_any}}\ [\mathbf{true}]\ \mathbf{false}$$

Property 2.

Between two subsequent "subaction gap" signals (action pattern `PDind_any_sgap`*) at most two asynchronous packets have traveled over the* BUS*.*

We model the fact that a packet has traveled over the BUS by means of the `LDcon_any` action pattern, which stands for the reception of a confirmation on the `LDcon` gate by some TRANS requester.

$$\mathbf{AG}_{\mathbf{true}}\ [\texttt{PDind\_any\_sgap}]\ \mathbf{AG}_{\neg(\texttt{PDind\_any\_sgap} \vee \texttt{LDcon\_any})}\ [\texttt{LDcon\_any}]\ \mathbf{AG}_{\neg(\texttt{PDind\_any\_sgap} \vee \texttt{LDcon\_any})}\ [\texttt{LDcon\_any}]\ \mathbf{AG}_{\neg\texttt{PDind\_any\_sgap}}\ [\texttt{LDcon\_any}]\ \mathbf{false}$$

Property 3.

If a node $0 \leq$ `id` $\leq n-1$ *emitted a request on the* `LDreq` *gate (action pattern* `LDreq_id`*) and node* `id` *communicates a fair request on the* `PAreq` *gate (action pattern* `PAreq_id_fair`*) each time it receives a "subaction gap" signal on the* `PDind` *gate (action pattern* `PDind_id_sgap`*) — and before an "arbitration reset gap" signal (action pattern* `arbresgap`*) occurs — it also eventually receives a confirmation on the* `LDcon` *gate (action pattern* `LDcon_id`*).*

Recall (from Sect. 6) that the LTS model contains livelocks, which will not appear in a practical implementation due to time constraints. In order to ignore such unfair executions, we must use **fair** instead of **inev** to express the reachability of the `LDcon` action. Notice also the presence of the predicate ¬`arbresgap` as part of the action formulas guarding the last two **AG** operators below, in order to ensure that the formula refers to the same fairness interval (i.e., no `arbresgap` action occurred meanwhile).

$$\mathbf{AG}_{\mathbf{true}}\ [\texttt{LDreq\_id}]\ \mathbf{AG}_{\neg(\texttt{PDind\_id\_sgap} \vee \texttt{arbresgap} \vee \texttt{LDcon\_id})}\ [\texttt{PDind\_id\_sgap}]\ \mathbf{AG}_{\neg(\texttt{PAreq\_id\_fair} \vee \texttt{arbresgap})}\ [\texttt{PAreq\_id\_fair}]\ \mathbf{fair}(\mathbf{true}, \texttt{LDcon\_id})$$

Property 4.

Every request emitted by node $0 \leq id \leq n-1$ *on gate* `PAreq` *with parameter* `immediate` *(action pattern* `PAreq_id_immediate`*) is followed by a matching confirmation on gate* `PAcon` *with parameter* `won` *(action pattern* `PAcon_id_won`*).*

As for the previous property, we must use the **fair** operator to express the reachability of the `PAcon` action in the presence of livelocks.

$$\mathbf{AG}_{\mathbf{true}}\ [\texttt{PAreq\_id\_immediate}]\ \mathbf{fair}(\neg\texttt{PAreq\_id\_immediate}, \texttt{PAcon\_id\_won})$$

Property 5.

Between two subsequent "arbitration reset gap" signals (action pattern `arbresgap`*) no node* $0 \leq id \leq n-1$ *receives a confirmation on gate* `PAcon` *with parameter* `won` *(action pattern* `PAcon_id_won`*) upon a request on gate* `PAreq` *with parameter* `fair` *(action pattern* `PAreq_id_fair`*) more than once.*

Notice again the predicate ¬`arbresgap` used in the last two **AG** operators of the formula below, in order to ensure that no "arbitration reset gap" signal occurred after the one matched by the first box modality.

$$\mathbf{AG}_{\mathbf{true}}\ [\texttt{arbresgap}]\ \mathbf{AG}_{\neg\texttt{arbresgap}}\ [\texttt{PAreq\_id\_fair}]\ [\texttt{PAcon\_id\_won}]\ \mathbf{AG}_{\neg\texttt{arbresgap}}\ [\texttt{PAreq\_id\_fair}]\ [\texttt{PAcon\_id\_won}]\ \mathbf{false}$$

The properties 1, 3, and 4 are liveness properties; properties 2 and 5 are safety properties.

11 Verification

The five temporal logic formulas given in Sect. 10.2 were evaluated on the LTSs corresponding to the scenarios **S1**, **S2**, and **S3** using the XTL [23] prototype model checker.

It is worth noticing that, since the XTL language allows the definition of macro-notations (for action predicates as well as for temporal operators), the ACTL formulas given in Sect. 10.2 are almost identical to those written in the XTL source code.

