Four Formal Models of IEEE 1394 Link Layer

Hubert Garavel
Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France
hubert.garavel@inria.fr

Bas Luttik
Eindhoven University of Technology, The Netherlands
s.p.luttik@tue.nl

We revisit the IEEE 1394 high-performance serial bus ("FireWire"), which became a success story in formal methods after three PhD students, by using process algebra and model checking, detected a deadlock error in this IEEE standard. We present four formal models for the asynchronous mode of the Link Layer of IEEE 1394: the original model in μCRL, a simplified model in mCRL2, a revised model in LOTOS, and a novel model in LNT.

1 Introduction

IEEE 1394 (also called "FireWire") is an interface standard that specifies a serial bus architecture for high-speed communications. It can connect up to 63 peripherals in a tree or daisy-chain topology, and can perform both asynchronous and isochronous transfers simultaneously. It was developed between 1986 and 1995 by a large consortium gathering Apple, Panasonic, Philips, Sony, and many others contributors. This work resulted in an IEEE standard [43], followed by integration in many industrial products.

In the framework of the COST-247 action [22], a pan-European academic collaboration that took place between 1994 and 1997, the asynchronous mode of the link layer protocol of IEEE 1394 was selected as an interesting case study for formal methods. This protocol, which was close to being standardized, was thus studied by several young scientists at this time. At CWI Amsterdam, Bas Luttik developed a formal model [26, 27] in the μCRL language [19, 15] and stated five correctness properties that the protocol should satisfy. At INRIA Grenoble, Mihaela Sighireanu translated this model to LOTOS [20] and, using the XTL model checker [29] with the help of Radu Mateescu, discovered that the deadlock-freeness property did not hold, i.e., that the protocol could enter a deadlock state after a specific sequence of 50 transitions [37, 38, 39]. A detailed report about this bug, which would have been difficult to detect using step-by-step simulation or testing, can be found in [41]. The link layer protocol was also studied using theorem proving at the Universities of Kiel and Eindhoven by Lars Kühne, Jozef Hooman, and Willem-Paul de Roever [23].

Although the IEEE 1934 serial bus is no longer used today (deployed in Apple products from 1999 to 2016, it has been gradually replaced by USB 2, USB 3, and Gigabit Ethernet), it is an inspiring example for the formal methods community. From a historical perspective, it is a striking success story where three doctoral students discovered in a few weeks an unexpected deadlock in an IEEE standard designed and scrutinized over ten years by one hundred experts. Also, numerous research papers have been devoted to another aspect of IEEE 1934, its leader election algorithm ("root contention protocol"), the verification of which involves parameters, probabilities, and real time [35, 30, 33, 47, 7, 28, 48, 31, 34, 42, 4, 2, 24, 25, 32, 46, 49, 5].

F. Lang, M. Volk (Eds):
Models for Formal Analysis of Real Systems (MARS 2024)
EPTCS 399, 2024, pp. 21–100, doi:10.4204/EPTCS.399.5

Concerning the link layer protocol, formal methods evolved since 1997, as the μCRL and LOTOS languages have been replaced by newer languages, respectively mCRL2 [17, 16, 18, 1] and LNT [13, 10, 12, 36, 3], a descendent of the E-LOTOS standard [21]. Therefore, twenty-five years after, we revisit this case study to present, along with the original μCRL model, three companion models: a model written in mCRL2 by Jan Friso Groote, a recent revision of the LOTOS model developed by M. Sighireanu, and a novel model written in LNT.

The present article is organized as follows. Section 2 gives an overview of the IEEE 1394 architecture and explains the behaviour of the Link layer and neighbour layers. Section 3 presents four formal models in μCRL, mCRL2, LOTOS, and LNT, and discusses their main features from a modelling point of view — the models themselves being fully provided in Annexes A to D. Section 4 briefly reports about the verification (model checking and equivalence checking) done on these models. Finally, Section 5 gives a few concluding remarks.

2 IEEE 1394 bus

In this section, we present a description of IEEE 1394 that bridges the gap between the general description given in the IEEE standard [43] and the four formal models provided in the present article. The text in this section is based upon the technical report [26] in which the μCRL model first appeared — actually, this model was developed from a draft version [44] of the IEEE standard, but we believe that there is no significant difference between the draft and the standard in this respect.

First, we present the architecture as defined in the standard. Then, we focus our attention on the link layer of the protocol, the behaviour of which is our primary modelling purpose. To provide a comprehensive description of the link layer interacting with its environment, we will need to include the external functional behaviour of the physical layer, and so that is described too.

2.1 Architecture

The IEEE 1394 standard deals both with the physical requirements and the protocol of the bus. The main feature of the standard is that it supports two modes of transaction: an *asynchronous mode* and an *isochronous mode*.

In asynchronous mode, one party (the sender) can send a message of arbitrary length to some other party (the receiver). Such a message may be sent at an arbitrary moment after the sender has gained access to the bus; the only timing restriction is that the interval during which a node may have access to the bus is bounded. In this mode, the receiver must confirm the receipt of the message by sending an acknowledgement.

In isochronous mode the sender is obliged to send messages at fixed rates, and messages are not acknowledged. This service is useful for fast transmission of large amounts of data (e.g., audio/video streams), if certainty at the side of the sender about the receipt of the data by the receiver is not important, whereas the arrival of the data at a constant rate is.

The IEEE 1394 serial bus architecture is roughly as depicted in Figure 1. It consists of a number of nodes (addressable entities that run their own part of the protocol) connected by a serial cable.

The protocol describing the behaviour of a node in asynchronous mode distinguishes three layers:

1. The *transaction layer* (the upper layer, indicated by TRANS in Figure 1) offers three types of transactions to the application(s) running on the node: *read transactions* (read data from another node), *write transactions* (write data to another node), and *lock transactions* (have some of its own

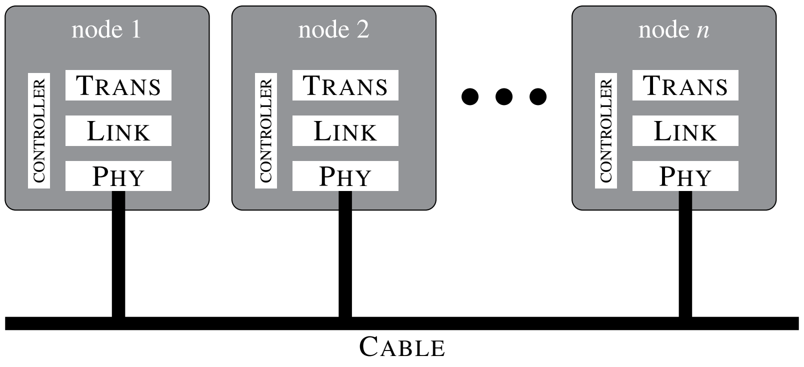

Figure 1: IEEE 1394 architecture

data processed by another node after which it is transferred back). Such transactions consist of a request and a response; the transaction layer can both handle *concatenated response* transactions (response follows request immediately) and *split* transactions (response not necessarily follows immediately on the request it belongs to).

2. The *link layer* (the middle layer, indicated by LINK in Figure 1) forms the interface between the transaction layer and the physical components of the bus (consisting of the physical layers, which are connected to each other by a serial cable). The link layer provides two types of services to the transaction layer:

Data request/response: By means of a LINK *data request*, the transaction layer instructs the link layer to send a packet to some particular node or to broadcast a packet to all other nodes. The transaction layer must react on a packet addressed to it by sending an acknowledge packet by means of a LINK *data response*.

Data indication/confirmation: By means of a LINK *data indication*, the link layer indicates the arrival of data (either request or response data). The receipt of an acknowledge packet is indicated to the transaction layer by means of a LINK *data confirmation*.

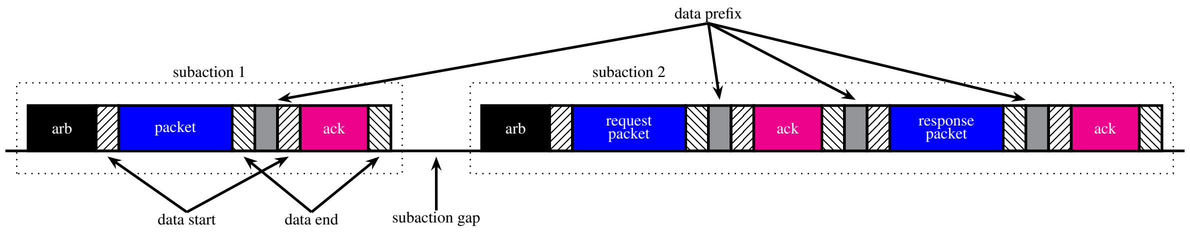

Figure 2: Subactions

The link layer divides the stream of data that it receives from the physical layer into an alternating sequence of *subactions* and *subaction gaps*, the latter being time intervals with a specified minimal length during which serial cable resides in an idle state (see Figure 2). A subaction either consists of a single packet (in case of a *split transaction*, see subaction 1) or of two packets (in case of a *concatenated response transaction*, see subaction 2). Within each subaction, a packet is delimited by special *data start* and *data end* signals[1]; the gap between two packets within a subaction must be filled with *data prefix* signals in order to distinguish these gaps from the subaction gaps.

Before a packet can be sent, the link layer must first gain access by issuing an arbitration procedure. Moreover, the link layer must transform the requests of the transaction layer into a certain packet format, computing and attaching checksums to parts of the data to be transmitted. It also decides whether incoming packets have been received properly by verifying the attached checksums. Every packet that is sent by any of the nodes is received by the link layer of every node. If a link layer determines that the packet was indeed addressed to the node it is part of, then it forwards the contents of the packet to the transaction layer. The link layer also handles the sending and receiving of acknowledgements.

3. The physical connection between a node and the serial line is called the physical layer (the lower layer, indicated by PHY in Figure 1). It listens to and puts signals on the serial cable, measures the lengths of the time intervals during which the cable resides in an idle state, and determines, together with the other physical layers, which node has control over the cable (arbitration). It provides the following services to the link layer:

Arbitration request/confirmation: The link layer instructs the physical layer to start an arbitration procedure by means of a PHY *arbitration request*. The result of this procedure (either `won` or `lost`) is communicated to the link layer by means of a PHY *arbitration confirmation*.

Data request/indication: The link layer instructs the physical layer to put some signal on the cable by means of a PHY *data request*. The physical layer indicates to the link layer the detection of a signal on the cable (or information about the status of the cable) by means of a PHY *data indication*.

Clock indication: To notify the link layer that it can (and should) put a signal on the cable, the physical layer communicates a PHY *clock indication*.

According to [43], there is also a so-called *node controller* that can influence each of the three layers. Since, in asynchronous mode, the role of this node controller is restricted to the ability to reset each of the three layers (force them into their initial state), we will not consider the node controller in this paper.

2.2 Link layer

We proceed to describe in more detail the behaviour in asynchronous mode of the link layer (the middle layer of the three-layered protocol), which is responsible for the construction of packets, the transmission of these over a serial (one-bit) line to other parties, and the computation and verification of checksums.

We model the process behaviour of the link layer according to the state machine depicted in [43, Figure 6-19, Page 170] and the accompanying informal explanation. The part of the state machine defining the behaviour in asynchronous mode has eight states Ln ($0 \leq n \leq 7$).

The link layer processes maintain a buffer (initially empty) to store a pending request from the transaction layer.

In its initial state, the link layer can either receive a data request from the transaction layer or a data indication from the PHY layer.

At a data request, a packet is constructed from the parameters that have been put into the buffer by the transaction layer. The link layer process then starts a fair arbitration procedure to gain access to the bus. If it wins the arbitration, then the underlying physical layer controls the cable and the link layer

[1]These and other "signals" of the link layer correspond to analog signals detected or emitted by the physical layer.

enters *send mode* (see below). However, it may also happen that the physical layer indicates the arrival of data: the packet to be sent is then stored in the buffer and the data is received first.

At a data indication, it must be checked whether the received signal is a `Start` signal. If so, this means that some other node has control over the cable and is sending a packet; the incoming packet must be received in *receive mode*. Otherwise, the signal (which is not a `Start` signal) can be ignored.

Send mode. As soon as a node has gained control over the cable, its physical layer starts emitting clock indications to inform the link layer that it should send a signal. The link layer must respond to every such clock indication and send the entire packet, one signal at a time, delimited by a `Start` and an `End` signal. The `End` signal also notifies the physical layer that the link layer is done sending the packet; it will cease to send clock indications. Depending on the value of the destination field, the link layer either informs the transaction layer that a broadcast packet was sent properly, or that it must wait for an acknowledge packet.

The acknowledge packet must arrive within some specific amount of time: if a subaction gap (`subactgap` signal) occurs before an acknowledgement with valid checksum has been received entirely (i.e., up to and including the terminating `End` signal), then the link layer will act as if the acknowledgement is missing (an acknowledge packet can be identified by its length; it consists of one signal). When a `Start` signal has been received, then the link layer expects to receive an acknowledge signal. If the next signal is indeed a data signal, then the link layer receives the terminating `End` signal, checks the validity of the received acknowledge signal, and sends an *acknowledgement received* (`ackrec`) to the transaction layer. If, instead, another data signal arrives, or if there is no terminating `End` signal, or if the acknowledge packet is invalid, then the link layer sends *acknowledgement missing* (`ackmiss`) to the transaction layer. Both in case of failure and in case of success, the link layer does wait for an indication of the physical layer that a subaction gap has occurred, before it returns to its initial state. Of course, if a subaction gap interferes in the above described behaviour, then the link layer should immediately send an `ackmiss` and return to its initial state.

Receive mode. If the link layer receives a `Start` signal, it enters *receive mode*, expecting to see a packet being put on the bus by some other link layer. Asynchronous packets consist of four signals. The link layer must receive at least two signals before it can determine whether the packet is addressed to it.

If it only receives one signal followed by a terminating `End`, this is an acknowledge packet, which should be ignored: the link layer will wait for the next subaction gap and return to the initial state.

If the second signal is indeed a destination signal, the link layer must check whether the incoming packet is either a packet addressed to it, or a broadcast packet, or a packet for some other node. In the first case, the link layer must notify the physical layer that it wants access to the bus as soon as the packet has been received entirely, in anticipation of sending an acknowledgement. This is done by means of an `immediate` arbitration request. Broadcast packets, however, are not acknowledged; so, in the second case, no such request is needed. In the third case, the link layer should completely ignore the packet and return to the initial state at the next subaction gap.

The third signal is expected to be a header signal, and the fourth signal should be a data signal. If the packet is correctly terminated by either an `End` signal or a `Prefix` signal, then the packet is forwarded to the transaction layer, either as a broadcast packet or as a packet that was addressed to this node. In both cases, the data checksum is verified. Observe that, in the broadcast case, a packet with an invalid data checksum is ignored. In the other case, the packet will have to be acknowledged, so upon winning a PHY *Arbitration confirmation*, the link layer continues in *send acknowledgement mode*.

Any deviation of the above described procedure will cause the link layer to ignore the packet; it will wait for a subaction gap and then returns to the initial state. Since an `immediate` arbitration request may have been dispatched, a PHY *Arbitration confirmation* of `won` may still arrive. In such a case, the link layer is granted access to the bus, but does not need to send an acknowledgement. Therefore, if the destination signal indicated that the packet was meant for this node, the arbitration confirmation must be received, and control over the cable must be terminated immediately by sending an `End` signal.

Send acknowledgement mode. While the link layer is waiting for the transaction layer to respond to a data indication with the proper acknowledgement code, it must keep the cable busy by sending a `Prefix` signal at every clock indication; this is to avoid the occurrence of a subaction gap. Depending on the type of the received packet, the transaction layer may need to issue a so-called *concatenated response* (for instance, the packet was a read request and the transaction layer immediately wants to send the requested data to the requesting node). By means of a data response, the transaction layer communicates the proper acknowledgement, as well as one of the values `release` or `hold`. The former means that no concatenated response is requested and that, after sending the acknowledgement, the link layer may release the bus and return to its initial state. The latter means that a concatenated response is requested and that the link layer should maintain control over the bus after sending the acknowledgement packet by responding to clock indications with `Prefix` signals. Upon a data request, the link layer can then go into *send mode* immediately.

2.3 Physical layer

To simulate and analyse the interaction of the link layers of n nodes, we need to model the external behaviour of underlying n physical layers connected by a cable, which, together, we shall refer to as the *bus*.

The bus needs to keep track of which of the n nodes have had control over the bus during a so-called *fairness interval*; to this aim, it maintains a table of n Booleans. During a fairness interval, each node is allowed to gain control over the bus at most once, by means of a `fair` arbitration request. It may also access the bus more than once as a consequence of an `immediate` arbitration request. As soon as the bus has been idle for some specified amount of time and at least one link layer has got access during the running fairness interval, an *arbitration reset gap* occurs to indicate that every node may, again, be granted access through `fair` arbitration. The time interval that the bus must idle before such an arbitration reset gap may occur should be longer than that of a subaction gap.

