Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS

Mihaela Sighireanu, Radu Mateescu

To cite this version:


HAL Id: inria-00073516

https://inria.hal.science/inria-00073516

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS

Mihaela Sighireanu, Radu Mateescu

N° 3172
Mai 1997

THEME 1
Validation of the Link Layer Protocol
of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS

Mihaela Sighireanu*, Radu Mateescu †

Thème 1 — Réseaux et systèmes
Projet VASY Recherche et Applications

Rapport de recherche n° 3172 — Mai 1997 — 37 pages

Abstract: This paper deals with the description in E-LOTOS of the asynchronous LINK layer protocol of the IEEE-1394 Standard and its verification using model-checking. The E-LOTOS descriptions are based on both the standard and the μCRL description written by Luttik. The verifications are performed using the CADP (CÆSAR/ALDEBARAN) toolbox. We translate the E-LOTOS descriptions in LOTOS using the TRAIAN tool, and then we generate the underlying LTS models corresponding to various scenarios using the CÆSAR compiler. We formally express in the ACTL temporal logic the five correctness properties of the LINK layer protocol stated in natural language by Luttik and we verify them on the LTS models using the XTL model-checker. We detect and correct a potential deadlock caused by the ambiguous semantics of the state machines given in the standard, which can be misleading for implementors of the IEEE-1394 protocol.


(Résumé : tsyp)


* E-mail: Mihaela.Sighireanu@inria.fr
† E-mail: Radu.Mateescu@inria.fr

Unité de recherche INRIA Rhône-Alpes
655, avenue de l’Europe, 38330 MONTBONNOT ST MARTIN (France)
Téléphone : (33) 76 61 52 00 — Télécopie : (33) 76 61 52 52
Validation du protocole de la couche liaison
du bus série IEEE-1394 ("FireWire") :
une expérience avec E-LOTOS

Résumé : Ce rapport présente la description en E-LOTOS de la partie asynchrone du protocole
de la couche liaison du standard IEEE-1394 et sa vérification en utilisant l’approche basée sur les
modèles (model-checking). La description E-LOTOS est basée sur le standard et sur la description
écrite en $\mu$CRL par Luttik. Les vérifications sont effectuées en utilisant la boîte à outils CADP
(CÆSAR/ALDEBARAN). La description E-LOTOS est traduite en LOTOS à l’aide de l’outil TRAIAN
et les modèles STE sous-jacents sont générés à l’aide du compilateur CÆSAR. Nous avons exprimé en
logique temporelle ACTL les cinq propriétés de correction du protocole de la couche liaison formulées
en langage naturel par Luttik et nous les avons vérifiées sur les modèles STE à l’aide de l’évaluateur
XTL. Nous avons détecté et corrigé un blocage potentiel dû à la sémantique ambiguë des machines
d’états données dans le standard qui peut induire en erreur les implémenteurs du protocole IEEE-1394.

Mots-clé : E-LOTOS, IEEE-1394, Ingénierie des protocoles, Logique temporelle, LOTOS, Méthodes
formelles, Protocole de couche liaison, Systèmes de transitions étiquetées, Techniques de description
formelle, Validation, Vérification.
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 much as possible.

One approach that proved its usefulness is to use formal verification throughout the design process, by means of specialized tools. For this purpose, the application must be described using an appropriate high-level language like μCRL\(^1\) [GP90], LOTOS\(^2\) [ISO88], E-LOTOS\(^3\) [Que97], etc. Such descriptions provide a formal, non-ambiguous basis upon which the verification of the desired correctness properties can be attempted.

A verification method that has been extensively studied over the last years, and for which various algorithms and tools have been developed, is model-checking. In this approach, the correctness properties 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 from the early steps of the design process.

This paper deals with the formal description in E-LOTOS and verification by model-checking of the LINK layer protocol for transmission of asynchronous packets, which is part of the “FireWire” high performance serial bus defined in the IEEE-1394 Standard [IEEE95].

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 BUS layer, the LINK layer, and the TRANS 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. Section 10 describes the verifications performed on these models by means of temporal logic. Section 11 gives some concluding remarks. Finally, Annex A gives the complete E-LOTOS description of data types used in the protocol.

2 The ISO language LOTOS and the E-LOTOS language

LOTOS [ISO88] is a standardized Formal Description Technique intended for the specification of communication protocols and distributed systems. Several tutorials for LOTOS are available, e.g. [BB88, Tur93].

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” sub-languages:

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 [Gut77], more specifically on the ACTONE specification language [dMRV92].

The control part of LOTOS is based on the process algebra approach for concurrency, and appears to combine the best features of CCS [Mil89] and CSP [Hoa85].

\(^1\)micro Common Representation Language
\(^2\)Language Of Temporal Ordering Specification
\(^3\)Extended LOTOS
LOTOS has been applied to describe complex systems formally, for example: OSI TP\textsuperscript{4} [ISO92, Annex H], FTAM\textsuperscript{5} basic file protocol [LL95], etc. It has been mostly used to describe software systems, although there are recent attempts to use it for asynchronous hardware description [CGM\textsuperscript{+}96].

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 [Que97], appeared in February 1997, proposes a revised version of LOTOS, named E-LOTOS. Compared to LOTOS, the language defined in [Que97] 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 was introduced instead of guarded commands combined with choice.

- **Imperative features**: to allow an imperative-like programming style, write-once 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 [Gar95].

This case-study is based on a version of E-LOTOS proposed in [SG96], which is slightly different from the Committee Draft one. The main differences are: (a) instead of record subtyping and anonymous records, we use named records and overloading of functions and constructors; (b) the shorthand notation ‘...’ can be used in a process instantiation to elide the actual gate parameters when they are identical to the formal ones defined in the corresponding process declaration; (c) although [Que97] introduces quantitative time, the fragment of E-LOTOS we consider here is untimed.

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 [ISO95] given in [Que97]. 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 ballotig within ISO, there are no tools for E-LOTOS available yet. A straightforward approach is to translate E-LOTOS programs in LOTOS, and then use the existing tools dedicated to LOTOS. For this case-study, we used the TRAIAN tool [Viv97], a prototype translator from E-LOTOS to LOTOS, and the CADP (CÆSAR/ALDÉBARAN) toolbox [FGK\textsuperscript{+}96], which provides state-of-the-art verification features.