The verifications were performed for all the LTSs that were successfully produced by model generation (see Sect. 9.2), previously minimized modulo strong bisimulation [26] using ALDÉBARAN. Table 2 summarizes the verification results obtained. For each experiment, the table gives the size (in number of states and transitions) of the minimized LTS and the times (in minutes and seconds) required for minimization and verification of the five properties.

Properties 2–5 are true in all scenarios.

Property 1 is false in all scenarios, meaning that an unexpected deadlock occurs in the protocol. We obtained

Table 2. Results of verification for IEEE-1394

			minimized LTS		*time* ($m's''$)	
sc.	n	k	*states*	*trans.*	*minim.*	*verif.*
	1	1	21	20	0.4″	4.6″
S1	2	1	155	214	0.7″	5.0″
	3	1	710	1 206	29.6″	6.5″
	1	1	21	20	0.4″	4.7″
S2	2	1	2 893	5 610	2′49.5″	20.7″
	2	2	225	351	1.4″	5.1″
S3	2	3	452	723	6.2″	5.7″
	2	4	663	1 071	19.5″	6.3″

a counterexample by considering the following formula, disjoint from property 1:

$$\textbf{init} \Longrightarrow \textbf{EF}_{\textbf{true}}\ \langle \neg(\texttt{arbresgap} \vee \texttt{LDcon\_any})\rangle\ [\textbf{true}]\ \textbf{false}$$

meaning that there is a sequence starting from the initial state and leading to a "real deadlock" (according to the definition given in Sect. 10.2):

$$q_{init} \xrightarrow{a_0} q_1 \xrightarrow{a_1} \cdots \xrightarrow{a_{l-1}} q_l \xrightarrow{a_l} q_{deadlock}$$

where $a_l \models \neg(\texttt{arbresgap} \vee \texttt{LDcon\_any})$.

The minimal sequence of this kind we were able to find using the `exhibitor` tool of CADP was of length $l = 50$.

By examining this sequence, we were able to identify the cause of the deadlock, which is the absence of the `LDres` action in the LINK state `Link4BRec` after receiving a broadcast packet (see Sect. 6). Although both the state machine of LINK and the μCRL description [22] do not specify the `LDres` action, the IEEE-1394 Standard [17] says that: "*The transaction layer shall communicate this response* [link data response, `LDres`] *after receiving a link data indication*". However, when the LINK layer receives a link data response in reaction to a link data indication of a broadcast, "*the link layer shall do nothing*" (p. 148). In this sense, the standard is ambiguous, because it does not specify clearly the semantics of the interconnection between the state machines: in the state machine diagram of LINK, no link data response can be accepted after a link data indication of broadcast arrival.

A corrected version of the `Link4BRec` body is given below. This behavior corrects the state machine given in [17, p. 174] and the μCRL description and is compatible with the explanatory text of the LINK services given at pp. 147–148 of the IEEE-1394 Standard.

```
if (getdcrc (d) == check) then
  LDind (!id, !broadrec(gethead(h),getdata(d)));
    LDres (!id, ?any of ACK, !no_op)  (* added *)
end if;
Link0 [...] (n, id, buf)
```

Using this corrected version of the LINK, we generated again the models for all scenarios given in Sect. 9.2. Since the sizes of the new LTSs are very close to those of the previous ones, we do not give them here.

On the new LTSs, all the five correctness properties have been checked to be true. This constitutes a strong indication that the resulting description is correct, since due to the fairness interval strategy of the BUS, the three particular scenarios considered in Sect. 9.2 cover all the relevant cases that may occur in the functioning of the protocol.

12 Conclusion

In this paper, we have presented the formal description in E-LOTOS of the LINK layer protocol of the "FireWire" high performance serial bus defined in the IEEE-1394 Standard [17] and its verification by model checking using the CADP (CÆSAR/ALDÉBARAN) protocol engineering toolbox. The E-LOTOS descriptions of the LINK and BUS layers were derived from the corresponding ones written in μCRL by Luttik [22]. The description of the TRANS layer is based on the state machines and text explanations given in [17].

The E-LOTOS description we obtained has 7 pages instead of 13 pages of μCRL in [22]. Especially for the data types, the gain in conciseness is significant: 4 pages of E-LOTOS instead of 8 pages of μCRL. Also, it is worth noticing that E-LOTOS allows a clearer and more readable description of the behavior expressions than LOTOS and μCRL. In this sense, the fragment of E-LOTOS language we used seems to be adequate to protocol description, and eliminates some deficiencies of LOTOS.

To perform model-checking verifications, we generated the Labeled Transition Systems (LTSs) of the protocol by translating the E-LOTOS descriptions in LOTOS using the TRAIAN compiler, and then using the CÆSAR and CÆSAR.ADT compilers. To obtain LTSs of tractable size, we limited the domains of the protocol parameters (at most three nodes connected to the BUS) and we considered only three finite scenarios for the TRANS layer.