When the bus is in idle state and the link layer of some node requests arbitration, the bus enters *decision mode*: it checks whether the requesting node already got access during the present fairness interval. If not, the bus confirms the arbitration request by indicating that the node has `won` arbitration and evolves into a *busy* state; otherwise, the bus indicates that the arbitration is `lost`.

When the bus is in busy state, it records which node has control over the bus, and which nodes have requested immediate arbitration. In this state, the bus may still receive `fair` arbitration requests, but they will be confirmed by reporting that the arbitration was `lost`. The node that must send a response to the packet put on the bus will issue an `immediate` arbitration request. No confirmation is sent, however, until the busy node releases its control. Furthermore, as long as some link layer still needs to send signals, the appropriate clock indications must be generated and signals must be distributed.

In distribution mode, the bus delivers signals to all nodes except the one that dispatched it. To obtain a realistic model, the potential loss or corruption of signals is taken into account through a function that assigns an error value to the checksum field of header signals, data signals, and acknowledge signals.

Moreover, an extra dummy value will be used to describe the situation in which packets with a invalid length are delivered. The following transmission errors are modelled:

- If the signal is a destination signal, then this signal may be invalidated. However, if this happens, the header checksum (which comes with the next signal) is no longer valid. The bus should register of which nodes invalid destinations have been distributed.
- Any signal, except for header signals having a corrupted checksum according to the above, may be delivered correctly.
- If the signal to be delivered is a header signal, a data signal or an acknowledge signal, then it may be delivered corrupted, or it may not be delivered at all.
- If the signal to be delivered is a data signal, then the packet may be extended by sending a dummy signal immediately after the data signal.

When a signal has been distributed to every node, it is checked whether this signal was an `End` signal. If so, the current busy node no longer requires access to the bus. It is then checked whether some node has requested `immediate` arbitration. If not, a `subactgap` is distributed to all nodes and the bus returns to its idle state. Otherwise, if other nodes have requested access, control over the bus must go to one of those nodes. The bus then sends arbitration confirmations and a clock indication to all nodes that requested `immediate` arbitration.

It may happen that more than one node has control over the bus. To resolve such a conflict situation, the bus must wait for `End` signals from nodes, until only one node has access. Then, a data request is received from this node. If it is not an `End` signal, the node becomes the busy one and this signal is distributed to all other nodes. However, if the received signal is an `End` signal, no node has control over the bus anymore; a `subactgap` signal is then distributed to all nodes, after which the bus returns to its idle state.

2.4 Transaction and application layers

To precisely model the lower layers of IEEE 1394, it is sufficient to combine in parallel n LINK processes and one BUS process, which describes n PHY processes and a cable. The μCRL and mCRL2 models given in Annexes A and B follow this approach for $n = 2$, with a simple MAIN process gathering two link layers and a bus.

For model-checking verification (i.e., using a model checker to exhaustively explore and analyze the reachable state space), it is desirable to describe the upper layers as well, namely, the external behaviour of the transaction layer and of the application running on top of it. To this aim, M. Sighireanu introduced in her E-LOTOS model [37] two additional processes: TRANS, which represents a transaction layer, and `Application`, which describes the application and which we note APPLI.

TRANS process. As mentioned in Section 2.1, the transaction layer provides read, write, and lock transactions to the application. Transactions follow the traditional four-step connection establishment of the OSI model: request, indication, response, and confirmation. Inside the TRANS process, outgoing requests and incoming responses are handled by two sub-processes running in parallel and synchronized together. Both types of transactions (concatenated and split) are dealt with. Further details can be found in [37, Section 7].

The deadlock problem mentioned in Section 1 is caused by a missing transition in the packet transmit/receive state machine of the link layer (precisely, in the `Link4BRec` sub-process of the μCRL and mCRL2 models). To fix this bug, one option is to modify the behaviour of the link layer to insert the missing transition, as shown in [41]. Another option (adopted in the LOTOS and LNT models to preserve compatibility with the μCRL and mCRL2 models) is to keep the LINK process unchanged and modify instead the TRANS process by removing the transition (synchronized with the LINK process) that causes the deadlock; interestingly, the 2008 revision of IEEE 1394 also kept the link-layer state machine unchanged (see [45, Figure 6-21, Page 162]). Finally, to determine the behaviour of TRANS, a parameter `v` was added, which is equal either to `ok` (deadlock-free version) or to `ko` (original version).

APPLI process. M. Sighireanu designed 11 different applications, which differ by the scenario chosen among three possibilities (see [37, Section 9.2] for details), the maximal number of nodes connected to the bus, and the maximal number of requests sent to the link layer. Combined with both variants of the TRANS process, this led to 22 different MAIN processes, hence 22 models to be verified.

NODE process. To factorize the vast amount of duplicated code among these 22 MAIN processes, H. Garavel introduced a new NODE process that expresses the parallel composition of three processes: a LINK, a TRANS, and an APPLI. Notice that, unlike the approach of [37, Section 9.2], the APPLI process is no longer invoked from within the TRANS process.

3 Formal models

In this section, we present in more detail the four formal models of the IEEE 1394 link layer, following the chronological order of their development.

3.1 Formal model in μCRL

The first formal model of the link layer was written in 1997 by B. Luttik and circulated among the COST-247 community. It was reviewed by H. Garavel, J.F. Groote, and M. Sighireanu, who provided comments that led to improvements and simplifications. It was published as an annex (nicely compacted using mathematical symbols) in [26, 27] and, since then, has remained fairly stable. The μCRL model given in Annex A is close to this original model, with three enhancements:

- It is "machine-readable", meaning that it can be executed by the μCRL toolset.
- It uses the **map** keyword added in the 1997 version of μCRL [15] to declare non-constructors, whereas the original model [26, 27] used the 1995 version of μCRL [19], which does not distinguish between constructors and non-constructors.
- It introduces `tau` internal actions in the `Resolve` and `Distribute` sub-processes of the BUS process, in order to eliminate two unguarded recursive calls that existed in the original model and that the μCRL toolset cannot handle — even if the recursion is actually bounded by the fixed number of LINK processes.

Notice that the μCRL model is quite large (809 non-blank lines), as the `Bool` and `Nat` types with all their basic functions must be defined in extension. This verbosity issue was solved in the three other formal models.

3.2 Formal model in LOTOS

In 1997, M. Sighireanu wrote a LOTOS model of the IEEE 1394 link layer, based on the draft μCRL model of B. Luttik. The development of both models at the same time led to clarifications, enhancements, and simplifications in each of them. The LOTOS model aimed at using the existing CADP toolset [8] to perform model-checking verification, and became an official demo example [40] of CADP in 1997. The LOTOS code was similar in essence to the μCRL code, but with a few differences:

- As mentioned in Section 2.4, it introduced TRANS and APPLI processes to describe the upper layers of IEEE 1394, as well as various MAIN processes specifying 22 verification scenarios.
- The LOTOS model was shorter because it imported predefined libraries containing, e.g., the `Boolean` and `NaturalNumber` types.
- The LOTOS model uses conditional rewrite rules (e.g., $C_1,...,C_n \implies L = R$) where the μCRL model needs to take a detour via user-defined `if`(C,E,E') functions to express conditional equalities.
- The μCRL rewriter does not consider a fixed ordering of the rewrite rules: it is the modeller's responsibility to define a confluent term rewrite system. On the contrary, the CÆSAR.ADT compiler [9] for LOTOS assumes that the rewrite rules defining each (non-constructor) function are ordered by decreasing priority; this allows more concise definitions of equality functions (e.g., the `eq` comparator for type `SIGNAL` has 16 rules in μCRL and 2 in LOTOS) and other functions (e.g., `is_dest`, `is_header`, `is_data`, and `is_ack` need 10 rules each in μCRL and 2 in LOTOS).
- The LOTOS model renames all local variables `i` to `j`, because the former is a reserved LOTOS keyword that denotes the internal action (i.e., Milner's τ action). Later versions of CADP lifted this restriction by making it possible to have LOTOS variables or functions named `i`.

This LOTOS model remained stable for many years with only, in 2005, a simplification of the handwritten C code used to iterate over data domains, which was reduced from 2134 to 156 lines by factorizing similar code fragments present in the various scenarios.

However, in 2023, H. Garavel did a full revision of the LOTOS model, prompted by the development of the LNT model in parallel. The volume of LOTOS code was reduced by one third (from 2091 to 1385 lines), without loss of functionality and still preserving strong bisimilarity. This was done by merging the two versions of the TRANS process into one parameterized process, by merging the five versions of the APPLI process into another parameterized process, and by introducing the NODE process to factorize duplicated LOTOS code. A few other changes were made to simplify the LOTOS code and make it closer to the μCRL code:

- Like in the μCRL model, two LOTOS processes `Link` and `Bus` have been added to serve as main entry points.
- The definitions of the LOTOS type `SIGNAL` and of its related types have been aligned on the μCRL ones by eliminating unnecessary auxiliary tuple types. Yet, to make the LOTOS model easier to understand, the four overloaded constructors `sig` of type `SIGNAL` have been renamed to `destsig`, `acksig`, `datasig`, and `headersig`, respectively (even if LOTOS and LNT also support overloading of constructor functions).
- To reflect the model-checking assumptions of [37, Section 9.2], each of the three types `DATA`, `HEADER`, and `ACK` is directly defined as a singleton (one-value) type, rather than defining it as a two-value type and later providing ad hoc C code that only enumerates one of these two values.

3.3 Formal model in mCRL2

In June 2005, the μCRL model was translated to mCRL2 by J.F. Groote and distributed as a demo example [14] in the mCRL2 toolset.

The mCRL2 spec is 60% shorter than the μCRL one (809 non-empty lines in μCRL vs 327 in mCRL2). Most of this reduction comes from data type definitions, the size of which was roughly divided by 6.4 in mCRL2. This is explained by two factors:

- Like LOTOS, mCRL2 benefits from built-in data types (e.g., `Bool`, `Nat`, etc.), together with their basic functions, which need not be defined in every model.
- Like functional languages (ML, Haskell, etc.) and E-LOTOS [21], mCRL2 types can be defined by their constructors. For instance, the `SIGNAL` type is defined using the **struct** construct of mCRL2 and the `BoolTABLE` type is concisely defined using the built-in `List` datatype. For such types, equality functions, recognizers (i.e., functions, such as `is_dest`, that check whether an expression matches a given constructor), and projections (i.e., functions, such as `first`, `second`, `third`, and `fourth` for type `quadruple`, that extract the various arguments of a constructor) are defined automatically.

The mCRL2 processes differ on minor points from the μCRL ones:

- The syntax of the "**if** *C* **then** *A* **else** *B*" construct has changed: it is noted "*C* `->` *A* `<>` *B*" in mCRL2 and "*A* `<|` *C* `|>` *B*" in μCRL.
- In the LINK process, the μCRL definitions of the `Link0` and `Link7` sub-processes contain summations (i.e., nondeterministic choices) ranging over natural numbers that are not restricted in any way. In the mCRL2 model, these summations are bounded by the number of LINK layers.
- In the mCRL2 model, each `tau` action introduced to guard recursion (see Section 3.1) is replaced by an action `internal`, which is later abstracted from.

3.4 Formal model in LNT

Besides developing a complete LOTOS model and using it for model-checking verification, M. Sighireanu also wrote an E-LOTOS model of the IEEE 1394 link layer that was, rather than the LOTOS model itself, presented in [37, 38, 39]. At this time, the E-LOTOS language was still being standardized and not finalized yet. In essence, the E-LOTOS model bears similarities with the mCRL2 model developed later, notwithstanding the syntactic differences between both languages.

The LNT model presented in Annex D does not derive from this E-LOTOS model, as its history is distinct. In 2022, the LOTOS model (taken in its original version) was partly translated to LNT by Oussama Oulkaid and Marck-Edward Kemeh, as part of an exercise for master students at the University of Grenoble. Their model was later reworked and reshaped by H. Garavel, in order to make it complete and strongly bisimilar to the LOTOS one. Because it had been obtained by systematic translation, this LNT model was very much in the same style as the μCRL, mCRL2, and LOTOS ones: namely, data types defined as term rewrite systems, and processes defined as state machines extended with local variables that can be read and modified on transitions.

Therefore, H. Garavel entirely revised this LNT model in order to obtain a "better" model that would exploit the characteristic features of LNT and demonstrate the full capabilities of this language. This revision was achieved by progressive transformations, checking at each step that strong bisimilarity is preserved. Concerning data specifications in the resulting LNT model, three main remarks can be made:

- The type definitions in LNT are similar (up to syntax) to mCRL2 ones, except that equality/inequality functions must be requested explicitly (using "**with** =" and "**with** <>" clauses) and that functions for extracting/updating constructor arguments must also be requested (using "**with** `get`" and "**with** `set`" clauses); this ensures that LNT models are self-contained and not cluttered with useless implicit functions.

- As regards function definitions, the LOTOS rewrite rules ordered by decreasing priority can be systematically translated to LNT pattern-matching **case** statements. However, this is not the only style permitted by LNT, and not necessarily the most concise and readable one. One can also define functions in a more imperative style, with the usual programming constructs (variable assignments, **if**-**then**-**else**, **return** statements, etc.), as shown, for instance, in the various functions manipulating values of type `BoolTABLE`.

- A salient difference between μCRL, LOTOS, and mCRL2, on the one hand, and LNT, on the other hand, concerns partial functions, i.e., functions that are not defined over the entire domain of their arguments (e.g., function `get` for the `BoolTABLE` type or functions `getdest`, `getdcrc`, `getdata`, `gethead`, `getadd`, and `corrupt` for the `SIGNAL` type). In μCRL, LOTOS, and mCRL2, partial definition is implicit, in the sense that some equations are not given, e.g., there is no equation to define "`get (`n`, empty)`". The LOTOS model of Annex C contains comments to warn about partial definitions, but this is left to the good will of the specifier.

  In LNT, the situation is different: any partial function triggers (based on control- and data-flow analysis) an error, which the specifier is expected to correct, either by properly dealing with the overlooked cases, or by explicitly inserting a "**raise** E" statement at each point where the function might terminate without returning a result — E being either an event declared as an exception that the function can raise, or the predefined event `UNEXPECTED` denoting an exception that cannot be caught and triggers a run-time error.

Concerning processes, the following five transformations have been repeatedly applied until an idiomatic LNT model was obtained:

- The guarded commands "`[`C`]` $\rightarrow A$ `[] [not(`C`)]` $\rightarrow B$" present in the LOTOS model have been translated to "**if** C **then** A **else** B **end if**" statements of LNT. The **then** and **else** branches have been permuted, negating the Boolean condition C, when B was much shorter than A. Also, nested **if** statements have been flattened whenever possible by using the (Ada-like) **elsif** clause of LNT.

- When this was convenient, calls to recursive processes have been replaced by the **loop** statements of LNT, possibly with a **break** statement to exit the loop. For instance, the `Link3`, `Link5`, and `Link7` processes of the μCRL, mCRL2, and LOTOS models have been replaced, in the LNT model, by **loop** statements. Indeed, in μCRL, mCRL2, and LOTOS, (finite or infinite) iteration must always be expressed using recursion, with two main drawbacks: (i) the mandatory use of recursion obfuscates the flow of control by requiring the definition of auxiliary recursive processes and "goto-like" calls to these processes; (ii) it also obfuscates the flow of data by requiring, for such processes, as many parameters as there are live variables at the point where these processes are called. Using iteration rather than recursion often leads to simpler, more readable models.

- In some cases, finite loops can be further simplified by turning them into **while** or **for** loops. For instance, the sub-process `Resolve2` of the μCRL, mCRL2, and LOTOS models can be rephrased as a **while** loop, whereas the sub-processes `Resolve`, `SubactionGap`, and `Distribute` can be described using **for** loops, hereby getting rid of the extra parameters that store the loop variables.

Notice that such iterative behaviour was quite clear from the textual description of these processes in [26], but only LNT enables one to express it in natural way.