\textsuperscript{4} Distributed Transaction Processing
\textsuperscript{5} File Transfer, Access and Management
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 [IEEE95].

We summarize below some important features of this protocol. An extended presentation is given in [Lut97]. 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 [ANS94]. 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 [IEEE95], 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 [Lut97], we leave these facilities out of our E-LOTOS specification.

![Figure 1: The Serial Bus architecture](image-url)

The architecture of the IEEE-1394 serial bus is depicted on Figure 1 and described by the E-LOTOS specification module IEEE_1394 below. This module includes (using the import clause) the modules

RR n° 3172
defining the data types (DATA), the BUS layer (BUS), the LINK layer (LINK), and the TRANS layer (TRANS). It 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 on Figure 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, PAcron, PAreq, and PCind. Note the use of the shorthand notation ‘...‘ in place of the actual list of gates in the instantiations of the Bus, Link, and Trans processes.

specification IEEE_1394 import DATA, BUS, LINK, TRANS is
gates LDreq: LDreqType, LDcon: LDconType,
LDind: LDindType, LDres: LDresType,
PDind: PDindType, PDreq: PDreqType,
PAcom: PAcronType, PAreq: PAreqType,
PCind: Nat, arbresgap, losesignal: none
behaviour

{ Trans [...] (n, 0)
 |[[LDreq, LDcon, LDind, LDres]]
Link [...] (n, 0)
}

... ...

{ Trans [...] (n, n-1)
 |[[LDreq, LDcon, LDind, LDres]]
Link [...] (n, n-1)
}

|[[PDind, PDreq, PAcron, PAreq, PCind]]
Bus [...] (n)
endspec

The following sections refine this top-level architecture by describing the data types used in the protocol and the BUS, Link, and TRANS 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 PHYAREQ 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).
Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus (“FireWire”)  

- **SIGNAL** is a union type modeling the data packet components\(^6\) and the control signals traveling over the **BUS**.
- **SIG\_TUPLE** is the type of 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**.
- **Bool\_TABLE** 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.

In our E-LOTOS specification of the protocol, the domain of **Nat** is limited to the interval \(0 \ldots n\), where \(n\) is the number of nodes connected to the **BUS**. All other data types are finite.

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 seven pages of algebraic data types in the \(\mu\)CRL description given in [Lut97].

5 The **BUS layer behaviour**

In order to model the interactions between the **LINK** layers of several nodes, we translated in E-LOTOS the \(\mu\)CRL descriptions of the external behaviour of **BUS** given in [Lut97]. 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 on Figure 2.

![Figure 2: The structure of the fairness interval](image)

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 **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 request gap” (**arbresgap**) actions.

---
\(^6\)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.
signals sent by the \texttt{Bus}. An \texttt{arbresgap} signal is emitted when, after some subactions, the \texttt{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 \texttt{Bus} behaviour is defined by the E-Lotos module \texttt{Bus}. According to [Lut97], we represent each state of the \texttt{Bus} protocol by an E-Lotos process parameterized (at least) by the total number of nodes \( n \). To improve readability, we use the \texttt{<Bus gates>\>} shorthand notation for the following list of typed gates: \texttt{PDind: PDindType, PDreq: PDreqType, PAcon: PAcconType, PAreq: PAreqType, PCind: Nat, arbresgap: none, losesignal: none}.

\begin{verbatim}
module BUS import DATA is
  process Bus [<Bus gates>] (n: Nat) : noexit is
    BusIdle [...] (n, init (n))
  endproc
  (* ... other Bus processes *)
endmod
\end{verbatim}

**Idle state** Initially, \texttt{Bus} is idle (state \texttt{BusIdle}). The parameter \( n \) denotes the number of \texttt{Links} connected to the \texttt{Bus}. The parameter \( t \) is a boolean table recording the accesses of \texttt{Links} during a fairness interval. For each \texttt{Link} \( j \), the corresponding entry \( t[j] \) is true if the \texttt{Link} \( j \) accessed the \texttt{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 \texttt{init}.

Every time an arbitration request with parameter \texttt{fair} is received from some \texttt{Link}, \texttt{Bus} checks whether the \texttt{Link} accessed it during the present fairness interval\(^7\); if not, \texttt{Bus} grants access to the requesting \texttt{Link} by means of the \texttt{PAcon} action with parameter \texttt{won} and evolves into the \texttt{BusBusy} state.

\begin{verbatim}
process BusIdle [<Bus gates>] (n: Nat, t: BoolTABLE) : noexit is
  local id: Nat, astat: PHY_AREQ in
  PAreq (?id, ?astat) [id \& t 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)
  endif
endloc
if not (zero (t)) then
  arbresgap;
  BusIdle [...] (n, init (n))
endif
endproc
\end{verbatim}

\(^7\)Here, unlike the \texttt{\mu}Ch description, we merge \texttt{BusIdle} and \texttt{DecideIdle} into the same process \texttt{BusIdle}.

INRIA
Busy state  In the Bus Busy 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 BUS Busy state, the BUS may still accept fair arbitration requests from LINKs (PAreq actions with parameter fair), but sends negative responses to requesters (PAcom 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 End 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 to send the 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.


is

local j: Nat in
  PAreq (??, !fair) [j lt n];
  PAcom !j !lost;
  BusBusy [...] (n, t, next, destfault, busy)
endloc

if (busy lt n) then
  local p: SIGNAL in
  PCind !busy;
  PDreq (!busy, ?p);
  Distribute [...] (n, t, next, destfault, busy, p, 0)
endloc
elsif zero (next) then
  SubactionGap [...] (n, t, 0)
else
  Resolve [...] (n, t, next, 0)
endif
endproc
process SubactionGap [<<Bus gates>>] (n: Nat, t: BoolTABLE, j: Nat) : noexit is
  if (j = n) then
    BusIdle [...] (n, t)
  else
    PDind (!j, !subactgap);
  SubactionGap [...] (n, t, succ (j))
endif
endproc

**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 End, which indicates the termination of the asynchronous packet, the parameter busy is set to n.