We performed verification by means of temporal logics using the XTL prototype model checker. We expressed in the ACTL temporal logic a set of five correctness properties given in natural language by Luttik [22] and we verified them on the IEEE-1394 LTS.

This verification approach allowed us to exhibit a missing transition in the state machine of LINK given in the IEEE-1394 Standard, which would induce a deadlock in the implementations that follow strictly this state machine (as it is the case with the μCRL description). We corrected our E-LOTOS description by adding this missing transition; on the new version of the protocol, all the correctness properties were successfully verified. We consider that, to avoid such ambiguities, the IEEE-1394 Standard should be improved by giving a precise definition of the state machine transitions and a formal semantics of state machine parallel composition.

The effort required to perform this case study was one man-month. The most important part of this effort was concentrated in finding the "good" abstractions of the TRANS layer behavior, in order to avoid a state explosion.[11] Although we were not able to generate the model for an infinite behavior of the TRANS, the search for appropriate abstractions increased our confidence in the good functioning of the protocol obtained after correction.

This experience reinforced our opinion that formal methods are very useful for the design and development of complex, critical applications. Using a formal approach,

[11] It is worth noticing that the tools are still under improvement; we hope to obtain better results in the future.

one can benefit not only from a disciplined methodology avoiding the ambiguities that may occur in semi-formal descriptions, but also from a basis for exhaustive verification.

Acknowledgements. Thanks are due to Hubert Garavel, Charles Pecheur, and the anonymous referees for their suggestions and careful reading of this paper, to Bas Luttik and Laurent Mounier for useful discussions, and to Bruno Vivien for implementing the TRAIAN translator. We are also grateful to Catherine Cassagne for granting us access to the Sun Enterprise-4000 computer of the ENSIMAG engineering school.