- Processes that are called only once (especially after recursion has been replaced by iteration) should be expanded in-line at the point where they are called. Doing so, the control flow becomes more readable (as each process call is similar to a "goto") and many process parameters are eliminated. M. Sighireanu applied this idea when designing her E-LOTOS model: the two μCRL sub-processes `DecideIdle` and `Link1` were expanded in-line [37, footnotes 7 and 8]. In the LNT model of Annex D, this idea was pushed beyond by also eliminating the sub-processes `Link3`, `Link3RA`, `Link3RE`, `Link4DH`, `Link4RH`, `Link4RD`, `Link4RE`, `Link4BRec`, `Link4DRec`, `Link5`, `Link6`, `Link7`, `Resolve`, and `Resolve2`. The sub-process `Link4`, although called only once, was not expanded in-line, because it is so large that its expansion would have increased the nesting depth too much. Also, a new `Link2` sub-process was added to factorize both sub-processes `Link2req` and `Link2resp` in a single one. As a result, the LINK process has only 6 (mutually recursive) processes in the LNT model, instead of 19 in the other models — maintaining an exact correspondence with the 8 states describing the asynchronous mode [43, Figure 6-19, Page 170] was not considered a requirement for the LNT model.

- Since the in-line expansion of processes often creates variables with nested scopes, three additional transformations may be suitable to keep the LNT model simple:
  - merging different variables that have the same type and are never used simultaneously, so as to decrease the number of variables.
  - enlarging the scope of nested variables by moving their declarations upward, so has to reduce the nesting depth of variable scopes;
  - renaming nested variables declared in the scope of another variable having the same name; for instance, after successively expanding the sub-process `Link7` in `Link6`, `Link6` in `Link5`, and `Link5` in `Link4DRec`, the `d` variable of `Link7` arrives in the scope of the `d` variable of `Link4DRec`; even if the innermost variable hides the outermost one in LNT (as in Algol-60), it may be suitable to give these variables different names to avoid confusion.

  These transformations sometimes conflict with each other, and their judicious application cannot be governed by strict laws: it is rather a matter of taste and circumstances.

4 Verification

The four formal models of the IEEE 1934 link layer have been checked by their respective compilers: the μCRL toolset, the mCRL2 toolset, and, for the LOTOS and LNT models, the CADP toolset.

The five correctness properties stated by B. Luttik [26, Section 4] have been formulated in the ACTL temporal logic [6] by R. Mateescu and M. Sighireanu [37, Section 10]. Using the XTL [29] model checker of CADP, these formulas have been checked on 16 out of 22 variants of the LOTOS model (totalling 80 model-checking jobs), the domains of the types `ACK`, `DATA`, and `HEADER` being limited to a single value. All the properties hold, except the first property (deadlock freeness), which is violated on the "original" models when the application layer executes its most complex scenarios.

The LNT model has been verified in two ways, using both model checking and equivalence checking. On the one hand, the ACTL formulas evaluate identically on the 16 variants of the LNT model. On the other hand, the labelled transition systems generated from 20 out of 22 variants of the LNT model are strongly bisimilar to those generated from the same variants of the LOTOS model. The labelled transition systems of the two remaining variants are too large for being generated directly, and would certainly benefit from compositional verification techniques [11]. In 14 cases out of 20, the labelled transition systems generated from LOTOS and LNT have the same size, whereas in 6 cases, those generated from LNT are slightly larger (+0.46% states, +0.43% transitions). Using version 2024-a "Eindhoven" of the CADP toolbox, these verifications were performed in less than 8 minutes on a Dell Latitude 5580 (Intel Core i5-7200U processor, 16 GB RAM) running Linux.

5 Conclusion

Revisiting the IEEE 1394 link layer problem, a true success story of formal methods, we presented and discussed four models written in μCRL, mCRL2, LOTOS, and LNT — the LOTOS model (revised in 2023) and the LNT model being novel contributions. In this respect, the present paper is a tentative "Rosetta stone" for comparing various modelling languages dedicated to communication protocols and concurrent systems. In a nutshell, our main findings are as follows:

- It appears that the three languages μCRL, mCRL2, and LOTOS are quite close, except that data type specifications are more concise in the latter two languages. Each of these three languages contains two separate sub-languages: one for specifying data types (using algebraic specifications or term rewrite rules), and another one for concurrent processes.

  These sub-languages sometimes use distinct symbols to express the same concept (e.g., **if-then-else** being noted differently in the data and process parts) and sometimes give the same symbol totally different meanings, e.g., in μCRL and mCRL2, the "$+$" operator (which denotes addition in the data part and nondeterministic choice in the process part), the "$||$" operator (which denotes logical disjunction in the data part and parallel composition in the process part), or closing parentheses (which denote the end of expressions in the data part and the end of a choice, a sequential composition, etc. in the process part).

  On the contrary, LNT is a unified language, without separate sub-languages: LNT functions and LNT processes are defined using the same notations (";" for sequential composition, **if-then-else** for conditionals, etc.), and LNT avoids, as much as possible, "overloaded" symbols.

- Although it has been argued that LOTOS supports very diverse "specification styles" [50], most LOTOS, μCRL, and mCRL2 models consist of a set of concurrent processes, each of which being specified using guarded commands and terminal recursion. Such a style is convenient for describing automata extended with state variables, but leads to models that are difficult to maintain when specifications evolve frequently, and does not scale well when automata complexity increases, resulting in large, poorly structured state machines scattered with "goto-like" transitions.

  In addition to supporting guarded commands and terminal recursion, LNT provides alternative specification styles suitable for the description of complex systems. In particular, LNT offers the classical primitives of structured programming, properly bracketed with an Ada-like syntax, which make large models easier to read and reduce the need for drawing state machines on paper.

To some extent, there is here a debate around the concept of minimality and how it should be interpreted. On the one hand, LOTOS, μCRL, and mCRL2 try to be minimal in the size of the language[2], the number of syntactic constructs, and the number of semantic rules. An explicit concern for μCRL and mCRL2 has been to ensure that the semantics are as simple and elegant as possible, only including constructs in

[2]The μ letter (which stands for "micro") in μCRL indeed expresses such a desire for minimality.

the language if they are needed for expressiveness; ease of modelling has been less of a concern so far. LNT also tries to be minimal, e.g., by unifying the sub-languages for functions and processes, the former being included in the latter, but it can be rightly argued that LNT is richer than the three other languages and requires more complex compilers that implement involved control- and data-flow analyses.

Perhaps the proper concept of minimality is not so much about the size of a language or of its compiler, but about the effort needed to learn the language, the time needed to write correct models, and the difficulty of understanding such models for engineers who do not have a strong background in formal methods. We hope that the present study will usefully contribute to this debate.

Acknowledgements

We are grateful to all those who contributed to the design and verification of the four formal models of the IEEE 1394 link layer, namely: Jan Friso Groote, Marck-Edward Kemeh, Radu Mateescu, Laurent Mounier, Oussama Oulkaid, Charles Pecheur, Judi Romijn, Mihaela Sighireanu, and Bruno Vivien. We also thank the anonymous reviewers for their constructive remarks.

References

[1] Olav Bunte, Jan Friso Groote, Jeroen J. A. Keiren, Maurice Laveaux, Thomas Neele, Erik P. de Vink, Wieger Wesselink, Anton Wijs & Tim A. C. Willemse (2019): *The mCRL2 Toolset for Analysing Concurrent Systems – Improvements in Expressivity and Usability*. In Tomás Vojnar & Lijun Zhang, editors: *Proceedings (Part II) of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019), Prague, Czech Republic Proceedings*, *Lecture Notes in Computer Science* 11428, Springer, pp. 21–39, doi:10.1007/978-3-030-17465-1_2.

[2] Vincenza Carchiolo, Michele Malgeri & Giuseppe Mangioni (2003): *Synthesis of LOTOS Specification of the IEEE-1394 Firewire Protocol*. In: *Proceedings of the 14th IEEE International Workshop on Rapid System Prototyping (RSP'03), San Diego, California, USA*, IEEE Computer Society Press, pp. 86–92, doi:10.1109/IWRSP.2003.1207034.

[3] David Champelovier, Xavier Clerc, Hubert Garavel, Yves Guerte, Christine McKinty, Vincent Powazny, Frédéric Lang, Wendelin Serwe & Gideon Smeding (2023): *Reference Manual of the LNT to LOTOS Translator (Version 7.1)*. Available at `http://cadp.inria.fr/publications/Champelovier-Clerc-Garavel-et-al-10.html`. INRIA, Grenoble, France.

[4] Conrado Daws, Marta Z. Kwiatkowska & Gethin Norman (2002): *Automatic Verification of the IEEE-1394 Root Contention Protocol with KRONOS and PRISM*. In Rance Cleaveland & Hubert Garavel, editors: *Proceedings of the 7th International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'02), Málaga, Spain*, *Electronic Notes in Theoretical Computer Science* 66, Elsevier, pp. 104–119, doi:10.1016/S1571-0661(04)80406-7.

[5] Conrado Daws, Marta Z. Kwiatkowska & Gethin Norman (2004): *Automatic Verification of the IEEE 1394 Root Contention Protocol with KRONOS and PRISM*. *International Journal on Software Tools for Technology Transfer (STTT)* 5(2–3), pp. 221–236, doi:10.1007/S10009-003-0118-5.

[6] Rocco De Nicola & Frits W. Vaandrager (1990): *Action versus State based Logics for Transition Systems*. In Irène Guessarian, editor: *Semantics of Systems of Concurrent Processes – Proceedings of the LITP Spring School on Theoretical Computer Science, La Roche Posay, France*, *Lecture Notes in Computer Science* 469, Springer, pp. 407–419, doi:10.1007/3-540-53479-2_17.

[7] Marco Devillers, W. O. David Griffioen, Judi Romijn & Frits W. Vaandrager (2000): *Verification of a Leader Election Protocol: Formal Methods Applied to IEEE 1394*. *Formal Methods in System Design* 16(3), pp. 307–320, doi:10.1023/A:1008764923992.

[8] Jean-Claude Fernandez, Hubert Garavel, Alain Kerbrat, Radu Mateescu, Laurent Mounier & Mihaela Sighireanu (1996): *CADP (CÆSAR/ALDEBARAN Development Package): A Protocol Validation and Verification Toolbox*. In Rajeev Alur & Thomas A. Henzinger, editors: *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96), New Brunswick, New Jersey, USA*, *Lecture Notes in Computer Science* 1102, Springer, pp. 437–440, doi:10.1007/3-540-61474-5_97.

[9] Hubert Garavel (1989): *Compilation of LOTOS Abstract Data Types*. In Son T. Vuong, editor: *Proceedings of the 2nd International Conference on Formal Description Techniques FORTE'89 (Vancouver B.C., Canada)*, North Holland, pp. 147–162. Available at `http://cadp.inria.fr/publications/Garavel-89-c.html`.

[10] Hubert Garavel (2008): *Reflections on the Future of Concurrency Theory in General and Process Calculi in Particular*. In Catuscia Palamidessi & Frank D. Valencia, editors: *Proceedings of the LIX Colloquium on Emerging Trends in Concurrency Theory, Ecole Polytechnique de Paris, France, November 13–15, 2006*, *Electronic Notes in Theoretical Computer Science* 209, Elsevier Science Publishers, pp. 149–164, doi:10.1016/J.ENTCS.2008.04.009. Also available as INRIA Research Report RR-6368.

[11] Hubert Garavel, Frédéric Lang & Laurent Mounier (2018): *Compositional Verification in Action*. In Falk Howar & Jiri Barnat, editors: *Proceedings of the 23rd International Conference on Formal Methods for Industrial Critical Systems (FMICS'18), Maynooth, Ireland – Essays Dedicated to Susanne Graf at the Occasion of Her 60th Birthday*, *Lecture Notes in Computer Science* 11119, Springer, pp. 189–210, doi:10.1007/978-3-030-00244-2_13.

[12] Hubert Garavel, Frédéric Lang & Wendelin Serwe (2017): *From LOTOS to LNT*. In Joost-Pieter Katoen, Rom Langerak & Arend Rensink, editors: *ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10500, Springer, pp. 3–26, doi:10.1007/978-3-319-68270-9_1.

[13] Hubert Garavel & Mihaela Sighireanu (1998): *Towards a Second Generation of Formal Description Techniques – Rationale for the Design of E-LOTOS*. In Jan-Friso Groote, Bas Luttik & Jos van Wamel, editors: *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems (FMICS'98), Amsterdam, The Netherlands*, CWI, Amsterdam, pp. 187–230. Available at `http://cadp.inria.fr/publications/Garavel-Sighireanu-98-a.html`.

[14] Jan Friso Groote: *IEEE 1394 Link Layer in mCRL2*. Available at `https://github.com/mCRL2org/mCRL2/tree/master/examples/industrial/1394`.

[15] Jan Friso Groote (1997): *The Syntax and Semantics of Timed μCRL*. Technical Report SEN-R9709, CWI, Amsterdam, The Netherlands. Available at `https://ir.cwi.nl/pub/4746`.

[16] Jan Friso Groote, Aad Mathijssen, Michel Reniers, Yaroslav Usenko & Muck van Weerdenburg (2007): *The Formal Specification Language mCRL2*. In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens & Roel Wieringa, editors: *Methods for Modelling Software Systems (MMOSS)*, *Dagstuhl Seminar Proceedings* 06351, Schloss Dagstuhl, Germany, pp. 1–34, doi:10.4230/DagSemProc.06351.12.

[17] Jan Friso Groote, Aad Mathijssen, Muck van Weerdenburg & Yaroslav S. Usenko (2006): *From μCRL to mCRL2: Motivation and Outline*. *Electronic Notes in Theoretical Computer Science* 162, pp. 191–196, doi:10.1016/j.entcs.2005.12.101.

[18] Jan Friso Groote & Mohammad Reza Mousavi (2014): *Modeling and Analysis of Communicating Systems*. The MIT Press, doi:10.7551/mitpress/9946.001.0001.

[19] Jan Friso Groote & Alban Ponse (1995): *The Syntax and Semantics of μCRL*. In A. Ponse, C. Verhoef & S.F.M. van Vlijmen, editors: *Proceedings of the 1st Workshop on the Algebra of Communicating Processes (ACP'94), Utrecht, The Netherlands*, Workshops in Computing Series, Springer, pp. 26–62, doi:10.1007/978-1-4471-2120-6_2.

[20] ISO/IEC (1989): *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. Available at `https://www.iso.org/standard/16258.html`.

[21] ISO/IEC (2001): *Enhancements to LOTOS (E-LOTOS)*. International Standard 15437:2001, International Organization for Standardization – Information Technology, Geneva. Available at `https://www.iso.org/standard/27680.html`.

[22] Mark Jorgensen & Hubert Garavel (1997): *Final Report of the COST-247 Action*. Available at `https://vasy.inria.fr/COST247`.

[23] Lars Kühne, Jozef Hooman & Willem-Paul de Roever (1997): *Towards Mechanical Verification of Parts of the IEEE P1394 Serial Bus*. In Ignac Lovrek, editor: Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), pp. 73–85.

[24] Marta Z. Kwiatkowska, Gethin Norman & Jeremy Sproston (2003): *Probabilistic Model Checking of Deadline Properties in the IEEE 1394 FireWire Root Contention Protocol*. Formal Aspects of Computing 14(3), pp. 295–318, doi:10.1007/S001650300007.

[25] Izak van Langevelde, Judi Romijn & Nicolae Goga (2003): *Founding FireWire Bridges through Promela Prototyping*. In: Proceedings of the 17th International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice, France, IEEE Computer Society, p. 239, doi:10.1109/IPDPS.2003.1213434.

[26] Bas Luttik (1997): *Description and Formal Specification of the Link Layer of P1394*. Report SEN-R9706, CWI, Software Engineering (SEN), Amsterdam, The Netherlands. Available at `https://ir.cwi.nl/pub/4758`.

[27] Bas Luttik (1997): *Description and Formal Specification of the Link Layer of P1394*. In Ignac Lovrek, editor: Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), pp. 43–56.

[28] Savi Maharaj & Carron Shankland (2000): *A Survey of Formal Methods Applied to Leader Election in IEEE 1394*. Journal of Universal Computer Science 6(11), pp. 1145–1163. Available at `http://www.jucs.org/jucs_6_11/a_survey_of_formal`.

[29] Radu Mateescu & Hubert Garavel (1998): *XTL: A Meta-Language and Tool for Temporal Logic Model-Checking*. In Tiziana Margaria, editor: Proceedings of the International Workshop on Software Tools for Technology Transfer (STTT'98), Aalborg, Denmark, BRICS, pp. 33–42. Available at `http://cadp.inria.fr/publications/Mateescu-Garavel-98.html`.

[30] Judi Romijn (1999): *A Timed Verification of the IEEE 1394 Leader Election Protocol*. In Stefania Gnesi & Diego Latella, editors: Proceedings of the 4th International Workshop on Formal Methods for Industrial Critical Systems (FMICS'99), Trento, Italy, pp. 3–29.

[31] Judi Romijn (2001): *A Timed Verification of the IEEE 1394 Leader Election Protocol*. Formal Methods in System Design 19(2), pp. 165–194, doi:10.1023/A:1011284000753.