process Distribute [<<Bus gates>>]
  (n: Nat, t, next, destfault: BoolTABLE, busy: Nat, p: SIGNAL, j: Nat)
  : noexit
is
  if (j = busy) then
    Distribute [...] (n, t, next, destfault, busy, p, succ (j))
  elseif (j lt n) and (j ne busy) then
    if not (is_header (p) and get (j, destfault)) then
      PDind (!j, !p);
    Distribute [...] (n, t, next, destfault, busy, p, succ (j))
  endif
[]
  if is_dest (p)
    choice ?dest: Nat []
PDind (!j, !sig (destsig (dest)));  
Distribute [...] (n, t, next, invert (j, destfault), 
busy, p, succ (j))
endch
elseif 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))
[]
loseedsignal;
Distribute [...] (n, t, next, destfault, busy, p, succ (j))
endif
[]
if is_data (p) then
PDind (!j, !p);
PDind (!j, !Dummy);
Distribute [...] (n, t, next, destfault, busy, p, succ (j))
endif
[]
PAreq (!j, !immediate) [not (get (j, next))];  
Distribute [...] (n, t, invert (j, next), destfault, busy, p, j)
else (* not (j lt n) *)
case p is
  End -> BusBusy [...] (n, t, next, destfault, n)
  | any -> BusBusy [...] (n, t, next, destfault, busy)
endcase
endif
endproc

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 Links recorded into the next table, Bus accepts only End signals from these Links, and eliminates them from the next table. If the remaining Link sends an End signal, Bus evolves into the SubactionGap state; otherwise Bus delivers the signal to the other Links by moving into the Distribute state.

process Resolve [<<Bus gates>>] (n: Nat, t, next: BoolTABLE, j: Nat) : noexit is
  if (j lt 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))
    endif
  else
    Resolve2 [...] (n, t, next)
  endif
endproc
process Resolve2 [<Bus gates>] (n: Nat, t, next: BoolTABLE) : noexit is
    if more (next) then
      local j: Nat in
        PDreq (?j, !End) [get (j, next) and (j lt n)];
        Resolve2 [...] (n, t, invert (j, next))
      endloc
    else
      local j: Nat, p: SIGNAL in
        PDreq (?j, ?p) [j it n];
        case p is
          End -> SubactionGap [...] (n, t, 0)
        | any -> Distribute [...] (n, t, init (n), init (n), j, p, 0)
        endcase
      endloc
    endif
endproc

6 The LINK layer behaviour

The LINK layer protocol is designed to transmit data packets over an unreliable medium, by splitting them in 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 [Lut97], 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 on Figure 3.

In the remainder of this Section, we give an E-Lotos description of the initial state and the three modes of LINK. This description is based on the µCRL description given in [Lut97], the state machine depicted in [IEE95, Figure 6-19, Page 174], 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\(^8\).

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.

The E-LOTOS processes corresponding to the initial state of LINK are given in the E-LOTOS module below. Compared to LOTOS, the main differences are:

- The gates of LINK are explicitly typed. To improve readability, in this Section we use the shorthand notation \(<\text{Link gates}>\) for the following list of typed gates: LDreq: LDreqType, LDcom: LDcomType, LDind: LDindType, LDres: LDresType, PDreq: PDreqType, PDind: PDindType, PAreq: PAreqType, PAcon: PAconType, PCind: Nat.

- The scope of variables is made explicit using ‘local’ statements.

- The equality function ‘=' is built-in, rather than user-defined.

```plaintext
module LINK import DATA is
    process Link [\(<\text{Link gates}>\)] (n, id: Nat) : noexit is
        Link0 [...] (n, id, void)
    endproc

    process Link0 [\(<\text{Link gates}>\)] (n, id: Nat, buf: SIG_TUPLE) : noexit is
        if is_void (buf) then
            local var dest: Nat, h: HEADER, d: DATA in
```

\(^8\)Here, unlike the µCRL description, we merge Link0 and Link1 into the same process Link0.
LDreq (!id, ?dest, ?h, ?d);
  Link0 [...] (n, id, quadruple (dhead, sig (destsig (dest)),
  sig (headersig (c, crc (c))),
  sig (datasig (d, crc (d)))))
endloc
else
  PAreq (!id, !fair);
  ( PAcon (!id, !won);
    Link2req [...] (n, id, buf)
  []
  PAcon (!id, !lost);
    Link0 [...] (n, id, buf)
  )
endif
[]
local var p: SIGNAL in
PDind (!id, ?p);
if (p = Start) then
  Link4 [...] (n, id, buf)
else
  Link0 [...] (n, id, buf)
endif
endloc
endproc
(* ... other Link processes *)
endmod

**Send Mode**  Being granted the access to Bus, LINK responds to every clock indication received on the PCind gate by sending a signal. 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 End. 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.

process Link2req [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit 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, !End);
  if (getdest (buf.dest) = n) then
    LDcon (!id, !broadsemt);
    Link0 [...] (n, id, void)
  else
    Link3 [...] (n, id, void)
Notice here the use of E-LOTOS built-in field projection ’.’ to access the packet fields (e.g., \texttt{buf.dest} gives the destination field of the buffer), rather than using user-defined selection functions (as in LOTOS and \texttt{µCRL}).

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 \texttt{LINK} will act as if the acknowledgement was missing. The acknowledgement packet begins with a \texttt{Start} signal, possibly preceded by any number of \texttt{Prefix} signals. When the \texttt{Start} signal arrives, \texttt{LINK} evolves into the \texttt{Link3RA} (“Receive Acknowledge”) state.

```
process Link3 [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
  local var p: SIGNAL in
  PFind (!id, ?p);
  if (p = Prefix) then
    Link3 [...] (n, id, buf)
  elseif (p = Start) then
    Link3RA [...] (n, id, buf)
  elseif (p = subactgap) then
    LDcon (!id, !ackmiss);
    Link0 [...] (n, id, buf)
  else
    LDcon (!id, !ackmiss);
    LinkWSA [...] (n, id, buf, n)
  endif
endloc
endproc
```

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

```
process Link3RA [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
  local var a: SIGNAL in
  PFind (!id, ?a);
  if (a = subactgap) then
    LDcon (!id, !ackmiss);
    Link0 [...] (n, id, buf)
  elseif is_physig (a) then
    LDcon (!id, !ackmiss);
    LinkWSA [...] (n, id, buf, n)
  else
    Link3RE [...] (n, id, buf, a)
  endif
endloc
endproc
```
process Link3RE [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE, a: SIGNAL) : noexit is
  local var e: SIGNAL in
  PDid (id, ?e);
  if valid_ack (a) and ((e = End) or (e = Prefix)) then
    LDcon (id, !ackrec (getack (a)))
  else
    LDcon (id, !ackmiss)
  endif;
  if (e = subactgap) then
    Link0 [...] (n, id, buf)
  else
    LinkWSA [...] (n, id, buf, n)
  endif
endloc
endproc

Note the use of the sequential composition operator ‘;’ of E-LOTOS, which allows a more concise description than the ‘>>’ and ‘accept’ operators of LOTOS.