References

1. ANSI/IEEE. Standard for Control and Status Register (CSR) Architecture for Microcomputer Buses. ANSI/IEEE Standard ISO/IEC 13213:1994, ANSI/IEEE Standard 1212, Institute of Electrical and Electronic Engineers 1994
2. Bolognesi, T., Brinksma, E.: Introduction to the ISO specification language LOTOS. Computer Networks and ISDN Systems, 14(1): 25–29, 1988
3. Bozga, M., Fernandez, J.-C., Kerbrat, A., Mounier, L.: Protocol verification with the ALDEBARAN toolset. Software Tools for Technology Transfer 1(1–2): 166–183, 1997
4. Chehaibar, G., Garavel, H., Mounier, L., Tawbi, N., Zulian, F.: Specification and Verification of the PowerScale Bus Arbitration Protocol: An Industrial Experiment with LOTOS. In: Gotzhein, R., Bredereke, J. (eds.): Proc. of FORTE/-PSTV '96 Kaiserslautern, Germany, IFIP, Chapman & Hall, 1996, pp. 435–450
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems 8(2): 244–263, 1986
6. de Meer, J., Roth, R., Vuong, S.: Introduction to algebraic specifications based on the language ACT ONE. Computer Networks and ISDN Systems 23(5): 363–392, 1992
7. Fernandez, J.C., Garavel, H., Kerbrat, A., Mateescu, R., Mounier, L., Sighireanu, M.: CADP (CÆSAR/ALDEBARAN Development Package): a protocol validation and verification toolbox. In: Alur, R., Henzinger, T.A. (eds.): Proc. of CAV '96, New Brunswick, New Jersey, USA. LNCS 1102. Berlin, Heidelberg, New York: Springer-Verlag, 1996, pp. 437–440
8. Garavel, H.: Compilation of LOTOS abstract data types. In: Vuong, S.T. (ed.): Proc. of FORTE '89, Vancouver B.C., Canada, Amsterdam, North-Holland, 1989, pp. 147–162
9. Garavel, H.: On the introduction of gate typing in E-LOTOS. In: Dembinski, P., Sredniawa, M. (eds.): Proc. of PSTV '95, Warsaw, Poland. IFIP, Chapman & Hall, 1995
10. Garavel, H.: An overview of the Eucalyptus toolbox. In: Brezočnik, Z., Kapus, T. (eds.): Proc. of the COST 247 International Workshop on Applied Formal Methods in System Design, Maribor, Slovenia, June 1996. University of Maribor, Slovenia, 1996, pp. 76–88
11. Garavel, H., Jorgensen, M., Mateescu, R., Pecheur, C., Sighireanu, M., Vivien, B.: CADP '97 – status, applications and perspectives. In: Lovrek, I. (ed.): Proc. of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design, Zagreb, Croatia, June 1997
12. Garavel, H., Sifakis, J.: Compilation and verification of LOTOS specifications. In: Logrippo, L., Probert, R.L., Ural, H. (eds.): Proc. of PSTV '90, Ottawa, Canada, IFIP, North-Holland 1990, pp. 379–394
13. Groote, J.F., Ponse, A.: The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, December 1990
14. Guttag, J.: Abstract data types and the development of data structures. Communications of the ACM 20(6): 396–404, 1977
15. Hennessy, M., Milner, R.: Algebraic laws for nondeterminism and concurrency. Journal of the ACM, 32: 137–161, 1985
16. Hoare, C.A.R.: Communicating Sequential Processes. Englewood Cliffs, NJ: Prentice-Hall 1985
17. IEEE: Standard for a High Performance Serial Bus. IEEE Standard 1394-1995, Institute of Electrical and Electronic Engineers 1995
18. ISO/IEC: LOTOS – A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour. International Standard 8807, International Organization for Standardization – Information Processing Systems – Open Systems Interconnection, Geneva, September 1988
19. ISO/IEC: Distributed Transaction Processing – Part 3: Protocol Specification. International Standard 10026-3, ISO – Information Processing Systems – OSI, Geneva, 1992
20. ISO/IEC: ODP Trading Function. Draft Rec X9tr 13235, ISO/IEC, June 1995
21. Lai, R., Lo, A.: An analysis of the ISO FTAM Basic File Protocol specified in LOTOS. Australian Computer Journal 27(1): 1–7, 1995
22. Luttik, B.: Description and formal specification of the Link layer of P1394. In: Lovrek, I. (ed.): Proc. of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design, Zagreb, Croatia, June 1997
23. Mateescu, R., Garavel, H.: XTL: A meta-language and tool for temporal logic model-checking. In: Margaria, T. (ed.): Proc. of the International Workshop on Software Tools for Technology Transfer STTT '98, Aalborg, Denmark, July 1998
24. Milner, R.: Communication and Concurrency. Englewood Cliffs, NJ: Prentice-Hall 1989
25. de Nicola, R., Vaandrager, W.F.: Action versus state based logics for transition systems. In: Proc. Ecole de Printemps on Semantics of Concurrency. LNCS 469. Berlin, Heidelberg, New York: Springer-Verlag, 1990, pp. 407–419
26. Park, D.: Concurrency and automata on infinite sequences. In: Deussen, P. (ed.): Theoretical Computer Science. LNCS 104. Berlin, Heidelberg, New York: Springer-Verlag, 1981, pp. 167–183
27. Queille, J.P., Sifakis, J.: Fairness and related properties in transition systems – a temporal logic to deal with fairness. Acta Informatica 19: 195–220, 1983
28. Quemada, J. (ed.): Committee Draft on Enhancements to LOTOS. Draft international standard, ISO/IEC JTC1/SC21/WG7 Project 1.21.20.2.3, February 1998
29. Sighireanu, M.: LOTOS NT Manual V1.0. INRIA, Grenoble, May 1998. Available at `http://www.inrialpes.fr/vasy/people/Mihaela.Sighireanu/LOTOSNT/manual.ps.gz`
30. Sighireanu, M., Garavel, H.: E-LOTOS User Language. Rapport SPECTRE 96-06, VERIMAG, Grenoble, October 1996. In: ISO/IEC JTC1/SC21 Third Working Draft on Enhancements to LOTOS (1.21.20.2.3). Output document of the edition meeting, Kansas City, MO, May 1996
31. Turner, K.J. (ed.): Using formal description techniques – an introduction to ESTELLE, LOTOS, and SDL. New York: John Wiley 1993
32. Vivien, B.: Etude et réalisation d'un compilateur E-LOTOS à l'aide du générateur de compilateurs SYNTAX/FNC-2. Mémoire d'ingénieur, CNAM, Grenoble, December 1997. Available at `http://www.inrialpes.fr/vasy/Publications/Vivien-97.html`