[32] Judi Romijn (2003): *False Loop Detection in the IEEE 1394 Tree Identify Phase*. Formal Aspects of Computing 14(3), pp. 319–327, doi:10.1007/S001650300008.

[33] Carron Shankland & Alberto Verdejo (1999): *Time, E-LOTOS, and the FireWire*. In Marco Ajmone Marsan, Juan Quemada, Tomás Robles & Manuel Silva, editors: Proceedings of the Workshop on Formal Methods and Telecommunications (WFMT'99), Zaragoza, Spain, Prensas Universitarias de Zaragoza, pp. 103–119. Available at `http://maude.sip.ucm.es/alberto-verdejo/papers/FireWire99.html`.

[34] Carron Shankland & Alberto Verdejo (2001): *A Case Study in Abstraction Using E-LOTOS and the FireWire*. Computer Networks 37(3/4), pp. 481–502, doi:10.1016/S1389-1286(01)00190-6.

[35] Carron Shankland & Mark van der Zwaag (1998): *The Tree Identify Protocol of IEEE 1394 in μCRL*. Formal Aspects of Computing 10(5-6), pp. 509–531, doi:10.1007/s001650050030.

[36] Mihaela Sighireanu, Alban Catry, David Champelovier, Hubert Garavel, Frédéric Lang, Guillaume Schaeffer, Wendelin Serwe & Jan Stoecker (2023): *LOTOS NT User's Manual (Version 3.12)*. INRIA/CONVECS, Grenoble, France, `https://vasy.inria.fr/ftp/traian/manual.pdf`, 88 pages.

[37] Mihaela Sighireanu & Radu Mateescu (1997): *Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS*. Research Report RR-3172, INRIA, France. Available at `http://cadp.inria.fr/publications/Sighireanu-Mateescu-97.html`.

[38] Mihaela Sighireanu & Radu Mateescu (1997): *Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus ("FireWire"): an Experiment with E-LOTOS*. In Ignac Lovrek, editor: *Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia)*, pp. 57–72. Full version available as INRIA Research Report RR-3172.

[39] Mihaela Sighireanu & Radu Mateescu (1998): *Verification of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): an Experiment with E-LOTOS*. *Springer International Journal on Software Tools for Technology Transfer (STTT)* 2(1), pp. 68–88, doi:10.1007/S100090050018.

[40] Mihaela Sighireanu, Radu Mateescu & Hubert Garavel: *CADP Demo № 23*. Available at `http://cadp.inria.fr/ftp/demos/demo_23`.

[41] Mihaela Sighireanu, Radu Mateescu & Hubert Garavel (1998): *VASY Reports a Deadlock in the IEEE 1394 "Firewire" Standard*. Available at `https://vasy.inria.fr/press/firewire.html`.

[42] David P. L. Simons & Mariëlle Stoelinga (2001): *Mechanical Verification of the IEEE 1394a Root Contention Protocol Using Uppaal2k*. *International Journal on Software Tools for Technology Transfer (STTT)* 3(4), pp. 469–485, doi:10.1007/S100090100059.

[43] IEEE Computer Society (1995): *IEEE Standard for a High Performance Serial Bus*. IEEE Standard 1394-1995, Institution of Electrical and Electronic Engineers, doi:10.1109/IEEESTD.1996.81049.

[44] IEEE Computer Society (1995): *P1394 Standard for a High Performance Serial Bus*. Technical Report, Institution of Electrical and Electronic Engineers. Draft 8.0v2.

[45] IEEE Computer Society (2008): *IEEE Standard for a High Performance Serial Bus*. IEEE Standard 1394-2008, Institution of Electrical and Electronic Engineers, doi:10.1109/IEEESTD.2008.4659233.

[46] Mariëlle Stoelinga (2003): *Fun with FireWire: A Comparative Study of Formal Verification Methods Applied to the IEEE 1394 Root Contention Protocol*. *Formal Aspects of Computing* 14(3), pp. 328–337, doi:10.1007/S001650300009.

[47] Mariëlle Stoelinga & Frits W. Vaandrager (1999): *Root Contention in IEEE 1394*. In Joost-Pieter Katoen, editor: *Proceedings of the 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems (ARTS'99), Bamberg, Germany*, *Lecture Notes in Computer Science* 1601, Springer, pp. 53–74, doi:10.1007/3-540-48778-6_4.

[48] Alberto Verdejo, Isabel Pita & Narciso Martí-Oliet (2000): *The Leader Election Protocol of IEEE 1394 in Maude*. In Kokichi Futatsugi, editor: *Proceedings of the 3rd International Workshop on Rewriting Logic and its Applications (WRLA 2000), Kanzawa, Japan*, *Electronic Notes in Theoretical Computer Science* 36, Elsevier, pp. 383–404, doi:10.1016/S1571-0661(05)80133-1.

[49] Alberto Verdejo, Isabel Pita & Narciso Martí-Oliet (2003): *Specification and Verification of the Tree Identify Protocol of IEEE 1394 in Rewriting Logic*. *Formal Aspects of Computing* 14(3), pp. 228–246, doi:10.1007/S001650300003.

[50] C. Vissers, G. Scollo, M. van Sinderen & E. Brinksma (1991): *Specification Styles in Distributed Systems Design and Verification*. *Theoretical Computer Science* 89(1), pp. 179–206, doi:10.1016/0304-3975(90)90111-T.

A Formal model in μCRL

A.1 Types and functions in μCRL

% Boolean type

sort `Bool`
func

```
  T,F: -> Bool
```

map
```
  eq: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  eq(T,b)=b
  eq(b,T)=b
  eq(b,F)=not(b)
  eq(F,b)=not(b)
```

map
```
  and: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  and(T,b)=b
  and(b,T)=b
  and(b,F)=F
  and(F,b)=F
```

map
```
  or: Bool#Bool -> Bool
```
var
```
  b: Bool
```
rew
```
  or(T,b)=T
  or(b,T)=T
  or(b,F)=b
  or(F,b)=b
```

map
```
  not: Bool -> Bool
  if: Bool#Bool#Bool -> Bool
```
var
```
  b1,b2: Bool
```
rew
```
  not(F)=T
  not(T)=F
  if(T,b1,b2)=b1
  if(F,b1,b2)=b2
```

% Natural number type

sort `NAT`
func
```
  0,1,2: -> NAT
```
% 0,1,2,3,4,5,6,7,8,9: −> NAT
map `succ: NAT -> NAT`

map

```
  eq: NAT#NAT -> Bool
var
  n,m: NAT
rew
  1=succ(0)
  2=succ(1)
  eq(0,0)=T
  eq(succ(n),0)=F
  eq(0,succ(n))=F
  eq(succ(n),succ(m))=eq(n,m)

map
  lt: NAT#NAT -> Bool
var
  n,m: NAT
rew
  lt(0,0)=F
  lt(succ(n),0)=F
  lt(0,succ(n))=T
  lt(succ(n),succ(m))=lt(n,m)

% Data/Control/Acknowledge elemens and their CRC computation

sort CHECK
func
  bottom,check: -> CHECK
map
  eq: CHECK#CHECK -> Bool
rew
  eq(bottom,bottom)=T
  eq(check,check)=T
  eq(check,bottom)=F
  eq(bottom,check)=F

sort DATA
func
  d1,d2: -> DATA
map
  crc: DATA -> CHECK
  eq: DATA#DATA -> Bool
rew
  crc(d1)=check
  crc(d2)=check
  eq(d1,d1)=T
  eq(d1,d2)=F
  eq(d2,d1)=F
  eq(d2,d2)=T

sort HEADER
func
  h1,h2: -> HEADER
map
```

```
  crc: HEADER -> CHECK
  eq: HEADER # HEADER -> Bool
rew
  crc(h1)=check
  crc(h2)=check
  eq(h1,h1)=T
  eq(h1,h2)=F
  eq(h2,h1)=F
  eq(h2,h2)=T

sort ACK
func
  a1,a2: -> ACK
map
  crc: ACK -> CHECK
  eq : ACK # ACK -> Bool
rew
  crc(a1)=check
  crc(a2)=check
  eq(a1,a1)=T
  eq(a1,a2)=F
  eq(a2,a1)=F
  eq(a2,a2)=T

sort SIGNAL
func
  sig: NAT -> SIGNAL
  sig: HEADER#CHECK -> SIGNAL
  sig: DATA#CHECK -> SIGNAL
  sig: ACK#CHECK -> SIGNAL

  Start,End: -> SIGNAL
  Prefix,subactgap: -> SIGNAL
  dhead,Dummy: -> SIGNAL
map
  is_start,is_end: SIGNAL -> Bool
  is_prefix,is_sagap: SIGNAL -> Bool
  is_dummy,is_dhead: SIGNAL -> Bool
  eq: SIGNAL#SIGNAL -> Bool
var
  n,n' : NAT
  h,h' : HEADER
  d,d' : DATA
  a,a' : ACK
  c,c' : CHECK
  s : SIGNAL
rew
  is_start(Start)=T
  is_start(End)=F
  is_start(Prefix)=F
  is_start(subactgap)=F
  is_start(dhead)=F
  is_start(Dummy)=F
  is_start(sig(n))=F
  is_start(sig(h,c))=F
  is_start(sig(d,c))=F
  is_start(sig(a,c))=F
  eq(Start,s)=is_start(s)
  eq(s,Start)=is_start(s)

  is_end(End)=T
  is_end(Start)=F
  is_end(Prefix)=F
  is_end(subactgap)=F
  is_end(dhead)=F
  is_end(Dummy)=F
  is_end(sig(n))=F
  is_end(sig(h,c))=F
  is_end(sig(d,c))=F
  is_end(sig(a,c))=F
  eq(End,s)=is_end(s)
  eq(s,End)=is_end(s)

  is_prefix(Prefix)=T
  is_prefix(Start)=F
  is_prefix(End)=F
  is_prefix(subactgap)=F
  is_prefix(dhead)=F
  is_prefix(Dummy)=F
  is_prefix(sig(n))=F
  is_prefix(sig(h,c))=F
  is_prefix(sig(d,c))=F
  is_prefix(sig(a,c))=F
  eq(Prefix,s)=is_prefix(s)
  eq(s,Prefix)=is_prefix(s)

  is_sagap(subactgap)=T
  is_sagap(Start)=F
  is_sagap(End)=F
  is_sagap(Prefix)=F
  is_sagap(dhead)=F
  is_sagap(Dummy)=F
  is_sagap(sig(n))=F
  is_sagap(sig(h,c))=F
  is_sagap(sig(d,c))=F
  is_sagap(sig(a,c))=F
  eq(subactgap,s)=is_sagap(s)
  eq(s,subactgap)=is_sagap(s)

  is_dhead(subactgap)=F
  is_dhead(Start)=F
  is_dhead(End)=F
  is_dhead(Prefix)=F
  is_dhead(dhead)=T
```

```
  is_dhead(Dummy)=F
  is_dhead(sig(n))=F
  is_dhead(sig(h,c))=F
  is_dhead(sig(d,c))=F
  is_dhead(sig(a,c))=F
  eq(dhead,s)=is_dhead(s)
  eq(s,dhead)=is_dhead(s)

  is_dummy(subactgap)=F
  is_dummy(Start)=F
  is_dummy(End)=F
  is_dummy(Prefix)=F
  is_dummy(dhead)=F
  is_dummy(Dummy)=T
  is_dummy(sig(n))=F
  is_dummy(sig(h,c))=F
  is_dummy(sig(d,c))=F
  is_dummy(sig(a,c))=F
  eq(Dummy,s)=is_dummy(s)
  eq(s,Dummy)=is_dummy(s)

  eq(sig(n),sig(n'))=eq(n,n')
  eq(sig(n),sig(h,c))=F
  eq(sig(n),sig(d,c))=F
  eq(sig(n),sig(a,c))=F
  eq(sig(h,c),sig(n'))=F
  eq(sig(h,c),sig(h',c'))=and(eq(h,h'),eq(c,c'))
  eq(sig(h,c),sig(d,c'))=F
  eq(sig(h,c),sig(a,c'))=F
  eq(sig(d,c),sig(n))=F
  eq(sig(d,c),sig(h,c'))=F
  eq(sig(d,c),sig(d',c'))=and(eq(d,d'),eq(c,c'))
  eq(sig(d,c),sig(a,c'))=F
  eq(sig(a,c),sig(n))=F
  eq(sig(a,c),sig(h,c'))=F
  eq(sig(a,c),sig(d,c'))=F
  eq(sig(a,c),sig(a',c'))=and(eq(a,a'),eq(c,c'))
```

map
```
  is_dest,is_header: SIGNAL -> Bool
  is_data,is_ack: SIGNAL -> Bool
```
var
```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```
rew
```
  is_dest(sig(n))=T
  is_dest(sig(h,c))=F
  is_dest(sig(d,c))=F
  is_dest(sig(a,c))=F
  is_dest(Start)=F
  is_dest(End)=F
  is_dest(Prefix)=F
  is_dest(subactgap)=F
  is_dest(dhead)=F
  is_dest(Dummy)=F

  is_header(sig(h,c))=T
  is_header(sig(n))=F
  is_header(sig(d,c))=F
  is_header(sig(a,c))=F
  is_header(Start)=F
  is_header(End)=F
  is_header(Prefix)=F
  is_header(subactgap)=F
  is_header(dhead)=F
  is_header(Dummy)=F

  is_data(sig(d,c))=T
  is_data(sig(n))=F
  is_data(sig(h,c))=F
  is_data(sig(a,c))=F
  is_data(Start)=F
  is_data(End)=F
  is_data(Prefix)=F
  is_data(subactgap)=F
  is_data(dhead)=F
  is_data(Dummy)=F

  is_ack(sig(a,c))=T
  is_ack(sig(n))=F
  is_ack(sig(h,c))=F
  is_ack(sig(d,c))=F
  is_ack(Start)=F
  is_ack(End)=F
  is_ack(Prefix)=F
  is_ack(subactgap)=F
  is_ack(dhead)=F
  is_ack(Dummy)=F
```

map
```
  is_physig,is_terminator: SIGNAL -> Bool
```
var
```
  s : SIGNAL
```
rew
```
  is_physig(s)=or(is_start(s),or(is_end(s),or(is_prefix(s),is_sagap(s))))
  is_terminator(s)=or(is_end(s),is_prefix(s))
```

map
```
  is_hda: SIGNAL -> Bool
```
var
```
  s : SIGNAL
```

rew
```
  is_hda(s)=or(is_header(s),or(is_data(s),is_ack(s)))
```

map
```
  valid_hpart, valid_ack: SIGNAL -> Bool
```
var
```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```
rew
```
  valid_ack(sig(a,c))=eq(c,check)
  valid_ack(sig(h,c))=F
  valid_ack(sig(d,c))=F
  valid_ack(sig(n))=F
  valid_ack(Start)=F
  valid_ack(End)=F
  valid_ack(Prefix)=F
  valid_ack(subactgap)=F
  valid_ack(Dummy)=F
  valid_ack(dhead)=F

  valid_hpart(sig(h,c))=eq(c,check)
  valid_hpart(sig(n))=F
  valid_hpart(sig(d,c))=F
  valid_hpart(sig(a,c))=F
  valid_hpart(Start)=F
  valid_hpart(End)=F
  valid_hpart(Prefix)=F
  valid_hpart(subactgap)=F
  valid_hpart(Dummy)=F
  valid_hpart(dhead)=F
```

map
```
  getdest: SIGNAL -> NAT
  getdcrc: SIGNAL -> CHECK
  getdata: SIGNAL -> DATA
  gethead: SIGNAL -> HEADER
  getack: SIGNAL -> ACK
  corrupt: SIGNAL -> SIGNAL
```
var
```
  n : NAT
  h : HEADER
  d : DATA
  a : ACK
  c : CHECK
```
rew
```
  getdest(sig(n)) = n
  gethead(sig(h,c)) = h
  getdcrc(sig(d,c)) = c
  getdata(sig(d,c)) = d
  getack (sig(a,c)) = a

  corrupt(sig(h,c)) = sig(h,bottom)
  corrupt(sig(d,c)) = sig(d,bottom)
  corrupt(sig(a,c)) = sig(a,bottom)

sort SIG_TUPLE
func
  quadruple: SIGNAL#SIGNAL#SIGNAL#SIGNAL -> SIG_TUPLE
  void: -> SIG_TUPLE
map
  first,second,third,fourth: SIG_TUPLE -> SIGNAL
  is_void: SIG_TUPLE -> Bool
var
  x1,x2,x3,x4: SIGNAL
rew
  first(quadruple(x1,x2,x3,x4))=x1
  second(quadruple(x1,x2,x3,x4))=x2
  third(quadruple(x1,x2,x3,x4))=x3
  fourth(quadruple(x1,x2,x3,x4))=x4

  is_void(void)=T
  is_void(quadruple(x1,x2,x3,x4))=F

sort PAR
func
  fair,immediate: -> PAR
map
  eq: PAR#PAR -> Bool
rew
  eq(fair,fair)=T
  eq(immediate,immediate)=T
  eq(fair,immediate)=F
  eq(immediate,fair)=F

sort PAC
func
  won,lost: -> PAC
map
  eq: PAC#PAC -> Bool
rew
  eq(won,won)=T
  eq(lost,lost)=T
  eq(won,lost)=F
  eq(lost,won)=F

sort LDC
func
  ackrec: ACK -> LDC
  ackmiss,broadsent: -> LDC

sort LDI
```

```
func
  good,broadrec: HEADER#DATA -> LDI
  dcrc_err: HEADER -> LDI

sort BOC
func
  release,hold: -> BOC
map
  eq: BOC#BOC -> Bool
rew
  eq(release,release)=T
  eq(hold,hold)=T
  eq(release,hold)=F
  eq(hold,release)=F
```