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 PDid gate) before determining whether the packet is addressed to itself or to another LINK. If anything goes wrong, it waits for the next “subaction gap” signal (see the LinkWSA state) and returns to its initial state afterwards.

process Link4 [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
  local var dh: SIGNAL in
  PDid (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)
  endif
endloc
endproc

If the second signal received on the PDid 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 destined 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 (see the LinkWSA state).
process Link4DH [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
  local var dest: SIGNAL in
  PDind (!id, ?dest);
    if is_dest (dest) and (getdest (dest) = id) then
      PArq (!id, !immediate)
    endif;
    if is_dest(dest) and ((getdest(dest) = id) or (getdest(dest) = n)) then
      Link4RH [...] (n, id, buf, getdest (dest))
    elseif (dest = subactgap) then
      Link0 [...] (n, id, buf)
    else
      LinkWSA [...] (n, id, buf, n)
    endif
  endloc
endproc

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

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

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

If the packet is correctly terminated by an End 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 of the above behaviour will cause LINK to ignore the entire packet; it goes into the LinkWSA state (see below) where it waits for a “subaction gap” signal.

```plaintext
process Link4RE [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE, dest: Nat, h, d: SIGNAL) : noexit
  is
    local var e: SIGNAL in
    PBind (!id, ?e);
    if ((e = End) or (e = Prefix)) then
      if (dest = id) then
        Link4DRec [...] (n, id, buf, h, d)
      else
        Link4BRec [...] (n, id, buf, h, d)
      endif
    else
      LinkWSA [...] (n, id, buf, dest)
    endif
  endloc
endproc

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

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 behaviour (which follows strictly the state machine given in [IEE95, page 174] and the μCRL description of [Lut97]) is erroneous.

```plaintext
process Link4BRec [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE, h, d: SIGNAL) : noexit
  is
    if (getaccur (d) = check) then
      LDind (!id, !broadrec (gethead (h), getdata (d)))
    endif;
    Link0 [...] (n, id, buf)
endproc
```

**Send acknowledge mode** While waiting for TRANS to respond to a packet indication, LINK keeps BUS into the “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.
process Link5 [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
local var a: ACK, b: BOC in
LDreq (!id, ?a, ?b);
   Link6 [...] (n, id, buf, sig (acksig (a, crc (a))), b)
endloc
PCind !id;
PDreq (!id, !Prefix);
Link5 [...] (n, id, buf)
endproc

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 holds the BUS by responding to clock indications with a Prefix signal (see state Link7).

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

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.

process Link7 [<<Link gates>>] (n, id: Nat, buf: SIG_TUPLE) : noexit is
PCind !id;
PDreq (!id, !Prefix);
Link7 [...] (n, id, buf)
local var dest: Nat, h: HEADER, d: DATA in
LDreq (!id, ?dest, ?h, ?d);
Link2resp [...] (n, id, buf, quadruple (dhead, sig (destsig (dest)),
    sig (headersig (c, crc (c))),
    sig (datasig (d, crc (d)))))
endloc
endproc

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 \textsc{Link}. Therefore, if the destination signal indicates that the packet was meant for this \textsc{Link},
the arbitration confirmation must be received and \textsc{Bus} control must be terminated immediately by
sending an \texttt{End} signal.

\begin{verbatim}
process \texttt{LinkWSA [<<Link gates>>]} (n, id: Nat, buf: SIG_TUPLE, dest: Nat) : noexit is
  loop in
    local var p: SIGNAL in
    PDind (!id, ?p);
    if (p = subactgap) then
      break
    endif
  endloc
  []
  if (dest = id) then
    PAcon (!id, !won);
    PCind !id; PDreq (!id, !End);
    break
  endif
endloop;
Link0 [...] (n, id, buf)
endproc
\end{verbatim}

Notice the use of the breakable ‘loop’ construct of \E-Lotos instead of recursive processes as in
\Lotos.

\section{The TRANS layer behaviour}

In order to verify the \textsc{Link} layer protocol, we had to specify the external behaviour of \textsc{TRANS} w.r.t.
\textsc{Link}, although it was not formalized in [Lut97]. Our description of the \textsc{TRANS} behaviour is based
on the state machine diagrams and informal explanation given in [IEE95].

For each node, the \textsc{TRANS} layer provides read, write and lock transactions to the application running
on the node. Transactions use four service primitives of the \textsc{Link} layer, following to the \textsc{Oslo} connection
establishment diagram (shown in Figure 4): (1) \texttt{Request} (performed on the \texttt{LDreq} gate) is used by a
\textsc{TRANS} requester to start the transaction; (2) \texttt{Indication} (performed on the \texttt{LDind} gate) is used to
notify the \textsc{TRANS} responder of an incoming request; (3) \texttt{Response} (performed on the \texttt{LDres} gate) is
used by the \textsc{TRANS} responder to return status or data to the \textsc{TRANS} requester; (4) \texttt{Confirmation}
(performed on the \texttt{LDcon} gate) is used to notify the \textsc{TRANS} requester of the arrival of the corresponding
response.

At any time, the \textsc{TRANS} entity of a node can process outgoing (request) and incoming (response) trans-
sactions. The \textsc{TRANS} behaviour is defined by the \E-Lotos module below. The requester (\texttt{TransReq})
and the responder (\texttt{TransRes}) processes, which handle the outgoing and incoming transactions, respec-
tively, evolve in parallel and synchronize on the \texttt{LDreq} gate. To improve readability, we use the \texttt{<<Trans gates>>} shorthand notation for the following list of typed gates: \texttt{LDreq: LDreqType,}
\texttt{LDcon: LDconType, LDind: LDindType, LDres: LDresType}. 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) : noexit is
    TransReq [...] (n, id)
    |[LDreq]|
    TransRes [...] (n, id)
  endproc
  (* ... processes TransReq and TransRes *)
endmod

In the remainder of this Section, we present the TransReq and TransRes processes, considering only the part of the TRANS behaviour that is relevant w.r.t. the LINK.

**Request transaction**  When beginning an outgoing transaction, the TRANS requester sends a data request to the LINK on the LDreq gate. The request consists of a destination node identifier (dest), a header (h), and a data (d). Then, the requester waits for a confirmation on the LDcon gate, which can indicate either a broadcast completion (broadsent), or an acknowledgement of the request (ackrec), or a negative acknowledgement (ackmiss). In all cases, it returns to its initial state afterwards.

process TransReq [<<Trans gates>>] (n, id: Nat) : noexit is
  loop forever var dest: Nat, h: HEADER, d: DATA in
    LDreq (!id, ?dest, ?h, ?d) [dest le n];
    (if (dest = n) then
     LDcon (!id, !broadsent)
    else
     LDcon (!id, ackrec (any: ACK))
    endif
  []
    LDcon (!id, !ackmiss)
  )
endloop
endproc
Notice the use of the ‘loop forever’ construct of E-LOTOS, which allows to describe the cyclical behaviour of the requester in a more concise way than using recursive process calls, as in LOTOS.

**Response transaction**  The responder awaits an indication of a transaction request on the **LInd** gate. If a broadcast is indicated, the responder sends a response to the **LINK** by means of an **LRes** 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 requester is in its initial state (this is ensured by the synchronization on the **LReq** 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 responder returns to its initial state.