Appendix A: Description of the data structures

This annex presents the E-LOTOS module `DATA` defining the data types (informally described in Sect. 4) used in the protocol. The types `Nat` and `Bool` are predefined E-LOTOS types for natural numbers and booleans.

```
module DATA is

   type ACK is
      a1, a2
   end type

   type CHECK is
      bottom, check
   end type

   type DATA is
      d1, d2
   end type

   type HEADER is
      h1, h2
   end type

   type PHY_ACONF is
      won, lost
   end type

   type PHY_AREQ is
      fair, immediate
   end type

   function crc (d: DATA): CHECK is
      return check
   end function

   function crc (h: HEADER): CHECK is
      return check
   end function

   function crc (a: ACK): CHECK is
      return check
   end function

   type BOC is
      release, hold, no_op
   end type

   type ACKSIG is
      acksig (a: ACK, c: CHECK)
   end type

   type DESTSIG is
      destsig (dest: Nat)
   end type

   type HEADERSIG is
      headersig (h: HEADER, c: CHECK)
   end type

   type DATASIG is
      datasig (d: DATA, c: CHECK)
   end type

   type SIGNALS is
      sig (dest: DESTSIG),
      sig (a: ACKSIG),
      sig (h: HEADERSIG),
      sig (d: DATASIG),
      dhead,
      Start,
      EndS,
      Prefix, subactgap, Dummy
   end type

   function is_dest (x: SIGNALS): Bool is
      case x is
        sig (any of DESTSIG) -> return true
      | any -> return false
      end case
   end function

   function is_header (x: SIGNALS): Bool is
      case x is
        sig (any of HEADERSIG) -> return true
      | any -> return false
      end case
   end function

   function is_data (x: SIGNALS): Bool is
      case x is
        sig (any of DATASIG) -> return true
      | any -> return false
      end case
   end function

   function is_ack (x: SIGNALS): Bool is
      case x is
        sig (any of ACKSIG) -> return true
      | any -> return false
      end case
   end function

   function is_physig (x: SIGNALS): Bool is
      case x is
        Start
      | EndS
      | Prefix
      | subactgap -> return true
      | any -> return false
      end case
   end function

   function valid_ack (x: SIGNALS): Bool is
      case x is
        sig (acksig (any of ACK, check))
            -> return true
      | any -> return false
      end case
   end function

   function valid_hpart (x: SIGNALS): Bool is
      case x is
        sig (headsig (any of HEADER, check))
            -> return true
      | any -> return false
      end case
   end function

   function getdest (x: SIGNALS): Nat is
      case x is
        sig (destsig (xn: Nat)) -> return xn
      | any -> return 0
      end case
   end function

   function gethead (x: SIGNALS): HEADER is
      case x is
        sig (headersig (xh: HEADER, any))
            -> return xh
      | any -> return h1
```

```
   end case
end function

function getdcrc (x: SIGNALS): CHECK is
   case x is
     sig (datasig (any, xc: CHECK))
         -> return xc
   | any -> return check
   end case
end function

function getdata (x: SIGNALS): DATA is
   case x is
     sig (datasig (xd: DATA, any))
         -> return xd
   | any -> return d1
   end case
end function

function getack (x: SIGNALS): ACK is
   case x is
     sig (acksig (xa: ACK, any))
         -> return xa
   | any -> return a1
   end case
end function

function corrupt (x: SIGNALS): SIGNALS is
   case x is
     sig (headersig(xh: HEADER, any))
     -> return sig (headersig(xh,bottom))
   | sig (datasig (xd: DATA, any))
     -> return sig (datasig (xd,bottom))
   | sig (acksig (xa: ACK, any))
     -> return sig (acksig (xa,bottom))
   | any -> return x
   end case
end function

type SIG_TUPLE is
   quadruple (dh: SIGNALS, dest: SIGNALS,
              h: SIGNALS, d: SIGNALS),
   void
end type

function is_void (x: SIG_TUPLE): Bool is
   return (x == void)
end function

type LIN_DCONF is
   ackrec (a: ACK), ackmiss, broadsent
end type

type LIN_DIND is
   good (h: HEADER, d: DATA),
   broadrec (h: HEADER, d: DATA),
   dcrc_err (h: HEADER)
end type

function is_broadcast (x: LIN_DIND): Bool is
   case x is
     broadrec (any, any) -> return true
   | any -> return false
   end case
end function

type LDreqType is record
   id:Nat,dest:Nat,h:HEADER,d:DATA
end type

type LDconType is record
   id: Nat, dc: LIN_DCONF
end type

type LDindType is record
   id: Nat, di: LIN_DIND
end type

type LDresType is record
   id: Nat, a: ACK, b: BOC
end type

type PDreqType is record
   id: Nat, s: SIGNALS
end type

type PDindType is record
   id: Nat, s: SIGNALS
end type

type PAreqType is record
   id: Nat, ar: PHY_AREQ
end type

type PAconType is record
   id: Nat, ac: PHY_ACONF
end type

type BoolTABLE is
   empty,
   btable (x: Nat, b: Bool, t: BoolTABLE)
end type

function init (x: Nat): Booltable
is
   case x is
     0   -> return empty
   | any -> return btable (x-1, false,
                           init (x-1))
   end case
end function

function invert (x: Nat, t: BoolTABLE)
               : BoolTABLE
is
   case t is
     empty -> return empty
   | btable (xnat: Nat, xbool: Bool,
             xbt: BoolTABLE) ->
        if (xnat == x) then
           return btable (xnat, not (xbool),
                          xbt)
        else
           return btable (xnat, xbool,
                          invert (x, xbt))
        end if
   end case
end function

function get (x: Nat, t: BoolTABLE)
            : Bool is
   case t is
     empty -> return true
   | btable (xnat: Nat, xbool: Bool,
             xbt: BoolTABLE) ->
        if (xnat == x) then
           return xbool
        else
           return get (x, xbt)
        end if
   end case
end function
```

```
   function zero (t: BoolTABLE): Bool is
      case t is
        empty -> return true
      | btable (any, true, any)
              -> return false
      | btable (any, false,
                xbt: BoolTABLE)
              -> return zero (xbt)
      end case
   end function

   function one (t: BoolTABLE): Bool is
      case t is
        empty -> return false
      | btable (any,true,xbt:BoolTABLE)
              -> return zero (xbt)
      | btable (any,false,xbt: BoolTABLE)
              -> return one (xbt)
      end case
   end function

   function more (t: BoolTABLE): Bool is
      return not (zero (t)) and not (one (t))
   end function

end module
```