A.2 The LINK process in μCRL

```
act
  LDreq: NAT#NAT#HEADER#DATA
  LDcon: NAT#LDC
  LDind: NAT#LDI
  LDres: NAT#ACK#BOC

  sPDreq,rPDind: NAT#SIGNAL
  sPAreq: NAT#PAR
  rPAcon: NAT#PAC
  rPCind: NAT

proc

LINK(n:NAT,i:NAT)=
   ( Link0(n,i,void) )

Link0(n:NAT,id:NAT,buffer:SIG_TUPLE)=
(
    sum(dest:NAT,
      sum(h:HEADER,
        sum(d:DATA,
          LDreq(id,dest,h,d).
            Link0(n,id,quadruple(dhead,
                                 sig(dest),
                                 sig(h,crc(h)),
                                 sig(d,crc(d))))
        )
      )
    )
  <| is_void(buffer) |>
    sPAreq(id,fair).Link1(n,id,buffer)
)
+
  sum(p:SIGNAL,
    rPDind(id,p).
      ( Link4(n,id,buffer) <| is_start(p) |> Link0(n,id,buffer) )
  )

Link1(n:NAT,id:NAT,p:SIG_TUPLE)=
  rPAcon(id,won).Link2req(n,id,p)
+
  rPAcon(id,lost).Link0(n,id,p)

Link2req(n:NAT,id:NAT,p:SIG_TUPLE)=
   ( rPCind(id).sPDreq(id,Start).
     rPCind(id).sPDreq(id,first(p)).
     rPCind(id).sPDreq(id,second(p)) ) .
   ( rPCind(id).sPDreq(id,third(p)).
     rPCind(id).sPDreq(id,fourth(p)).
     rPCind(id).sPDreq(id,End) ).
    (
      LDcon(id,broadsent).Link0(n,id,void)
     <| eq(getdest(second(p)),n) |>
      Link3(n,id,void)
    )

Link3(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(p:SIGNAL,
    rPDind(id,p).
      (
        Link3(n,id,buffer)
      <| is_prefix(p) |>
        (
          Link3RA(n,id,buffer)
        <| is_start(p) |>
          (
            LDcon(id,ackmiss).Link0(n,id,buffer)
          <| is_sagap(p) |>
            LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
          )
        )
      )
  )

Link3RA(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(a:SIGNAL,
    rPDind(id,a).
      (
        (
          LDcon(id,ackmiss).Link0(n,id,buffer)
        <| is_sagap(a) |>
          LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
        )
      <| is_physig(a) |>
        Link3RE(n,id,buffer,a)
      )
  )

Link3RE(n:NAT,id:NAT,buffer:SIG_TUPLE,a:SIGNAL)=
  sum(e:SIGNAL,
    rPDind(id,e).
      (
        LDcon(id,ackrec(getack(a))).LinkWSA(n,id,buffer,n)
      <| and(valid_ack(a),is_terminator(e)) |>
        (
          LDcon(id,ackmiss).Link0(n,id,buffer)
        <| is_sagap(e) |>
          LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
        )
      )
  )

Link4(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(dh:SIGNAL,
    rPDind(id,dh).
      (
        (
          Link0(n,id,buffer)
        <| is_sagap(dh) |>
          LinkWSA(n,id,buffer,n)
        )
      <| is_physig(dh) |>
        Link4DH(n,id,buffer)
      )
  )

Link4DH(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(dest:SIGNAL,
    rPDind(id,dest).
      (
        (
          sPAreq(id,immediate).Link4RH(n,id,buffer,id)
        <| eq(getdest(dest),id) |>
          (
            Link4RH(n,id,buffer,n)
          <| eq(getdest(dest),n) |>
            LinkWSA(n,id,buffer,n)
          )
        )
      <| is_dest(dest) |>
        (
          Link0(n,id,buffer)
        <| is_sagap(dest) |>
          LinkWSA(n,id,buffer,n)
        )
      )
  )
```

```
Link4RH(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT)=
  sum(h:SIGNAL,
    rPDind(id,h).
      (
        Link4RD(n,id,buffer,dest,h)
      <| valid_hpart(h) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4RD(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT,h:SIGNAL)=
  sum(d:SIGNAL,
    rPDind(id,d).
      (
        Link4RE(n,id,buffer,dest,h,d)
      <| is_data(d) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4RE(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT,h:SIGNAL,d:SIGNAL)=
  sum(e:SIGNAL,
    rPDind(id,e).
      (
        (
          Link4DRec(n,id,buffer,h,d)
        <| eq(dest,id) |>
          Link4BRec(n,id,buffer,h,d)
        )
      <| is_terminator(e) |>
        LinkWSA(n,id,buffer,dest)
      )
  )

Link4DRec(n:NAT,id:NAT,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
  LDind(id,good(gethead(h),getdata(d))).rPAcon(id,won).Link5(n,id,buffer)
<| eq(getdcrc(d),check) |>
  LDind(id,dcrc_err(gethead(h))).rPAcon(id,won).Link5(n,id,buffer)

Link4BRec(n:NAT,id:NAT,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
  LDind(id,broadrec(gethead(h),getdata(d))).Link0(n,id,buffer)
<| eq(getdcrc(d),check) |>
  Link0(n,id,buffer)

Link5(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  sum(a:ACK,
    sum(b:BOC,
      LDres(id,a,b).Link6(n,id,buffer,sig(a,crc(a)),b)
    )
  )
+
  rPCind(id).sPDreq(id,Prefix).Link5(n,id,buffer)
```

```
Link6(n:NAT,id:NAT,buffer:SIG_TUPLE,p:SIGNAL,b:BOC)=
  ( rPCind(id).sPDreq(id,Start).rPCind(id).sPDreq(id,p) ) .
      ( rPCind(id).
        (
          sPDreq(id,End).Link0(n,id,buffer)
        <| eq(b,release) |>
          sPDreq(id,Prefix).Link7(n,id,buffer)
        )
      )

Link7(n:NAT,id:NAT,buffer:SIG_TUPLE)=
  rPCind(id).sPDreq(id,Prefix).Link7(n,id,buffer)
+
  sum(dest:NAT,
    sum(h:HEADER,
      sum(d:DATA,
        LDreq(id,dest,h,d).
          Link2resp(n,id,buffer,quadruple(dhead,
                                          sig(dest),
                                          sig(h,crc(h)),
                                          sig(d,crc(d))))
      )
    )
  )

Link2resp(n:NAT,id:NAT,buffer:SIG_TUPLE,p:SIG_TUPLE)=
  ( rPCind(id).sPDreq(id,Start).
    rPCind(id).sPDreq(id,first(p)).
    rPCind(id).sPDreq(id,second(p)) ).
    ( rPCind(id).sPDreq(id,third(p)).
      rPCind(id).sPDreq(id,fourth(p)).
      rPCind(id).sPDreq(id,End)).
        ( LDcon(id,broadsent).Link0(n,id,buffer)
             <| eq(getdest(second(p)),n) |>
               Link3(n,id,buffer)
        )

LinkWSA(n:NAT,id:NAT,buffer:SIG_TUPLE,dest:NAT)=
  sum(p:SIGNAL,
    rPDind(id,p).
      (
        Link0(n,id,buffer)
      <| is_sagap(p) |>
        LinkWSA(n,id,buffer,dest)
      )
  )
+
  (
    rPAcon(id,won).rPCind(id).sPDreq(id,End).Link0(n,id,buffer)
  <| eq(dest,id) |>
    delta
  )
```

A.3 The BUS process in μCRL

```
sort BoolTABLE
func
  empty: -> BoolTABLE
  btable: NAT#Bool#BoolTABLE -> BoolTABLE

map
  inita: NAT -> BoolTABLE
  invert: NAT#BoolTABLE -> BoolTABLE
  get: NAT#BoolTABLE -> Bool
  if: Bool#BoolTABLE#BoolTABLE -> BoolTABLE
  eq:BoolTABLE#BoolTABLE->Bool
var
  n,m : NAT
  b : Bool
bt1,bt2 : BoolTABLE
rew
  eq(bt1, bt1)=T
  inita(0)=empty
  inita(succ(n))=btable(n,F,inita(n))

  invert(n,empty)=empty
  invert(n,btable(m,b,bt1))=
    if(eq(n,m),
      btable(m,not(b),bt1),
      btable(m,b,invert(n,bt1))
    )
  get(n,btable(m,b,bt1))=if(eq(n,m),b,get(n,bt1))
  get(n,empty)=F
  if(T,bt1,bt2)=bt1
  if(F,bt1,bt2)=bt2

map
  zero,one,more: BoolTABLE -> Bool
var
  n : NAT
  bt : BoolTABLE
rew
  zero(empty)=T
  zero(btable(n,T,bt))=F
  zero(btable(n,F,bt))=zero(bt)
  one(empty)=F
  one(btable(n,T,bt))=zero(bt)
  one(btable(n,F,bt))=one(bt)
  more(bt)=and(not(zero(bt)),not(one(bt)))

act
```

```
  rPAreq: NAT#PAR
  rPDreq,sPDind: NAT#SIGNAL
  sPAcon: NAT#PAC
  sPCind: NAT
  arbresgap
  losesignal
```

proc

```
BUS(n:NAT)=
  BusIdle(n, inita(n))

BusIdle(n:NAT,t:BoolTABLE)=
  sum(id:NAT,
    sum(astat:PAR,
      rPAreq(id,astat).DecideIdle(n,t,id,astat)))
+
  arbresgap.BusIdle(n,inita(n)) <| not(zero(t)) |> delta

DecideIdle(n:NAT,t:BoolTABLE,id:NAT,astat:PAR)=
  ( sPAcon(id,won).BusBusy(n,invert(id,t),inita(n),inita(n),id) )
  <| not(get(id,t)) |>
  ( sPAcon(id,lost).BusIdle(n,t) )

BusBusy(n:NAT,
        t:BoolTABLE,
        next:BoolTABLE,
        destfault:BoolTABLE,
        busy:NAT)=
(
  (
    sPCind(busy).
      sum(p:SIGNAL,
        rPDreq(busy,p).Distribute(n,t,next,destfault,busy,p,0)
      )
  )
  <| lt(busy,n) |>
  (
      SubactionGap(n,t,0)
    <| zero(next) |>
      Resolve(n,t,next,0)
  )
)
+
  sum(j:NAT,
    rPAreq(j,fair).sPAcon(j,lost).BusBusy(n,t,next,destfault,busy)
  )
+
  sum(j:NAT,
    rPAreq(j,immediate).
      ( BusBusy(n,t,invert(j,next),destfault,busy)
          <| not(get(j,next)) |> delta )
  )

SubactionGap(n:NAT,t:BoolTABLE,i:NAT)=
  BusIdle(n,t)
<| eq(i,n) |>
  sPDind(i,subactgap).SubactionGap(n,t,succ(i))

Resolve(n:NAT,t:BoolTABLE,next:BoolTABLE,i:NAT)=
(
  (
    ( sPAcon(i,won).sPCind(i).Resolve(n,t,next,succ(i)) )
  <| get(i,next) |>
    ( tau.Resolve(n,t,next,succ(i)) )
  )
<| lt(i,n) |>
  Resolve2(n,t,next)
)

Resolve2(n:NAT,t:BoolTABLE,next:BoolTABLE)=
(
  sum(j:NAT,
    rPDreq(j,End).
      (
        Resolve2(n,t,invert(j,next))
      <| get(j,next) |>
        delta
      )
  )
<| more(next) |>
  sum(j:NAT,
    sum(p:SIGNAL,
      rPDreq(j,p).
        (
          SubactionGap(n,t,0)
        <| is_end(p) |>
          Distribute(n,t,inita(n),inita(n),j,p,0)
        )
    )
  )
)

Distribute(n:NAT,
           t:BoolTABLE,
           next:BoolTABLE,
           destfault:BoolTABLE,
           busy:NAT,
           p:SIGNAL,
           i:NAT)=
(
  (
    (
      %% Signals can be handed over correctly
      ( sPDind(i,p).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| or(not(is_header(p)),not(get(i,destfault))) |>
              delta )
    +
      %% Destination signals may be corrupted
      ( sum(dest:NAT,
          sPDind(i,sig(dest)).
            Distribute(n,t,next,invert(i,destfault),busy,p,succ(i))
        ) <| is_dest(p) |> delta )
    +
      %% Headers/Data/Acks may be corrupted
      ( sPDind(i,corrupt(p)).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| is_hda(p) |> delta )
    +
      %% Headers/Data/Acks may get lost
      ( losesignal.Distribute(n,t,next,destfault,busy,p,succ(i))
          <| is_hda(p) |> delta )
    +
      %% Packets may be too large
      ( sPDind(i,p).sPDind(i,Dummy).
          Distribute(n,t,next,destfault,busy,p,succ(i))
            <| is_data(p) |> delta )
    +
      ( rPAreq(i,immediate).
          ( Distribute(n,t,invert(i,next),destfault,busy,p,i)
              <| not(get(i,next)) |> delta ) )
    )
  <| not(eq(i,busy)) |>
    tau.Distribute(n,t,next,destfault,busy,p,succ(i))
  )
<| lt(i,n) |>
  (
    BusBusy(n,t,next,destfault,n)
  <| is_end(p) |>
    BusBusy(n,t,next,destfault,busy)
  )
)
```

A.4 The MAIN process in μCRL

act

```
  PDind,PDreq: NAT#SIGNAL
  PAcon: NAT#PAC
  PAreq: NAT#PAR
  PCind: NAT
```

comm

```
  rPDind|sPDind=PDind
  rPDreq|sPDreq=PDreq
  rPAcon|sPAcon=PAcon
  rPAreq|sPAreq=PAreq
  rPCind|sPCind=PCind
```

proc

```
 P1394(n:NAT)=
   hide({PDind, PDreq, PAcon, PAreq, PCind, arbresgap,losesignal},
     encap( {rPDind, sPDind, rPDreq, sPDreq, rPAcon,
             sPAcon, rPAreq, sPAreq, rPCind, sPCind},
         BUS(2) || LINK(2,0) || LINK(2,1)
       )
     )
```

% note: for 3 links, use BUS(3) || LINK(3,0) || LINK(3,1) || LINK(3,2), etc.

init `P1394(2)`

B Formal model in mCRL2

B.1 Types and functions in mCRL2

```
sort CHECK = struct bottom | check;

sort DATA = struct d1 | d2;

map crc : DATA -> CHECK;
eqn crc(d1)=check;
     crc(d2)=check;

sort HEADER = struct h1 | h2;

map crc : HEADER -> CHECK;
eqn crc(h1)=check;
     crc(h2)=check;

sort ACK = struct a1 | a2;

map crc : ACK -> CHECK;
eqn crc(a1)=check;
     crc(a2)=check;

sort SIGNAL = struct sig(getdest:Nat) ? is_dest |
                     sig(gethead:HEADER,gethcrc:CHECK) ? is_header |
                     sig(getdata:DATA,getdcrc:CHECK) ? is_data |
                     sig(getack:ACK,getacrc:CHECK) ? is_ack |
                     Start ? is_start |
                     End ? is_end |
                     Prefix ? is_prefix |
                     subactgap ? is_sagap |
                     dhead ? is_dhead |
                   Dummy ? is_dummy;

map is_physig,is_terminator : SIGNAL -> Bool;
    getcrc : SIGNAL -> CHECK;
var s : SIGNAL;
eqn is_physig(s) = is_start(s) || is_end(s) || is_prefix(s) || is_sagap(s);
    is_terminator(s)=is_end(s) || is_prefix(s);
    getcrc(s)=if(is_header(s),gethcrc(s),
             if(is_data(s),getdcrc(s),
             if(is_ack(s),getacrc(s),
                        bottom)));

map is_hda : SIGNAL -> Bool;
    valid_hpart, valid_ack : SIGNAL -> Bool;
var s : SIGNAL;
eqn is_hda(s)=is_header(s) || is_data(s) || is_ack(s);
    valid_ack(s)=if(is_ack(s),getacrc(s)==check,false);
    valid_hpart(s)=if(is_header(s),gethcrc(s)==check,false);

map corrupt : SIGNAL -> SIGNAL;
var h : HEADER;
    d : DATA;
    a : ACK;
    c : CHECK;
eqn corrupt(sig(h,c)) = sig(h,bottom);
    corrupt(sig(d,c)) = sig(d,bottom);
    corrupt(sig(a,c)) = sig(a,bottom);

sort SIG_TUPLE =
      struct quadruple (first:SIGNAL,
                        second:SIGNAL,
                        third:SIGNAL,
                        fourth:SIGNAL)
       | void ? is_void;

sort PAR = struct fair | immediate;

sort PAC = struct won | lost;

sort LDC = struct ackrec(ACK)
         | ackmiss
         | broadsent;

sort LDI = struct good (HEADER,DATA)
         | broadrec (HEADER,DATA)
         | dcrc_err (HEADER);

sort BOC = struct release | hold;
```