```plaintext
process TransRes [<<Trans gates>>] (n, id: Nat) : noexit is
    loop forever var l: LIN_DIN in
        LInd (!id, ?l);
        if is_broadrec (l) then
            LRes (!id, any: ACK, !no_op)
        else
            LRes (!id, any: ACK, !hold); (* concatenated transaction *)
            LReq (!id, any: Nat, any: HEADER, any: DATA)
            []
            LRes (!id, any: ACK, !release) (* split transaction *)
        endif
    endloop
endproc
```

Notice again the use of the ‘loop forever’ construct of E-LOTOS, and the use of ‘any’ patterns when the values exchanged are not relevant.

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

# The CADP verification toolbox

The CADP\(^9\) toolbox 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 behavioural 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 lower-level formalisms such as finite state machines and networks of communicating automata.**

---

\(^9\)CÉSAR/ALDÉBARAN Development Package

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

- **CÆSAR [GS90] and CÆSAR. ADT [Gar89]** are compilers that translate a Lotos program into a Labelled Transition System (LTS for short) describing its exhaustive behaviour. This LTS can be represented either *explicitly*, as a set of states and transitions, or *implicitly*, as a library of C functions allowing to execute the program behaviour in a controlled way.

- **Aldébaran [FKM93]** is a verification tool for comparing or minimizing LTSs with respect to (bi)simulation relations [Par81, Mil89]. 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 behavioural specifications.

We also used two new tools (not yet integrated in CADP):

- **XTL (eXecutable Temporal Language) [Mat94]** 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 informations 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 [CES86] and ACTL [NV90] have already been implemented in XTL.

- **Traian [Viv97]** is a prototype translator from E-Lotos to Lotos, which is currently under development. The current version supports a subset of E-Lotos [SG96] sufficient for this case-study. It translates into Lotos all the constructs used in this paper, namely declarations (of types, functions, and processes), simple behaviour expressions (parallel composition, choice, ‘if-then-else’, sequential composition, local declarations, loop constructs, action denotation, process calls), and simple data expressions (‘if-then-else’, local declarations, normal forms, 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 the 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 $M = \langle Q, A, T, q_{init} \rangle$ 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 $G \ V_1, \ldots, \ V_n$ where $G$ is a *gate* and $V_1, \ldots, V_n$ ($n \geq 0$) are the values exchanged (i.e., sent or received) during the rendezvous at $G$. For the *silent* action $\tau$, the value list must be empty ($n = 0$);
• $T \subseteq Q \times A \times Q$ is the transition relation. A transition $<q_1, a, q_2> \in T$ (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 $\text{Path}(q)$ the set of all paths $q = q_0 \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \ldots$ 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 to generate finite LTSs of tractable size; (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; (4) Minimization of the resulting LTSs modulo strong bisimulation [Par81] using the ALDÉBARAN tool.

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 of requests on the LDreq gate. The behaviour of the TRANS process presented in Section 7 becomes:

```plaintext
TransReq [...] (n, id)
|[[LDreq]]
|
TransReq [...] (n, id)
|
Application [LDreq]
```

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

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

**S2.** All the applications perform a single request (behaviour LDreq (!id, !dest, !h, !d); stop).

**S3.** All the applications are passive (behaviour stop), except one (e.g., node 0) that performs only $k$ broadcast requests (behaviour LDreq (!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.

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 $k = 1$ has been 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.
\begin{table}
\renewcommand{\arraystretch}{1.5}
\begin{tabular}{|c|c|c|c|c|}
\hline
sc. & $n$ & $k$ & protocol LTS states & time trans. \\
\hline
S1 & 1 & 1 & 42 & 42 \hspace{0.5cm} 0:00'53'' \\
   & 2 & 1 & 1,552 & 1,836 \hspace{0.5cm} 0:01'00'' \\
   & 3 & 1 & 85,780 & 109,775 \hspace{0.5cm} 0:15'41'' \\
S2 & 2 & 1 & 658,468 & 822,453 \hspace{0.5cm} 0:55'45'' \\
   & 3 & 1 & >6,942,719 & >17,226,065 \hspace{0.5cm} 9:34'31'' \\
S3 & 2 & 2 & 4,894 & 6,716 \hspace{0.5cm} 0:01'07'' \\
   & 3 & 2 & 26,136 & 37,975 \hspace{0.5cm} 0:01'22'' \\
   & 4 & 2 & 76,600 & 115,770 \hspace{0.5cm} 0:03'30'' \\
\hline
\end{tabular}
\caption{Results of Ltss generation for IEEE-1394}
\end{table}

The state explosion problem prevented us from studying more complex scenarios with a greater number of nodes and/or requests. We could identify 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 behaviour.

\section{Verification using temporal logic}

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 Section 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 [NV90], 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, and finally, we discuss the verification results obtained.
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 $\alpha$ of this logic have the following syntax:

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

The construction $\{G V_1, ..., V_n\}$ 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 purposes, and unlike the original ACTL logic, we also allow action patterns (of the form $\{\tau\}$) matching $\tau$-actions.

Of course, the usual derived boolean operators are also allowed: we write $\text{false}$ for $\neg \text{true}$, $\alpha \lor \alpha'$ for $\neg(\neg\alpha \land \neg\alpha')$, and $\alpha \implies \alpha'$ for $\neg\alpha \lor \alpha'$.

The action formulas $\alpha$ 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 $\alpha$ 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:

$$
\begin{align*}
& a \models \text{true} \quad \text{always;} \\
& a \models \{G V_1, ..., V_n\} \text{ iff } a = G V_1, ..., V_n; \\
& a \models \neg \alpha \quad \text{ iff } a \not\models \alpha; \\
& a \models \alpha \land \alpha' \quad \text{ iff } a \models \alpha \text{ and } a \models \alpha'.
\end{align*}
$$

The formulas $\varphi$ of the ACTL fragment we used are defined by the following syntax:

$$\varphi ::= \text{true} \mid \neg \varphi \mid \varphi \land \varphi' \mid \text{init} \mid \text{EX}_a \varphi \mid \text{AX}_a \varphi \mid \text{E} [\varphi \cup \varphi'] \mid \text{E} [\varphi \cap \varphi'] \mid \text{A} [\varphi \cup \varphi'] \mid \text{A} [\varphi \cap \varphi']$$

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

$$
\begin{align*}
& q \models \text{true} \quad \text{always;} \\
& q \models \neg \varphi \quad \text{ iff } q \not\models \varphi; \\
& q \models \varphi \land \varphi' \quad \text{ iff } q \models \varphi \text{ and } q \models \varphi'; \\
& q \models \text{init} \quad \text{ iff } q = q_{\text{init}}; \\
& q \models \text{EX}_a \varphi \quad \text{ iff } \exists q' \in T \text{ such that } a \models \alpha \text{ and } q' \models \varphi; \\
& q \models \text{AX}_a \varphi \quad \text{ iff } \forall q' \in T \text{, } a \models \alpha \text{ and } q' \models \varphi; \\
& q \models \text{E} [\varphi \cup \varphi'] \quad \text{ iff } \exists q (= q_0) \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} ... \in \text{Path}(q), \exists k \geq 0 \text{ such that } q_k \models \varphi' \text{ and } \forall i \in [0; k - 1], q_i \models \varphi \text{ and } a_i \models \alpha; \\
& q \models \text{E} [\varphi \cap \varphi'] \quad \text{ iff } \exists q (= q_0) \overset{a_1}{\rightarrow} q_1 \overset{a_2}{\rightarrow} ... \in \text{Path}(q), \exists k > 0 \text{ such that } q_k \models \varphi' \text{ and } \forall i \in [0; k - 1], q_i \models \varphi \text{ and }
\end{align*}
$$
Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus (“FireWire”)

\[ q \models A[\varphi_0 U \varphi'] \quad \text{iff} \quad \forall j \in [0; k - 2], a_j \models \alpha \text{ and } a_{k-1} \models \alpha'; \\
q(= q_0) \xrightarrow{a_0} q_1 \xrightarrow{a_1} \ldots \in \text{Path}(q), \\
\exists k \geq 0 \text{ such that } q_k \models \varphi' \text{ and } \forall i \in [0; k - 1], q_i \models \varphi \text{ and } a_i \models \alpha; \\
q \models A[\varphi_0 U_a \varphi'] \quad \text{iff} \quad \forall j \in [0; k - 2], a_j \models \alpha \text{ and } a_{k-1} \models \alpha'.
\]

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

Besides the usual derived boolean operators, we allow the following useful derived modalities:

\[
\langle \alpha \rangle \varphi = \text{EX}_\alpha \varphi \\
[\alpha] \varphi = \neg \langle \alpha \rangle \neg \varphi \\
\text{EF}_\alpha \varphi = E [\text{true}_\alpha U \varphi] \\
\text{AF}_\alpha \varphi = A [\text{true}_\alpha U \varphi] \\
\text{EG}_\alpha \varphi = \neg \text{AF}_\alpha \neg \varphi \\
\text{AG}_\alpha \varphi = \neg \text{EF}_\alpha \neg \varphi
\]