Appendix B: Description of the LINK layer

This annex presents the E-LOTOS module `LINK` defining the processes used to model the LINK layer behavior (informally described in Sect. 6). To improve readability, we use the shorthand notation `<<Link gates>>` for the following list of typed gates: `LDreq: LDreqType`, `LDcon: LDconType`, `LDind: LDindType`, `LDres: LDresType`, `PDreq: PDreqType`, `PDind: PDindType`, `PAreq: PAreqType`, `PAcon: PAconType`, `PCind: Nat`.

```
module LINK import DATA is

  process Link [<<Link gates>>]
              (n, id: Nat)
  is
    Link0 [...] (n, id, void)
  end process

  process Link0 [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE) is
    if is_void (buf) then
      var dest: Nat, h: HEADER, d: DATA in
        LDreq (!id, ?dest, ?h, ?d);
        Link0 [...] (n, id, quadruple (dhead,
              sig (destsig (dest)),
              sig (headersig (h, crc (h))),
              sig (datasig (d, crc (d)))))
      end var
    else
      PAreq (!id, !fair);
        (PAcon (!id, !won);
           Link2req [...] (n, id, buf)
         []
         PAcon (!id, !lost);
           Link0 [...] (n, id, buf))
    end if
   []
    var p: SIGNALS in
      PDind (!id, ?p);
        if (p == Start) then
          Link4 [...] (n, id, buf)
        else
          Link0 [...] (n, id, buf)
        end if
    end var
  end process

  process Link2req [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE) is
    PCind !id; PDreq (!id, !Start);
    PCind !id; PDreq (!id, !buf.dh);
    PCind !id; PDreq (!id, !buf.dest);
    PCind !id; PDreq (!id, !buf.header);
    PCind !id; PDreq (!id, !buf.data);
    PCind !id; PDreq (!id, !EndS);
    if (getdest (buf.dest) == n) then
      LDcon (!id, !broadsent);
        Link0 [...] (n, id, void)
    else
      Link3 [...] (n, id, void)
    end if
  end process

  process Link3 [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE) is
    var p: SIGNALS in
      PDind (!id, ?p);
        if (p == Prefix) then
          Link3 [...] (n, id, buf)
        elsif (p == Start) then
          Link3RA [...] (n, id, buf)
        elsif (p == subactgap) then
          LDcon (!id, !ackmiss);
            Link0 [...] (n, id, buf)
        else
          LDcon (!id, !ackmiss);
            LinkWSA [...] (n, id, buf, n)
        end if
    end var
  end process

  process Link3RA [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE) is
    var a: SIGNALS in
      PDind (!id, ?a);
        if (a == subactgap) then
          LDcon (!id, !ackmiss);
            Link0 [...] (n, id, buf)
        elsif is_physig (a) then
          LDcon (!id, !ackmiss);
            LinkWSA [...] (n, id, buf, n)
        else
          Link3RE [...] (n, id, buf, a)
        end if
    end var
  end process

  process Link3RE [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE,
           a: SIGNALS) is
    var e: SIGNALS in
      PDind (!id, ?e);
        if valid_ack (a) and
           ((e == EndS) or (e == Prefix)) then
          LDcon (!id, !ackrec (getack (a)))
        else
          LDcon (!id, !ackmiss)
        end if;
        if (e == subactgap) then
          Link0 [...] (n, id, buf)
```

```
      else
        LinkWSA [...] (n, id, buf, n)
      end if
  end var
end process

process Link4 [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE) is
  var dh: SIGNALS in
    PDind (!id, ?dh);
      if (dh == subactgap) then
        Link0 [...] (n, id, buf)
      elsif is_physig (dh) then
        LinkWSA [...] (n, id, buf, n)
      else
        Link4DH [...] (n, id, buf)
      end if
  end var
end process

process Link4DH [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE) is
  var dest: SIGNALS in
    PDind (!id, ?dest);
      if is_dest (dest) and
         (getdest (dest) == id) then
        PAreq (!id, !immediate)
      end if;
      if is_dest (dest) and