B.2 The LINK process in mCRL2

```
act
  LDreq : Nat#Nat#HEADER#DATA;
  LDcon : Nat#LDC;
  LDind : Nat#LDI;
  LDres : Nat#ACK#BOC;

  sPDreq,rPDind : Nat#SIGNAL;
  sPAreq : Nat#PAR;
  rPAcon : Nat#PAC;
  rPCind : Nat;

proc LINK(n:Nat,i:Nat)=Link0(n,i,void);

     Link0(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        is_void(buffer) ->
           ( sum dest:Nat,h:HEADER,d:DATA.
                ( dest<=n) -> LDreq(id,dest,h,d).
                     Link0(n,id,quadruple(dhead,
                                 sig(dest),
                                 sig(h,crc(h)),
                                 sig(d,crc(d))))<>delta) <>
             sPAreq(id,fair).Link1(n,id,buffer) +
        sum p:SIGNAL.
           rPDind(id,p).
              (is_start(p) -> Link4(n,id,buffer) <> Link0(n,id,buffer));

     Link1(n:Nat,id:Nat,p:SIG_TUPLE)=
        rPAcon(id,won).Link2req(n,id,p) +
        rPAcon(id,lost).Link0(n,id,p);

     Link2req(n:Nat,id:Nat,p:SIG_TUPLE)=
        rPCind(id).sPDreq(id,Start).
        rPCind(id).sPDreq(id,first(p)).
        rPCind(id).sPDreq(id,second(p)) .
        rPCind(id).sPDreq(id,third(p)).
        rPCind(id).sPDreq(id,fourth(p)).
        rPCind(id).sPDreq(id,End).
        ( (getdest(second(p))==n) ->
            LDcon(id,broadsent).Link0(n,id,void) <>
            Link3(n,id,void));

     Link3(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        sum p:SIGNAL.
           rPDind(id,p).
           ( is_prefix(p) -> Link3(n,id,buffer) <>
           ( is_start(p) -> Link3RA(n,id,buffer) <>
           ( is_sagap(p) -> LDcon(id,ackmiss).Link0(n,id,buffer) <>
                            LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
           )));

     Link3RA(n:Nat,id:Nat,buffer:SIG_TUPLE)=
        sum a:SIGNAL.
```

```
      rPDind(id,a).
      ( is_sagap(a) -> LDcon(id,ackmiss).Link0(n,id,buffer) <>
        ( is_physig(a) -> LDcon(id,ackmiss).LinkWSA(n,id,buffer,n) <>
                         Link3RE(n,id,buffer,a)));

Link3RE(n:Nat,id:Nat,buffer:SIG_TUPLE,a:SIGNAL)=
   sum e:SIGNAL.
      rPDind(id,e).
      ((valid_ack(a) && is_terminator(e)) ->
              LDcon(id,ackrec(getack(a))).LinkWSA(n,id,buffer,n) <>
        ( is_sagap(e) ->
              LDcon(id,ackmiss).Link0(n,id,buffer) <>
              LDcon(id,ackmiss).LinkWSA(n,id,buffer,n)
      ) );

Link4(n:Nat,id:Nat,buffer:SIG_TUPLE)=
   sum dh:SIGNAL.
      rPDind(id,dh).
      ( is_physig(dh) ->
        ( is_sagap(dh) ->
           Link0(n,id,buffer) <>
           LinkWSA(n,id,buffer,n)) <>
        Link4DH(n,id,buffer));

Link4DH(n:Nat,id:Nat,buffer:SIG_TUPLE)=
   sum dest:SIGNAL.rPDind(id,dest).
      ( is_dest(dest) ->
        ( (getdest(dest)==id) ->
             sPAreq(id,immediate).Link4RH(n,id,buffer,id) <>
             ( (getdest(dest)==n) ->
                 Link4RH(n,id,buffer,n) <>
                 LinkWSA(n,id,buffer,n)
             )
        ) <>
        ( is_sagap(dest) ->
             Link0(n,id,buffer) <>
             LinkWSA(n,id,buffer,n)
      ) );

Link4RH(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat)=
   sum h:SIGNAL.rPDind(id,h).
      ( valid_hpart(h) ->
           Link4RD(n,id,buffer,dest,h) <>
           LinkWSA(n,id,buffer,dest)
      );

Link4RD(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat,h:SIGNAL)=
   sum d:SIGNAL.
      rPDind(id,d).
        ( is_data(d) ->
             Link4RE(n,id,buffer,dest,h,d) <>
             LinkWSA(n,id,buffer,dest)
```

```
       );

Link4RE(n,id:Nat,buffer:SIG_TUPLE,dest:Nat,h:SIGNAL,d:SIGNAL)=
   sum e:SIGNAL.
      rPDind(id,e).
        ( is_terminator(e) ->
             ( (dest==id) ->
                  Link4DRec(n,id,buffer,h,d) <>
                  Link4BRec(n,id,buffer,h,d)
             ) <>
             LinkWSA(n,id,buffer,dest)
        );

Link4DRec(n:Nat,id:Nat,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
   (getcrc(d)==check) ->
      LDind(id,good(gethead(h),getdata(d))).rPAcon(id,won).Link5(n,id,buffer)
      <>
      LDind(id,dcrc_err(gethead(h))).rPAcon(id,won).Link5(n,id,buffer);

Link4BRec(n:Nat,id:Nat,buffer:SIG_TUPLE,h:SIGNAL,d:SIGNAL)=
   (getcrc(d)==check) ->
      LDind(id,broadrec(gethead(h),getdata(d))).Link0(n,id,buffer) <>
      Link0(n,id,buffer);

Link5(n,id:Nat,buffer:SIG_TUPLE)=
   sum a:ACK,b:BOC.LDres(id,a,b).Link6(n,id,buffer,sig(a,crc(a)),b) +
   rPCind(id).sPDreq(id,Prefix).Link5(n,id,buffer);

Link6(n:Nat,id:Nat,buffer:SIG_TUPLE,p:SIGNAL,b:BOC)=
   rPCind(id).sPDreq(id,Start).rPCind(id).sPDreq(id,p).rPCind(id).
      ( (b==release) ->
           sPDreq(id,End).Link0(n,id,buffer) <>
           sPDreq(id,Prefix).Link7(n,id,buffer)
      );

Link7(n,id:Nat,buffer:SIG_TUPLE)=
   rPCind(id).sPDreq(id,Prefix).Link7(n,id,buffer) +
   sum dest:Nat,h:HEADER,d:DATA. (dest<=n) ->
        LDreq(id,dest,h,d). Link2resp(n,id,buffer,
                 quadruple(dhead,sig(dest),sig(h,crc(h)),sig(d,crc(d))))<>delta;

Link2resp(n:Nat,id:Nat,buffer:SIG_TUPLE,p:SIG_TUPLE)=
   rPCind(id).sPDreq(id,Start).
   rPCind(id).sPDreq(id,first(p)).
   rPCind(id).sPDreq(id,second(p)).
   rPCind(id).sPDreq(id,third(p)).
   rPCind(id).sPDreq(id,fourth(p)).
   rPCind(id).sPDreq(id,End).
   ( (getdest(second(p))==n) ->
        LDcon(id,broadsent).Link0(n,id,buffer) <>
        Link3(n,id,buffer)
   );
```

```
    LinkWSA(n:Nat,id:Nat,buffer:SIG_TUPLE,dest:Nat)=
      sum p:SIGNAL.rPDind(id,p).
        ( is_sagap(p) ->
            Link0(n,id,buffer) <>
            LinkWSA(n,id,buffer,dest)
        ) +
      (dest==id) -> rPAcon(id,won).rPCind(id).sPDreq(id,End).Link0(n,id,buffer)<>delta;
```

B.3 The BUS process in mCRL2

```
sort BoolTABLE = List(struct pair(Nat,getbool:Bool));

map inita : Nat -> BoolTABLE;
    invert : Nat#BoolTABLE -> BoolTABLE;
    get : Nat#BoolTABLE -> Bool;
var n,m : Nat;
    b : Bool;
    bt1,bt2 : BoolTABLE;
eqn inita(0)=[];
    n>0 -> inita(n)=pair(Int2Nat(n-1),false)|>inita(Int2Nat(n-1));

    invert(n,[])=[];
    invert(n,pair(m,b)|>bt1)=
      if(n==m,pair(m,!b)|>bt1,pair(m,b)|>invert(n,bt1));
    get(n,[])=false;
    get(n,pair(m,b)|>bt1)=if(n==m,b,get(n,bt1));

map zero,one,more: BoolTABLE -> Bool;
var n : Nat;
    bt : BoolTABLE;
eqn zero([])=true;
    zero(pair(n,true)|>bt)=false;
    zero(pair(n,false)|>bt)=zero(bt);
    one([])=false;
    one(pair(n,true)|>bt)=zero(bt);
    one(pair(n,false)|>bt)=one(bt);
    more(bt)=!zero(bt) && !one(bt);

act rPAreq: Nat#PAR;
    rPDreq,sPDind: Nat#SIGNAL;
    sPAcon: Nat#PAC;
    sPCind: Nat;
    arbresgap;
    losesignal;
    internal;

proc BUS(n:Nat)=BusIdle(n, inita(n));

    BusIdle(n:Nat,t:BoolTABLE)=
      sum id:Nat,astat:PAR.(id<=n) ->
      rPAreq(id,astat).DecideIdle(n,t,id,astat)<>delta +
      !zero(t)->arbresgap.BusIdle(n,inita(n))<>delta;

DecideIdle(n:Nat,t:BoolTABLE,id:Nat,astat:PAR)=
   (!get(id,t)) ->
     sPAcon(id,won).BusBusy(n,invert(id,t),inita(n),inita(n),id) <>
     sPAcon(id,lost).BusIdle(n,t);

BusBusy(n:Nat,t,next,destfault:BoolTABLE,busy:Nat)=
   (busy<n) ->
      ( sPCind(busy).
          (sum p:SIGNAL.rPDreq(busy,p).Distribute(n,t,next,destfault,busy,p,0))
      ) <>
      ( zero(next) ->
          SubactionGap(n,t,0) <>
            Resolve(n,t,next,0)
      ) +
  sum j:Nat.(j<=n) ->
     rPAreq(j,fair).sPAcon(j,lost).BusBusy(n,t,next,destfault,busy)<>delta +
  sum j:Nat.(j<=n) -> rPAreq(j,immediate).
     (!get(j,next) -> BusBusy(n,t,invert(j,next),destfault,busy)<>delta)<>delta;

SubactionGap(n:Nat,t:BoolTABLE,i:Nat)=
   (i==n) ->
      BusIdle(n,t) <>
      sPDind(i,subactgap).SubactionGap(n,t,i+1);

Resolve(n:Nat,t,next:BoolTABLE,i:Nat)=
   (i<n) ->
   (get(i,next) ->
       sPAcon(i,won).sPCind(i).Resolve(n,t,next,i+1) <>
       internal.Resolve(n,t,next,i+1)
   ) <>
   Resolve2(n,t,next);

Resolve2(n:Nat,t:BoolTABLE,next:BoolTABLE)=
   more(next) ->
      (sum j:Nat.(j<=n) -> rPDreq(j,End).(get(j,next) ->
      Resolve2(n,t,invert(j,next))<>delta)<>delta) <>
      (sum j:Nat,p:SIGNAL.(j<=n) ->
         rPDreq(j,p).
            (is_end(p) ->
                SubactionGap(n,t,0) <>
                Distribute(n,t,inita(n),inita(n),j,p,0)
      )<>delta);

Distribute(n:Nat,t,next,destfault:BoolTABLE,busy:Nat,p:SIGNAL,i:Nat)=
   (i<n) ->
   ( (i!=busy) ->
     ( %% Signals can be handed over correctly
       (!is_header(p) || !get(i,destfault)) ->
          sPDind(i,p).Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Destination signals may be corrupted
       sum dest:Nat.(is_dest(p) && dest<=n) ->
          sPDind(i,sig(dest)).
            Distribute(n,t,next,invert(i,destfault),busy,p,i+1)<>delta +
       %% Headers/Data/Acks may be corrupted
       is_hda(p) ->
          sPDind(i,corrupt(p)).
            Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Headers/Data/Acks may get lost
       is_hda(p) ->
          losesignal.Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       %% Packets may be too large
       is_data(p) ->
          sPDind(i,p).sPDind(i,Dummy).
            Distribute(n,t,next,destfault,busy,p,i+1)<>delta +
       (!get(i,next)) ->
          rPAreq(i,immediate).
            Distribute(n,t,invert(i,next),destfault,busy,p,i)<>delta
   ) <>
   %% i==busy
   internal.Distribute(n,t,next,destfault,busy,p,i+1)
  ) <>
  %% i>=n
  ( is_end(p) ->
       BusBusy(n,t,next,destfault,n) <>
       BusBusy(n,t,next,destfault,busy)
  );
```

B.4 The MAIN process in mCRL2

```
act
  cPDreq,cPDind : Nat#SIGNAL;
  cPAreq : Nat#PAR;
  cPAcon : Nat#PAC;
  cPCind : Nat;

proc P1394(n:Nat)=
        allow({LDreq,LDcon,LDind,LDres},
           hide({arbresgap,losesignal,internal,cPDind,cPDreq,cPAcon,cPAreq,cPCind},
              comm({rPDind|sPDind->cPDind,rPDreq|sPDreq->cPDreq,rPAcon|sPAcon->cPAcon,
                    rPAreq|sPAreq->cPAreq,rPCind|sPCind->cPCind},
                  allow({LDreq,LDcon,LDind,LDres,arbresgap,losesignal,internal,
                              rPDind|sPDind,rPDreq|sPDreq,rPAcon|sPAcon,
                    rPAreq|sPAreq,rPCind|sPCind},
                          BUS(2) || LINK(2,0) || LINK(2,1)))));

% note: for 3 links, use BUS(3) || LINK(3,0) || LINK(3,1) || LINK(3,2), etc.

init P1394(2);
```