The \( \langle \alpha \rangle \varphi \) and \( [\alpha] \varphi \) operators are the well-known Hennessy-Milner modalities [HM85]. A state \( q \) satisfies \( \langle \alpha \rangle \varphi \) (resp. \( [\alpha] \varphi \)) if some (resp. all) of its direct successors reached after an action satisfying \( \alpha \) satisfies (resp. satisfy) \( \varphi \). A state \( q \) satisfies \( \text{EF}_\alpha \varphi \) (resp. \( \text{AF}_\alpha \varphi \)) if some path (resp. all paths) issued from \( q \) leads (resp. lead) via actions satisfying \( \alpha \) to a state satisfying \( \varphi \). A state \( q \) satisfies \( \text{EG}_\alpha \varphi \) (resp. \( \text{AG}_\alpha \varphi \)) if for some path (resp. all paths) issued from \( q \), every prefix consisting of actions that satisfy \( \alpha \) leads to a state satisfying \( \varphi \).

10.2 The correctness properties

The expected functioning of the IEEE-1394 LINK layer protocol was informally characterized by Luttik [Lut97] in the form of five correctness requirements stated in natural language. Here, 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 Section 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, whenever possible, we will use the shorthand notation “INEV(\( \alpha_1, \alpha_2 \)) = A [true_{\alpha_1} U_{\alpha_2} true]”, meaning that the program eventually performs an action satisfying \( \alpha_2 \), possibly preceded only by actions satisfying \( \alpha_1 \).

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 behaviours 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 behaviours allowed us to identify these “correct” terminating states: they can occur only after an “arbitration reset gap” signal (action pattern arbresetgap) followed by 0 or more confirmations that are sent back to the TRANS layer (action pattern LDecomany). 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.

\[
\text{init} \iff \neg \text{EF}_{\text{true}} \langle \neg (\text{arbresgap} \lor \text{LDcon}_{\text{any}}) \rangle \quad \text{EF}_{\text{LDcon}_{\text{any}}}[\text{true}] \quad \text{false}
\]

**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.

\[
\begin{align*}
AG_{\text{true}}[\text{PDInd}_{\text{any,sgap}}] \\
AG_{\neg (\text{PDInd}_{\text{any,sgap}} \lor \text{LDcon}_{\text{any}})}[\text{LDcon}_{\text{any}}] \\
AG_{\neg (\text{PDInd}_{\text{any,sgap}} \lor \text{LDcon}_{\text{any}})}[\text{LDcon}_{\text{any}}] \\
AG_{\text{PDInd}_{\text{any,sgap}}}[\text{LDcon}_{\text{any}}] \quad \text{false}
\end{align*}
\]

**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 request on the PAreq gate (action pattern PAreq_{id}) 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}).

\[
\begin{align*}
AG_{\text{true}}[\text{LDreq}_{id}] \\
AG_{\neg (\text{PDInd}_{id,sgap} \lor \text{arbresgap} \lor \text{LDcon}_{id})}[\text{PDInd}_{id,sgap}] \\
AG_{\neg (\text{PAreq}_{id} \lor \text{arbresgap})}[\text{PAreq}_{id}] \\
\text{INEV}(\neg \text{arbresgap}, \text{LDcon}_{id})
\end{align*}
\]

Here we added the predicate “\(\neg \text{arbresgap}\)” as first argument to the INEV operator (and also as part of the action predicates guarding the last two AG operators) in order to ensure that we refer to the same fairness interval (i.e., no arbresgap action occurred meanwhile).

**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}).

\[
AG_{\text{true}}[\text{PAreq}_{id,immediate}] \quad \text{INEV}(\neg \text{PAreq}_{id,immediate}, \text{PAcon}_{id,won})
\]

**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.

\[
\begin{align*}
AG_{\text{true}}[\text{arbresgap}] \\
AG_{\neg \text{arbresgap}}[\text{PAreq}_{id,fair}][\text{PAcon}_{id,won}] \\
AG_{\neg \text{arbresgap}}[\text{PAreq}_{id,fair}][\text{PAcon}_{id,won}] \quad \text{false}
\end{align*}
\]

Notice again the predicate “\(\neg \text{arbresgap}\)” used in the last two AG operators in order to ensure that no “arbitration reset gap” signal occurred since the one matched by the first box modality.

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

### 10.3 The verification results

The five temporal logic formulas given in Section 10.2 were evaluated on the LTSS corresponding to the scenarios S1, S2, and S3 using the XTL [Mat94] 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 Section 10.2 are almost identical to those written in the XTL source code.

The verifications were performed for all the LTSSs produced by model generation (see Section 9). For each LTSS, the time needed for evaluating the five formulas was less than one minute.

Properties 2 through 5 are true on all scenarios.

Property 1 is false on all scenarios, meaning that an unexpected deadlock occurs in the protocol. We obtained a counter-example by considering the following formula, disjunct from property 1:

$$\text{init} \implies \mathbf{EF} \text{true} \ (\neg(\text{arbresgap} \lor \text{LDcon}_\text{any})) \ \texttt{[true]} \ \text{false}$$

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

$$q_{\text{init}} \xrightarrow{a_0} q_{1} \xrightarrow{a_1} \ldots \xrightarrow{a_{l-1}} q_{l} \xrightarrow{a_{l}} q_{\text{deadlock}}$$

where $$q_{l} \models \neg(\text{arbresgap} \lor \text{LDcon}_\text{any})$$. 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 Section 6). Although both the state machine of LINK and the µCRL description [Lut97] do not specify the LDres action, the IEEE-1394 Standard [IEE95] 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” (page 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 correct version of the Link4BRec body is given below. This behaviour corrects the state machine given in [IEE95, page 174] and the µCRL description and is compatible with the explanatory text of the LINK services given at pages 147–148 of the IEEE-1394 Standard.