         ((getdest (dest) == id) or
         (getdest (dest) == n)) then
        Link4RH [...] (n, id, buf,
                       getdest (dest))
      elsif (dest == subactgap) then
        Link0 [...] (n, id, buf)
      else
        LinkWSA [...] (n, id, buf, n)
      end if
  end var
end process

process Link4RH [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         dest: Nat) is
  var h: SIGNALS in
    PDind (!id, ?h);
      if valid_hpart (h) then
        Link4RD [...] (n,id,buf,dest,h)
      else
        LinkWSA [...] (n,id,buf,dest)
      end if
  end var
end process

process Link4RD [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         dest: Nat, h: SIGNALS) is
  var d: SIGNALS in
    PDind (!id, ?d);
      if is_data (d) then
        Link4RE [...] (n,id,buf,dest,h,d)
      else
        LinkWSA [...] (n,id,buf,dest)
      end if
  end var
end process

process Link4RE [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         dest: Nat, h, d: SIGNALS) is
  var e: SIGNALS in
    PDind (!id, ?e);
      if ((e == EndS) or (e == Prefix)) then
        if (dest == id) then
          Link4DRec [...] (n,id,buf,h,d)
        else
          Link4BRec [...] (n,id,buf,h,d)
        end if
      else
        LinkWSA [...] (n,id,buf,dest)
      end if
  end var
end process

process Link4DRec [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         h, d: SIGNALS) is
  if (getdcrc (d) == check) then
    LDind (!id, !good (gethead (h),
                       getdata (d)))
  else
    LDind (!id, !dcrc_err (gethead (h)))
  end if;
  PAcon (!id, !won); Link5 [...] (n,id,buf)
end process

process Link4BRec [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         h, d: SIGNALS) is
  if (getdcrc (d) == check) then
    LDind (!id, !broadrec (gethead (h),
                           getdata (d)))
  end if;
  Link0 [...] (n, id, buf)
end process

process Link5 [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE) is
  var a: ACK, b: BOC in
    LDres (!id, ?a, ?b);
      Link6 [...] (n, id, buf,
            sig (acksig (a, crc (a))), b)
  end var
  []
  PCind !id; PDreq (!id, !Prefix);
    Link5 [...] (n, id, buf)
end process

process Link6 [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE,
         p: SIGNALS, b: BOC) is
  PCind !id; PDreq (!id, !Start);
  PCind !id; PDreq (!id, !p);
  PCind !id;
  if (b == release) then
    PDreq (!id, !EndS);
      Link0 [...] (n, id, buf)
  else
    PDreq (!id, !Prefix);
      Link7 [...] (n, id, buf)
  end if
end process

process Link7 [<<Link gates>>]
        (n, id: Nat, buf: SIG_TUPLE) is
  PCind !id; PDreq (!id, !Prefix);
    Link7 [...] (n, id, buf)
  []
  var dest: Nat, h: HEADER, d: DATA in
    LDreq (!id, ?dest, ?h, ?d);
    Link2resp [...] (n,id,buf,
              quadruple(dhead,
```

```
              sig (destsig (dest)),
              sig (headersig (h, crc (h))),
              sig (datasig (d, crc (d)))))
    end var
  end process

  process Link2resp [<<Link gates>>]
          (n, id: Nat, buf, p: SIG_TUPLE) is
    PCind !id; PDreq (!id, !Start);
    PCind !id; PDreq (!id, !p.dh);
    PCind !id; PDreq (!id, !p.dest);
    PCind !id; PDreq (!id, !p.header);
    PCind !id; PDreq (!id, !p.data);
    PCind !id; PDreq (!id, !EndS);
    if (getdest (p.dest) == n) then
      LDcon (!id, !broadsent);
        Link0 [...] (n, id, buf)
    else
      Link3 [...] (n, id, buf)
    end if
  end process

  process LinkWSA [<<Link gates>>]
          (n, id: Nat, buf: SIG_TUPLE,
           dest: Nat) is
    loop
      var p: SIGNALS in
        PDind (!id, ?p);
          if (p == subactgap) then break end if
      end var
      []
      if (dest == id) then
        PAcon (!id, !won);
        PCind !id; PDreq (!id, !EndS); break
      end if
    end loop;
    Link0 [...] (n, id, buf)
  end process

end module
```