C Formal model in LOTOS

C.1 Types and functions in LOTOS

```
type CHECK is Boolean
   sorts
      CHECK
   opns
      bottom (*! constructor *) : -> CHECK
      check (*! constructor *) : -> CHECK
      eq : CHECK, CHECK -> Bool
   eqns
      forall x, y : CHECK
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
endtype

(*-------------------------------------------------------------------------*)

type DATA is CHECK
   sorts
      DATA
   opns
      d1 (*! constructor *) : -> DATA
      (* for verification, this type is restricted to a single value *)
      (* d2 {*! constructor *} : -> DATA *)
      crc : DATA -> CHECK
      eq : DATA, DATA -> Bool
   eqns
      forall x, y : DATA
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*-------------------------------------------------------------------------*)

type HEADER is CHECK
   sorts
      HEADER
   opns
      h1 (*! constructor *) : -> HEADER
      (* for verification, this type is restricted to a single value *)
      (* h2 {*! constructor *} : -> HEADER *)
      crc : HEADER -> CHECK
      eq : HEADER, HEADER -> Bool
   eqns
      forall x, y : HEADER
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*-------------------------------------------------------------------------*)

type ACK is CHECK
   sorts
      ACK
   opns
      a1 (*! constructor *) : -> ACK
      (* for verification, this type is restricted to a single value *)
      (* a2 {*! constructor *} : -> ACK *)
      crc : ACK -> CHECK
      eq : ACK, ACK -> Bool
   eqns
      forall x, y : ACK
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
      ofsort CHECK
         crc (x) = check;
endtype

(*-------------------------------------------------------------------------*)

type BOC is CHECK
   sorts
      BOC
   opns
      release (*! constructor *),
      hold (*! constructor *),
      no_op (*! constructor *) : -> BOC
      eq : BOC, BOC -> Bool
   eqns
      forall x, y : BOC
      ofsort Bool
         eq (x, x) = true;
         (* otherwise *) eq (x, y) = false;
endtype

(*-------------------------------------------------------------------------*)

type PHY_AREQ is CHECK
   sorts
      PHY_AREQ
   opns
      fair (*! constructor *),
      immediate (*! constructor *) : -> PHY_AREQ
      eq : PHY_AREQ, PHY_AREQ -> Bool
   eqns
     forall x, y : PHY_AREQ
     ofsort Bool
        eq (x, x) = true;
        (* otherwise *) eq (x, y) = false;
endtype

(*---------------------------------------------------------------------------*)

type PHY_ACONF is CHECK
   sorts
      PHY_ACONF
   opns
      won (*! constructor *),
      lost (*! constructor *) : -> PHY_ACONF
      eq : PHY_ACONF, PHY_ACONF -> Bool
   eqns
     forall x, y : PHY_ACONF
     ofsort Bool
        eq (x, x) = true;
        (* otherwise *) eq (x, y) = false;
endtype

(*---------------------------------------------------------------------------*)

type SIGNAL is ACK, CHECK, DATA, HEADER, NaturalNumber
   sorts
      SIGNAL
   opns
      destsig (*! constructor *) : Nat -> SIGNAL
      headsig (*! constructor *) : HEADER, CHECK -> SIGNAL
      datasig (*! constructor *) : DATA, CHECK -> SIGNAL
      acksig (*! constructor *) : ACK, CHECK -> SIGNAL
      dhead (*! constructor *) : -> SIGNAL
      Start (*! constructor *) : -> SIGNAL
      End (*! constructor *) : -> SIGNAL
      Prefix (*! constructor *) : -> SIGNAL
      subactgap (*! constructor *) : -> SIGNAL
      Dummy (*! constructor *) : -> SIGNAL
      is_dest, is_header, is_data, is_ack, is_physig : SIGNAL -> Bool
      valid_hpart, valid_ack : SIGNAL -> Bool
      getdest : SIGNAL -> Nat
      getdcrc : SIGNAL -> CHECK
      getdata : SIGNAL -> DATA
      gethead : SIGNAL -> HEADER
      getack : SIGNAL -> ACK
      corrupt : SIGNAL -> SIGNAL
      eq : SIGNAL, SIGNAL -> Bool
   eqns
      forall n : Nat, c : CHECK, h : HEADER, d : DATA, a : ACK, s, s1, s2 : SIGNAL
      ofsort Bool
```

```
        is_dest (destsig (n)) = true;
        (* otherwise *) is_dest (s) = false;
        is_header (headsig (h, c)) = true;
        (* otherwise *) is_header (s) = false;
        is_data (datasig (d, c)) = true;
        (* otherwise *) is_data (s) = false;
        is_ack (acksig (a, c)) = true;
        (* otherwise *) is_ack (s) = false;
        is_physig (Start) = true;
        is_physig (End) = true;
        is_physig (Prefix) = true;
        is_physig (subactgap) = true;
        (* otherwise *) is_physig (s) = false;
        valid_ack (acksig (a, c)) = eq (c, check);
        (* otherwise *) valid_ack (s) = false;
        valid_hpart (headsig (h, c)) = eq (c, check);
        (* otherwise *) valid_hpart (s) = false;
     ofsort Nat
        getdest (destsig (n)) = n;
        (* otherwise getdest (s) is undefined *)
     ofsort HEADER
        gethead (headsig (h, c)) = h;
        (* otherwise gethead (s) is undefined *)
     ofsort CHECK
        getdcrc (datasig (d, c)) = c;
        (* otherwise getdcrc (s) is undefined *)
     ofsort DATA
        getdata (datasig (d, c)) = d;
        (* otherwise getdata (s) is undefined *)
     ofsort ACK
        getack (acksig (a, c)) = a;
        (* otherwise getack (s) is undefined *)
     ofsort SIGNAL
        corrupt (headsig (h, c)) = headsig (h, bottom);
        corrupt (datasig (d, c)) = datasig (d, bottom);
        corrupt (acksig (a, c)) = acksig (a, bottom);
     ofsort Bool
        eq (s1, s1) = true;
        (* otherwise *) eq (s1, s2) = false;
endtype

(*-----------------------------------------------------------------------*)

type SIG_TUPLE is Boolean, SIGNAL
   sorts
      SIG_TUPLE
   opns
      quadruple (*! constructor *) : SIGNAL, SIGNAL, SIGNAL, SIGNAL -> SIG_TUPLE
      void (*! constructor *) : -> SIG_TUPLE
      first, second, third, fourth : SIG_TUPLE -> SIGNAL
      is_void : SIG_TUPLE -> Bool
   eqns
```

```
      forall s1, s2, s3, s4 : SIGNAL
      ofsort SIGNAL
         first (quadruple (s1, s2, s3, s4)) = s1;
         second (quadruple (s1, s2, s3, s4)) = s2;
         third (quadruple (s1, s2, s3, s4)) = s3;
         fourth (quadruple (s1, s2, s3, s4)) = s4;
      ofsort Bool
         is_void (void) = true;
         is_void (quadruple (s1, s2, s3, s4)) = false;
endtype

(*-------------------------------------------------------------------------*)

type LIN_DCONF is ACK
   sorts
      LIN_DCONF
   opns
      ackrec (*! constructor *) : ACK -> LIN_DCONF
      ackmiss (*! constructor *),
      broadsent (*! constructor *) : -> LIN_DCONF
endtype

(*-------------------------------------------------------------------------*)

type LIN_DIND is Boolean, DATA, HEADER
   sorts
      LIN_DIND
   opns
      good (*! constructor *),
      broadrec (*! constructor *) : HEADER, DATA -> LIN_DIND
      dcrc_err (*! constructor *) : HEADER -> LIN_DIND
      is_broadrec : LIN_DIND -> Bool
   eqns
      forall h: HEADER, d: DATA, xind: LIN_DIND
      ofsort Bool
         is_broadrec (broadrec (h, d)) = true;
         (* otherwise *) is_broadrec (xind) = false;
endtype

(*-------------------------------------------------------------------------*)

type BoolTABLE is Boolean, NaturalNumber
   sorts
      BoolTABLE
   opns
      empty (*! constructor *) : -> BoolTABLE
      btable (*! constructor *) : Nat, Bool, BoolTABLE -> BoolTABLE
      init : Nat -> BoolTABLE
      invert : Nat, BoolTABLE -> BoolTABLE
      get : Nat, BoolTABLE -> Bool
      zero, one, more : BoolTABLE -> Bool
   eqns
```

```
      forall n, n1, n2 : Nat, b : Bool, t : BoolTABLE
      ofsort BoolTABLE
         init (0) = empty;
         init (Succ (n)) = btable (n, false, init (n));
         invert (n, empty) = empty;
         n1 eq n2 => invert (n1, btable (n2, b, t)) = btable (n2, not (b), t);
         n1 ne n2 => invert (n1, btable (n2, b, t)) = btable (n2, b, invert (n1, t));
      ofsort Bool
         (* get (n, empty) is undefined *)
         n1 eq n2 => get (n1, btable (n2, b, t)) = b;
         n1 ne n2 => get (n1, btable (n2, b, t)) = get (n1, t);
      ofsort Bool
         zero (empty) = true;
         zero (btable (n, true, t)) = false;
         zero (btable (n, false, t)) = zero (t);
         one (empty) = false;
         one (btable (n, true, t)) = zero (t);
         one (btable (n, false, t)) = one (t);
         more (t) = not (zero (t)) and not (one (t));
endtype

(*------------------------------------------------------------------------*)

type Version is
   sorts
      Version
   opns
      ko (*! constructor *),
      ok (*! constructor *) : -> Version
endtype

(*------------------------------------------------------------------------*)

type Scenario is Boolean, Natural
   sorts
      Scenario
   opns
      scenario_1 (*! constructor *),
      scenario_2 (*! constructor *),
      scenario_3_2 (*! constructor *),
      scenario_3_3 (*! constructor *),
      scenario_3_4 (*! constructor *) : -> Scenario
      _eq_ : Scenario, Scenario -> Bool
   eqns
      forall s1, s2: Scenario
      ofsort Bool
         s1 eq s1 = true;
         (* otherwise *) s1 eq s2 = false;
endtype
```

C.2 The LINK process in LOTOS

```
process Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat) : noexit :=
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id, void)
endproc

(* ---------------------------------------------------------------------- *)

process Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   [is_void (buffer)] ->
      LDreq !id ?dest: Nat ?h: HEADER ?d: DATA;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, quadruple (dhead,
                               destsig (dest),
                               headsig (h, crc (h)),
                               datasig (d, crc (d))))
   []
   [not (is_void (buffer))] ->
      PAreq !id !fair;
      Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   PDind !id ?p: SIGNAL;
   (
   [eq (p, Start)] ->
     Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
           (n, id, buffer)
   []
   [not (eq (p, Start))] ->
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   )
endproc

(* ---------------------------------------------------------------------- *)

process Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PAcon !id !won;
   Link2req [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   PAcon !id !lost;
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer)
endproc

(* ---------------------------------------------------------------------- *)

process Link2req [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
```

```
               (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !first (buffer);
   PCind !id;
   PDreq !id !second (buffer);
   PCind !id;
   PDreq !id !third (buffer);
   PCind !id;
   PDreq !id !fourth (buffer);
   PCind !id;
   PDreq !id !End;
   (
   [getdest (second (buffer)) eq n] ->
      LDcon !id !broadsent;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, void)
   []
   [getdest (second (buffer)) ne n] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, void)
   )
endproc

(* ------------------------------------------------------------------ *)

process Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?p: SIGNAL;
   (
   [eq (p, Prefix)] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [eq (p, Start)] ->
      Link3RA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   []
   [eq (p, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (p, Prefix) or eq (p, Start) or eq (p, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
   )
endproc

(* ------------------------------------------------------------------ *)
```

```
process Link3RA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
  PDind !id ?a: SIGNAL;
  (
  [is_physig (a)] ->
    (
    [eq (a, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
    []
    [not (eq (a, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
    )
  []
  [not (is_physig (a))] ->
    Link3RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, a)
  )
endproc

(* ---------------------------------------------------------------------- *)

process Link3RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, a: SIGNAL) : noexit :=
  PDind !id ?e: SIGNAL;
  (
  [valid_ack (a) and (eq (e, End) or eq (e, Prefix))] ->
    LDcon !id !ackrec (getack (a));
    LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, n)
  []
  [not (valid_ack (a) and (eq (e, End) or eq (e, Prefix)))] ->
    (
    [eq (e, subactgap)] ->
      LDcon !id !ackmiss;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
    []
    [not (eq (e, subactgap))] ->
      LDcon !id !ackmiss;
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, n)
    )
  )
endproc

(* ---------------------------------------------------------------------- *)
```

```
process Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?dh: SIGNAL;
   (
   [is_physig (dh)] ->
      (
      [eq (dh, subactgap)] ->
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      []
      [not (eq (dh, subactgap))] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   []
   [not (is_physig (dh))] ->
      Link4DH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer)
   )
endproc

(* ------------------------------------------------------------------------ *)

process Link4DH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PDind !id ?dest: SIGNAL;
   (
   [is_dest (dest)] ->
      (
      [getdest (dest) eq id] ->
         PAreq !id !immediate;
         Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, id)
      []
      [getdest (dest) eq n] ->
         Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      []
      [(getdest (dest) ne n) and (getdest (dest) ne id)] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   []
   [not (is_dest (dest))] ->
      (
      [eq (dest, subactgap)] ->
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      []
      [not (eq (dest, subactgap))] ->
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      )
   )
endproc

(* --------------------------------------------------------------------------- *)

process Link4RH [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) : noexit :=
   PDind !id ?h: SIGNAL;
   (
   [valid_hpart (h)] ->
      Link4RD [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest, h)
   []
   [not (valid_hpart (h))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest)
   )
endproc

(* --------------------------------------------------------------------------- *)

process Link4RD [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id: Nat, buffer: SIG_TUPLE, dest: Nat, h: SIGNAL) : noexit :=
   PDind !id ?d: SIGNAL;
   (
   [is_data (d)] ->
      Link4RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest, h, d)
   []
   [not (is_data (d))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest)
   )
endproc

(* --------------------------------------------------------------------------- *)

process Link4RE [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id: Nat, buffer: SIG_TUPLE, dest: Nat, h: SIGNAL, d: SIGNAL)
               : noexit :=
   PDind !id ?e: SIGNAL;
   (
   [eq (e, End) or eq (e, Prefix)] ->
      (
      [dest eq id] ->
         Link4DRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, h, d)
      []
      [dest ne id] ->
         Link4BRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, h, d)
      )
   []
   [not (eq (e, End) or eq (e, Prefix))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (n, id, buffer, dest)
   )
endproc

(* ------------------------------------------------------------------- *)

process Link4DRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, h: SIGNAL, d: SIGNAL) : noexit :=
   [eq (getdcrc (d), check)]->
      LDind !id !good (gethead (h), getdata (d));
      PAcon !id !won;
      Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (getdcrc (d), check))] ->
      LDind !id !dcrc_err (gethead (h));
      PAcon !id !won;
      Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
endproc

(* ------------------------------------------------------------------- *)

process Link4BRec [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, h: SIGNAL, d: SIGNAL) : noexit :=
   [eq (getdcrc (d), check)] ->
      LDind !id !broadrec (gethead (h), getdata (d));
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (getdcrc (d), check))] ->
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
endproc

(* ------------------------------------------------------------------- *)

process Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   LDres !id ?a: ACK ?b: BOC;
   Link6 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer, acksig (a, crc (a)), b)
   []
   PCind !id;
   PDreq !id !Prefix;
   Link5 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer)
endproc
```

```
(* ---------------------------------------------------------------------- *)

process Link6 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id: Nat, buffer: SIG_TUPLE, p: SIGNAL, b: BOC) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !p;
   PCind !id;
   (
   [eq (b, release)] ->
      PDreq !id !End;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   []
   [not (eq (b, release))] ->
      PDreq !id !Prefix;
      Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
   )
endproc

(* ---------------------------------------------------------------------- *)

process Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id: Nat, buffer: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Prefix;
   Link7 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
         (n, id, buffer)
   []
   LDreq !id ?dest: Nat ?h: HEADER ?d: DATA;
   Link2resp [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer, quadruple (dhead,
                                       destsig (dest),
                                       headsig (h, crc (h)),
                                       datasig (d, crc (d))))
endproc

(* ---------------------------------------------------------------------- *)

process Link2resp [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, p: SIG_TUPLE) : noexit :=
   PCind !id;
   PDreq !id !Start;
   PCind !id;
   PDreq !id !first (p);
   PCind !id;
   PDreq !id !second (p);
   PCind !id;
   PDreq !id !third (p);
   PCind !id;
   PDreq !id !fourth (p);
   PCind !id;
   PDreq !id !End;
   (
   [getdest (second (p)) eq n] ->
      LDcon !id !broadsent;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [getdest (second (p)) ne n] ->
      Link3 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   )
endproc

(* ------------------------------------------------------------------------ *)

process LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) : noexit :=
   PDind !id ?p: SIGNAL;
   (
   [eq (p, subactgap)] ->
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   []
   [not (eq (p, subactgap))] ->
      LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer, dest)
   )
   []
   [dest eq id] ->
      PAcon !id !won;
      PCind !id;
      PDreq !id !End;
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
endproc
```