```plaintext
if (getdrcr (d) = check) then
    LDind (tid, !broadcast (h), getdata (d));
    LDres (!id, any ACK, !no_op) (* added to avoid deadlock *)
endif;
Link0 [...] (n, id, buf)
```

Using this correct version of the LINK, we generated again the models for all scenarios given in Section 9. Since the sizes of the new LTSSs are very close to the previous ones, we do not give them here.

On the new LTSS, all the five correctness properties have been checked to be true.

### 11 Conclusion

In this paper, we 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 [IEE95] 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 \( \mu \text{CRL} \) by Luttik [Lut97]. The description of the TRANS layer is based on the state machines and text explanations given in [IEE95].

The E-LOTOS description we obtained has 7 pages instead of 13 pages of \( \mu \text{CRL} \) in [Lut97]. Especially for the data types, the gain in conciseness is significant: 4 pages of E-LOTOS instead of 8 pages of \( \mu \text{CRL} \). Also, it is worth noticing that E-LOTOS allows a clearer and more readable description of the behaviour expressions than LOTOS and \( \mu \text{CRL} \). 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 Labelled Transition Systems (LTSS) of the protocol by translating the E-LOTOS descriptions in LOTOS using the TRAIAN compiler, and then using the C\textsc{esar} and \textsc{cesar.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 X\textsc{tl} prototype model-checker. We expressed in the ACTL temporal logic a set of five correctness properties given in natural language by Luttik [Lut97] 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 \( \mu \text{CRL} \) 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 used up 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 behaviour, in order to avoid state explosion\(^\text{10}\). Although we were not able to generate the model for an infinite behaviour of the TRANS, the search for appropriate abstractions increased our confidence on the good functioning of the protocol obtained after correction.

This experience enforced our opinion that formal methods are very useful to the design and development of complex, critical applications. Using a formal approach, one can benefit not only of a disciplined methodology avoiding the ambiguities that may occur in semi-formal descriptions, but also of a basis for exhaustive verification.

**Acknowledgements**

Thanks are due to Hubert Garavel and Charles Pecheur 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.

---

\(^{10}\text{It is worth noticing that the tools are still under improvement; we hope to obtain better results in the future.}\)
A Description of the data structures

This Annex presents the E-LOTOS module DATA defining the data types used in the protocol. The types Nat and Bool are predefined E-LOTOS types for natural numbers and booleans. The auxiliary module ENUM2 is used to define in a uniform manner the types CHECK, DATA, HEADER, ACK, PHY_REQ, and PHY_ACONF, which are all enumerated types with two values.

module ENUM2 is
  type ENUM2 is v1 | v2 endtype
endmod

module DATA
  import
    ENUM2 renaming (CHECK    for ENUM2, bottom for v1, check for v2),
    ENUM2 renaming (DATA     for ENUM2, d1 for v1, d2 for v2),
    ENUM2 renaming (HEADER   for ENUM2, h1 for v1, h2 for v2),
    ENUM2 renaming (ACK      for ENUM2, a1 for v1, a2 for v2),
    ENUM2 renaming (PHY_REQ  for ENUM2, fair for v1, immediate for v2),
    ENUM2 renaming (PHY_ACONF for ENUM2, won for v1, lost for v2)
  is
    function crc (d: DATA) : CHECK is check endfun

    function crc (h: HEADER) : CHECK is check endfun

    function crc (a: ACK) : CHECK is check endfun

    type BOC is release | hold | no_op endtype

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

    type DESTSIG is destsig (dest: Nat) endtype

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

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

    type SIGNAL is ACK, ACKSIG,
      sig (dest: DESTSIG) | sig (h: HEADERSIG)
      | sig (d: DATASIG)    | sig (a: ACKSIG)
      | dhead
      | Start | End | Prefix
      | subactgap | Dummy
    endtype

    function is_dest (x: SIGNAL) : Bool is
      case x is
        sig (?xdest: DESTSIG) -> true
        | otherwise -> false
      endcase
    endfun
function is_header (x: SIGNAL) : Bool is
  case x is
    sig (?xsigh: HEADERSIG) -> true
    otherwise -> false
  endcase
endfun

function is_data (x: SIGNAL) : Bool is
  case x is
    sig (?xsigd: DATASIG) -> true
    otherwise -> false
  endcase
endfun

function is_ack (x: SIGNAL) : Bool is
  case x is
    sig (?xsiga: ACKSIG) -> true
    otherwise -> false
  endcase
endfun

function is_physig (x: SIGNAL) : Bool is
  case
    x::Start or x::End or x::Prefix or x::subactgap -> true
    otherwise -> false
  endcase
endfun

function valid_ack (x: SIGNAL) : Bool is
  case x is
    sig (acksig (?xa: ACK, ?xc: CHECK)) -> xc = check
    otherwise -> false
  endcase
endfun

function valid_hpart (x: SIGNAL) : Bool is
  case x is
    sig (headsig (?xh: HEADER, ?xc: CHECK)) -> xc = check
    otherwise -> false
  endcase
endfun

function getdest (x: SIGNAL) : Nat is
  case x is
    sig (destsig (?xn: Nat)) -> xn
  endcase
endfun
function gethead (x: SIGNAL) : HEADER is
  case x is
    sig (headersig (?xh: HEADER, ?xc: CHECK)) -> xh
  endcase
endfun

function getdcrc (x: SIGNAL) : CHECK is
  case x is
    sig (datasig (?xd: DATA, ?xc: CHECK)) -> xc
  endcase
endfun

function getdata (x: SIGNAL) : DATA is
  case x is
    sig (datasig (?xd: DATA, ?xc: CHECK)) -> xd
  endcase
endfun

function getack (x: SIGNAL) : ACK is
  case x is
    sig (acksig (?xa: ACK, ?xc: CHECK)) -> xa
  endcase
endfun

function corrupt (x: SIGNAL) : SIGNAL is
  case x is
    sig (headersig (?xh: HEADER, ?xc: CHECK)) -> sig (headersig(xh, bottom))
    | sig (datasig (?xd: DATA, ?xc: CHECK)) -> sig (datasig (xd, bottom))
    | sig (acksig (?xa: ACK, ?xc: CHECK)) -> sig (acksig (xa, bottom))
    | otherwise -> x
  endcase
endfun

type SIG_TUPLE is
  quadruple (dh: SIGNAL, dest: SIGNAL, h: SIGNAL, d: SIGNAL)
  | void
endtype

function is_void (x: SIG_TUPLE) : Bool is
  case x is
    void -> true
    | otherwise -> false
  endcase
endfun

type LIN_DCONF is
  ackrec (a: ACK) | ackmiss | broadsent
endtype
type LIN_DIND is
  good (h: HEADER, d: DATA)
| broadrec (h: HEADER, d: DATA)
| dcrc_err (h: HEADER)
endtype

function is_broadcast (x: LIN_DIND) : Bool is
  case x is
    broadrec (?xh: HEADER, ?xd: DATA) -> true
  | otherwise -> false
  endcase
endfun
type LDreqType is (id: Nat, dest: Nat, h: HEADER, d: DATA) endtype
type LDcomType is (id: Nat, dc: LIN_DCONF) endtype
type LDindType is (id: Nat, di: LIN_DIND) endtype
type LDresType is (id: Nat, a: ACK, b: BOC) endtype
type PDreqType is (id: Nat, s: SIGNAL) endtype
type PDindType is (id: Nat, s: SIGNAL) endtype
type PAreqType is (id: Nat, ar: PHY_AREQ) endtype
type PAcomType is (id: Nat, ac: PHY_ACONF) endtype
type Boolean is
  empty
| btable (x: Nat, b: Bool, t: Boolean) endtype

function init (x: Nat) : Booltable is
  case x is
    0 -> empty
  | otherwise -> btable (x-1, false, init (x-1))
  endcase
endfunc

function invert (x: Nat, t: Boolean) : Boolean is
  case t is
    empty -> empty
  | btable (?xnat: Nat, ?xbool: Bool, ?xbt: Boolean) ->
    if (xnat = x) then
      btable (xnat, not (xbool), xbt)
    else
      btable (xnat, xbool, invert (x, xbt)
endif
endcase
endfunc

function get (x: Nat, t: BoolTABLE) : Bool is
  case t is
    empty -> true
    | btable (?xnat: Nat, ?xbool: Bool, ?xbt: BoolTABLE) ->
      if (xnat = x) then
        xbool
      else
        get (x, xbt)
      endif
  endcase
endfunc

function zero (t: BoolTABLE) : Bool is
  case t is
    empty -> true
    | otherwise -> false
  endcase
endfunc

function one (t: BoolTABLE) : Bool is
  case t is
    empty -> false
    | btable (any: Nat, any: Bool, ?xbt: BoolTABLE) -> zero (xbt)
  endcase
endfunc

function more (t: BoolTABLE) : Bool is
  not (zero (t)) and not (one (t))
endfunc
endmod

References