Appendix C: Description of the BUS layer

This annex presents the E-LOTOS module `BUS` defining the processes used to model the BUS layer behavior (informally described in Sect. 7). To improve readability, we use the `<<Bus gates>>` shorthand notation for the following list of typed gates: `PDind: PDindType`, `PDreq: PDreqType`, `PAcon: PAconType`, `PAreq: PAreqType`, `PCind: Nat`, `arbresgap: none`, `losesignal: none`.

```
module BUS import DATA is

   process Bus [<<Bus gates>>] (n: Nat) is
      BusIdle [...] (n, init (n))
   end process

   process BusIdle [<<Bus gates>>]
           (n: Nat, t: BoolTABLE) is
      var id: Nat, astat: PHY_AREQ in
         PAreq (?id, ?astat) [id < n];
         if (get (id, t) == false) then
            PAcon (!id, !won);
            BusBusy [...] (n, invert (id, t),
                    init (n), init (n), id)
         else
            PAcon (!id, !lost);
            BusIdle [...] (n, t)
         end if
      end var
      []
      if not (zero (t)) then
         arbresgap; BusIdle [...] (n, init (n))
      end if
   end process

   process BusBusy [<<Bus gates>>]
           (n: Nat, t, next,
            destfault: BoolTABLE, busy: Nat)
   is
     var j: Nat in
       PAreq (?j, !fair) [j < n];
         PAcon (!j, !lost);
         BusBusy [...] (n, t, next,
                        destfault, busy)
     end var
     []
     var j: Nat in
       PAreq (?j, !immediate)
             [not(get(j,next)) and (j < n)];
       BusBusy [...] (n, t, invert (j,next),
                      destfault, busy)
     end var
     []
     if (busy < n) then
       var p: SIGNALS in
         PCind !busy;
           PDreq (!busy, ?p);
             Distribute [...] (n, t, next,
                        destfault,
                        busy, p, 0)
       end var
     elsif zero (next) then
       SubactionGap [...] (n, t, 0)
     else
       Resolve [...] (n, t, next, 0)
     end if
   end process

   process SubactionGap [<<Bus gates>>]
           (n: Nat, t: BoolTABLE, j: Nat)
   is
     if (j == n) then
       BusIdle [...] (n, t)
     else
       PDind (!j, !subactgap);
         SubactionGap [...] (n, t, succ (j))
     end if
   end process

   process Distribute [<<Bus gates>>]
      (n: Nat,
       t, next, destfault: BoolTABLE,
       busy: Nat, p: SIGNALS, j: Nat)
   is
     if (j < n) then
       if (j == busy) then
         Distribute [...] (n, t, next,
                    destfault,
                    busy, p, succ (j))
       else (* j != busy *)
         if not (is_header (p) and
                 get (j, destfault)) then
           PDind (!j, !p);
             Distribute [...] (n, t, next,
                        destfault,
                        busy, p, succ (j))
         end if
         []
```

```
    if is_dest (p) then
      dest := any Nat;
        PDind (!j, !sig (destsig (dest)));
          Distribute [...] (n, t, next,
                     invert (j, destfault),
                     busy, p, succ (j))
    elsif is_header (p) or
          is_data (p) or
          is_ack (p)
    then
      PDind (!j, !corrupt (p));
        Distribute [...] (n, t, next,
                   destfault,
                   busy, p, succ (j))
      []
      losesignal;
        Distribute [...] (n, t, next,
                   destfault,
                   busy, p, succ (j))
    end if
    []
    if is_data (p) then
      PDind (!j, !p);
        PDind (!j, !Dummy);
          Distribute [...] (n, t, next,
                     destfault,
                     busy, p, succ (j))
    end if
    []
    PAreq (!j, !immediate)
             [not(get (j, next))];
      Distribute [...] (n, t, invert(j,next),
                 destfault,
                 busy, p, j)
  end if
else (* not (j < n) *)
  if p == EndS then
    BusBusy [...] (n, t, next,
                   destfault, n)
  else
    BusBusy [...] (n, t, next,
                   destfault, busy)
  end if
  end if
end process

process Resolve [<<Bus gates>>]
        (n: Nat, t,next: BoolTABLE, j: Nat)
is
  if (j < n) then
    if (get (j, next) == true) then
      PAcon (!j, !won);
        PCind !j;
          Resolve [...] (n,t,next,succ (j))
    else
      Resolve [...] (n,t,next,succ (j))
    end if
  else
    Resolve2 [...] (n, t, next)
  end if
end process

process Resolve2 [<<Bus gates>>]
        (n: Nat, t, next: BoolTABLE)
is
  if more (next) then
    var j: Nat in
      PDreq (?j, !EndS) [get (j, next) and
                         (j < n)];
        Resolve2 [...] (n,t,
                        invert (j,next))
    end var
  else
    var j: Nat, p: SIGNALS in
      PDreq (?j, ?p) [j < n];
        if p == EndS then
           SubactionGap [...] (n,t,0)
        else
           Distribute [...] (n,t,init (n),
                      init (n),j,p,0)
        end if
    end var
  end if
end process

end module
```