C.3 The BUS process in LOTOS

```
process Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
            (n: Nat) : noexit :=
   BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
endproc

(* ------------------------------------------------------------------------ *)

process BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n: Nat, t: BoolTABLE) : noexit :=
   PAreq ?id: Nat ?astat: PHY_AREQ [id lt n];
   DecideIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t, id,
              astat)
   []
   [not (zero(t))] ->
      arbresgap;
      BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
endproc

(* ------------------------------------------------------------------------ *)

process DecideIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n: Nat, t: BoolTABLE, id: Nat, astat: PHY_AREQ) : noexit :=
   [get (id, t) eq false] ->
      PAcon !id !won;
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n,
              invert (id, t), init (n), init (n), id)
   []
   [get (id, t) eq true] ->
      PAcon !id !lost;
      BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
endproc

(* ------------------------------------------------------------------------ *)

process BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n: Nat, t: BoolTABLE, next: BoolTABLE, destfault: BoolTABLE,
                busy: Nat) : noexit :=
   [busy lt n] ->
      PCind !busy;
      PDreq !busy ?p: SIGNAL;
      Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, next, destfault, busy, p, 0)
   []
   [not (busy lt n)] ->
      (
      [zero (next)] ->
         SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                      (n, t, 0)
      []
      [not (zero (next))] ->
          Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                  (n, t, next, 0)
      )
   []
   PAreq ?j: Nat !fair [j lt n];
   PAcon !j !lost;
   BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
           (n, t, next, destfault, busy)
   []
   PAreq ?j: Nat !immediate [not (get (j, next)) and (j lt n)];
   BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
```

```
          (n, t, invert (j, next), destfault, busy)
endproc

(* ------------------------------------------------------------------------ *)

process SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                  (n: Nat, t: BoolTABLE, j: Nat) : noexit :=
  [j eq n] ->
    BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
  []
  [j ne n] ->
    PDind !j !subactgap;
    SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n, t, succ (j))
endproc

(* ------------------------------------------------------------------------ *)

process Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
             (n: Nat, t: BoolTABLE, next: BoolTABLE, j: Nat) : noexit :=
  [j lt n] ->
    (
    [get (j, next) eq true] ->
      PAcon !j !won;
      PCind !j;
      Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, succ (j))
    []
    [get (j, next) eq false] ->
      Resolve [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, succ (j))
    )
  []
  [not (j lt n)] ->
    Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
            (n, t, next)
endproc

(* ------------------------------------------------------------------------ *)

process Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n: Nat, t: BoolTABLE, next: BoolTABLE) : noexit :=
  [more (next)] ->
    PDreq ?j: Nat !End [get (j, next) and (j lt n)];
    Resolve2 [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
            (n, t, invert (j, next))
  []
  [not (more (next))] ->
    PDreq ?j: Nat ?p: SIGNAL [j lt n];
    (
    [eq (p, End)] ->
      SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
```

```
                 (n, t, 0)
    []
    [not (eq (p, End))] ->
      Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, init (n), init (n), j, p, 0)
    )
endproc

(* ------------------------------------------------------------------------ *)

process Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n: Nat, t, next, destfault: BoolTABLE, busy: Nat, p: SIGNAL,
                   j: Nat) : noexit :=
  [j lt n] ->
    (
    [j ne busy] ->
      (
      [not (is_header (p)) or not (get (j, destfault))] ->
        PDind !j !p;
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, next, destfault, busy, p, succ (j))
      []
      [is_dest (p)] ->
        (
        choice dest: Nat []
          PDind !j !destsig (dest);
          Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                     (n, t, next, invert (j, destfault), busy, p, succ (j))
        )
      []
      [is_header (p) or (is_data (p) or is_ack (p))] ->
        PDind !j !corrupt (p);
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, next, destfault, busy, p, succ (j))
      []
      [is_header (p) or (is_data (p) or is_ack (p))] ->
        losesignal;
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, next, destfault, busy, p, succ (j))
      []
      [is_data (p)] ->
        PDind !j !p;
        PDind !j !Dummy;
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, next, destfault, busy, p, succ (j))
      []
      PAreq !j !immediate [not (get (j, next))];
        Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                   (n, t, invert (j, next), destfault, busy, p, j)
      )
    []
    [j eq busy] ->
```

```
      Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                (n, t, next, destfault, busy, p, succ (j))
    )
  []
  [not (j lt n)] ->
    (
    [eq (p, End)] ->
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, destfault, n)
    []
    [not (eq (p, End))] ->
      BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
              (n, t, next, destfault, busy)
    )
endproc
```

C.4 The TRANS process in LOTOS

```
process Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id: Nat, v: Version) : noexit :=
  hide TX0 in
    (
    TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
    |[TX0]|
    TransRes [LDind, LDres, TX0] (id, v)
    )
endproc

(*-------------------------------------------------------------------------*)

process TransReq [LDreq, LDcon, TDreq, TX0] (n, id: Nat) : noexit :=
  TDreq !id ?dest: Nat ?h: HEADER ?d: DATA [dest le n];
  (
  TX0;
  exit (dest, h, d)
  []
  exit (dest, h, d)
  ) >> accept dest: Nat, h: HEADER, d: DATA in
  (
  LDreq !id !dest !h !d;
    (
    [dest eq n] ->
      LDcon !id !broadsent;
      TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
    []
    [dest ne n] ->
      (
      choice a: ACK []
        LDcon !id !ackrec (a);
        TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
      )
    []
```

```
        LDcon !id !ackmiss;
        TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
      )
   )
endproc

(*--------------------------------------------------------------------------*)

process TransRes [LDind, LDres, TX0] (id: Nat, v: Version) : noexit :=
   LDind !id ?l: LIN_DIND;
   (
   [is_broadrec (l)] ->
      (
      [v = ko] ->
         (* original (incorrect) specification *)
         LDres !id !a1 !no_op;
         TransRes [LDind, LDres, TX0] (id, v)
      []
      [v = ok] ->
         (* correct specification *)
         TransRes [LDind, LDres, TX0] (id, v)
      )
   []
   [not (is_broadrec (l))] ->
      (
      choice a: ACK []
         (
         (* concatenated response = lock transaction *)
         TX0;
         LDres !id !a !hold;
         TransRes [LDind, LDres, TX0] (id, v)
         []
         (* split response *)
         LDres !id !a !release;
         TransRes [LDind, LDres, TX0] (id, v)
         )
      )
   )
endproc
```

C.5 The APPLI process in LOTOS

```
process Application [TDreq] (n: Nat, id: Nat, s: Scenario) : noexit :=
   [s eq scenario_1] ->
      [id eq 0] ->
         (
         (* send a request for transaction with a *different* node *)
         choice dest: Nat, h: HEADER, d: DATA []
            [(dest le n) and (dest ne id)] ->
               TDreq !id !dest !h !d;
               stop
```

```
        )
   []
   [s eq scenario_2] ->
      (
      (* send a request for transaction with a *different* node *)
      choice dest: Nat, h: HEADER, d: DATA []
         [(dest le n) and (dest ne id)] ->
            TDreq !id !dest !h !d;
            stop
      )
   []
   [(s eq scenario_3_2) or (s eq scenario_3_3) or (s eq scenario_3_4)] ->
      [id eq 0] ->
         (
         (* 2, 3 or 4 requests in sequence *)
         choice h: HEADER, d: DATA []
            TDreq !id !n !h !d;
            TDreq !id !n !h !d;
            (
            [s eq scenario_3_2] ->
               stop
            []
            [s eq scenario_3_3] ->
               TDreq !id !n !h !d;
               stop
            []
            [s eq scenario_3_4] ->
               TDreq !id !n !h !d;
               TDreq !id !n !h !d;
               stop
            )
        )
endproc
```

C.6 The NODE process in LOTOS

```
process Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n: Nat, id: Nat, v: Version, s: Scenario) : noexit :=
   hide TDreq in
      (
      Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id)
      |[LDreq, LDcon, LDind, LDres]|
      Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id, v)
      |[TDreq]|
      Application [TDreq] (n, id, s)
      )
endproc
```

C.7 The MAIN process in LOTOS

```
specification P1394 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind,
                  arbresgap, losesignal] : noexit

library
   BOOLEAN, NATURAL, DATA
endlib

behaviour

   (
   Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (2, 0, ko,
         scenario_3_4)
   |||
   Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (2, 1, ko,
         scenario_3_4)
   )
   |[PDreq, PDind, PAreq, PAcon, PCind]|
   Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (2)

where

   library
      APPLI, TRANS, LINK, BUS, NODE
   endlib

endspec
```

For model-checking purposes, a complementary file restricts the set of natural numbers, e.g., to the finite range $\{0,...,2\}$ in the above example.

D Formal model in LNT

D.1 Types and functions in LNT

```
module DATA is

type CHECK is
   bottom, check
   with =, <>
end type

----------------------------------------------------------------

type DATA is
   d1 -- , d2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (d: DATA): CHECK is
   use d; -- this parameter was not used in the LOTOS specification
   return check
end function
```

```
------------------------------------------------------------------------

type HEADER is
   h1 -- , h2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (h: HEADER): CHECK is
   use h; -- this parameter was not used in the LOTOS specification
   return check
end function

------------------------------------------------------------------------

type ACK is
   a1 -- , a2, ... for verification, this type is restricted to a single value
   with =, <>
end type

function crc (a: ACK): CHECK is
   use a; -- this parameter was not used in the LOTOS specification
   return check
end function

------------------------------------------------------------------------

type BOC is
   release, hold, no_op
   with =
end type

type PHY_AREQ is
   fair, immediate
   with =
end type

type PHY_ACONF is
   won, lost
   with =
end type

------------------------------------------------------------------------

type SIGNAL is
   destsig (dest: Nat),
   headsig (head: HEADER, crc: CHECK),
   datasig (data: DATA, crc: CHECK),
   acksig (ack: ACK, crc: CHECK),
   dhead,
   Start,
   End,
   Prefix,
   subactgap,
   Dummy
   with =, <>, get, set
end type

function is_dest (s: SIGNAL) : Bool is
   case s in
      destsig (any nat) -> return true
   | any -> return false
   end case
end function

function is_header (s: SIGNAL) : Bool is
   case s in
      headsig (any HEADER, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_data (s: SIGNAL) : Bool is
   case s in
      datasig (any DATA, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_ack (s: SIGNAL) : Bool is
   case s in
      acksig (any ACK, any CHECK) -> return true
   | any -> return false
   end case
end function

function is_physig (s: SIGNAL) : Bool is
   case s in
      Start | End | Prefix | subactgap -> return true
   | any -> return false
   end case
end function

function valid_hpart (s: SIGNAL) : Bool is
   return is_header (s) and then (s.crc = check)
end function

function valid_ack (s: SIGNAL) : Bool is
   return is_ack (s) and then (s.crc = check)
end function

function getdest (s: SIGNAL) : Nat is
   return s .[UNEXPECTED] dest
end function
```

```
function getdcrc (s: SIGNAL) : CHECK is
   assert is_data (s);
   return s .[UNEXPECTED] crc
end function

function getdata (s: SIGNAL) : DATA is
   return s .[UNEXPECTED] data
end function

function gethead (s: SIGNAL) : HEADER is
   return s .[UNEXPECTED] head
end function

function getack (s: SIGNAL) : ACK is
   return s .[UNEXPECTED] ack
end function

function corrupt (s: SIGNAL) : SIGNAL is
   case s in
      headsig (any HEADER, any CHECK) -> return s.{crc -> bottom}
   | datasig (any DATA, any CHECK) -> return s.{crc -> bottom}
   | acksig (any ACK, any CHECK) -> return s.{crc -> bottom}
   | any -> raise UNEXPECTED
   end case
end function

-----------------------------------------------------------------

type SIG_TUPLE is
   quadruple (dh, dest, header, data: SIGNAL),
   void
   with get
end type

function is_void (s: SIG_TUPLE) : Bool is
   case s in
      void -> return true
   | any -> return false
   end case
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_broadrec (x: LIN_DIND) : Bool is
   case x in
      broadrec (any HEADER, any DATA) -> return true
   | any -> return false
   end case
end function

------------------------------------------------------------------

type BoolTABLE is
   empty,
   btable (index: Nat, value: Bool, next: BoolTABLE)
   with =, get
end type

function init (n: Nat) : BoolTABLE is
   -- returns a table of size n initialized to false
   if n = 0 then
      return empty
   else
      return btable (n - 1, false, init (n - 1))
   end if
end function

function zero (t: BoolTABLE) : Bool is
   -- returns true iff no value in t is true
   if t = empty then
      return true
   elsif t.value then
      return false
   else
      return zero (t.next)
   end if
end function

function one (t: BoolTABLE) : Bool is
   -- returns true iff exactly one value in t is true
   if t = empty then
      return false
   elsif t.value then
      return zero (t.next)
   else
      return one (t.next)
   end if
end function
```

```
function more (t: BoolTABLE) : Bool is
   -- returns true iff more than one value in t is true
   return not (zero (t)) and not (one (t))
end function

function get (n: Nat, t: BoolTABLE) : Bool is
   -- returns the value associated with index n in t
   if t = empty then
      raise UNEXPECTED
   elsif t.index = n then
      return t.value
   else
      return get (n, t.next)
   end if
end function

function invert (n: Nat, t: BoolTABLE) : BoolTABLE is
   -- returns in which the value associated with index n is negated
   if t = empty then
      return empty
   elsif t.index = n then
      return btable (t.index, not (t.value), t.next)
   else
      return btable (t.index, t.value, invert (n, t.next))
   end if
end function

------------------------------------------------------------------------

type Version is
   ko, ok
end type

type Scenario is
   scenario_1, scenario_2, scenario_3_2, scenario_3_3, scenario_3_4
   with =
end type

function requests (s: Scenario): Nat is
   case s in
      scenario_3_2 -> return 2
   | scenario_3_3 -> return 3
   | scenario_3_4 -> return 4
   | any -> raise UNEXPECTED
   end case
end function

end module
```

D.2 Channels in LNT

```
module CHANNELS (DATA) is

channel Id is
   (n: Nat)
end channel

channel Sig is
   (id: Nat, flag: SIGNAL)
end channel

channel Areq is
   (id: Nat, flag: PHY_AREQ)
end channel

channel Acon is
   (id: Nat, flag: PHY_ACONF)
end channel

channel Ack is
   (id: Nat, a: ACK, b: BOC)
end channel

channel Dreq is
   (id: Nat, dest: Nat, h: HEADER, d: DATA)
end channel

channel Dind is
   (id: Nat, l: LIN_DIND)
end channel

channel Dcon is
   (id: Nat, l: LIN_DCONF)
end channel

end module
```

D.3 The LINK process in LNT

```
module LINK (DATA, CHANNELS) is

process Link [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
              PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat) is
   Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id, void)
end process

-----------------------------------------------------------------------

process Link0 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
               PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE) is
   select
      if is_void (buffer) then
```

```
        var dest: Nat, h: HEADER, d: DATA, b: SIG_TUPLE in
           LDreq (id, ?dest, ?h, ?d);
           b := quadruple (dhead,
                           destsig (dest),
                           headsig (h, crc (h)),
                           datasig (d, crc (d)));
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, b)
        end var
     else
        PAreq (id, fair);
        -- here, the LOTOS process Link1 was expanded in-line
        -- (see footnote 8 in the research report [Sighireanu-Mateescu-97])
        select
           PAcon (id, won);
           -- here, Link2 represents the LOTOS process Link2req
           Link2 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, void, buffer)
        []
           PAcon (id, lost);
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        end select
     end if
  []
     var p: SIGNAL in
        PDind (id, ?p);
        if p = Start then
           Link4 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        else
           Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer)
        end if
     end var
  end select
end process

---------------------------------------------------------------------

process Link1 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
  PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, p: SIGNAL) is
  -- process Link1 factors code repeated thrice in process Link3 below
  LDcon (id, ackmiss);
  if p = subactgap then
     Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
            (n, id, buffer)
  else
     LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer, n)
  end if
end process
```

```
-----------------------------------------------------------------

process Link2 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
  PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, p: SIG_TUPLE) is
   -- process Link2 unifies the two LOTOS processes Link2req and Link2resp
   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, End);
   if getdest (p.dest) = n then
      LDcon (id, broadsent);
      Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
             (n, id, buffer)
   else
      -- here, the LOTOS process Link3 was expanded in-line (called only once)
      var p, a, e: SIGNAL in
         loop L in
            PDind (id, ?p);
            if p <> Prefix then
               break L
            end if
         end loop;
         if p <> Start then
            Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                   (n, id, buffer, p)
         else
            -- here, the LOTOS process Link3RA was expanded (called only once)
            PDind (id, ?a);
            if is_physig (a) then
               Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                      (n, id, buffer, a)
            else
               -- here, the LOTOS process Link3RE was expanded (called only once)
               PDind (id, ?e);
               if valid_ack (a) and ((e = End) or (e = Prefix)) then
                  LDcon (id, ackrec (getack (a)));
                  LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                           PCind] (n, id, buffer, n)
               else
                  Link1 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                         (n, id, buffer, e)
               end if
            end if
         end if
      end var
   end if
end process

----------------------------------------------------------------

process Link4 [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
              PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE) is
   var s1, s2, s3, s4, s5: SIGNAL, dest: Nat in
      PDind (id, ?s1);
      if s1 = subactgap then
         Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
               (n, id, buffer)
      elsif is_physig (s1) then
         LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                 (n, id, buffer, n)
      else
         -- here, the LOTOS process Link4DH was expanded in-line (called only once)
         PDind (id, ?s2);
         if s2 = subactgap then
            Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id, buffer)
         elsif not (is_dest (s2)) or else
               ((getdest (s2) <> id) and (getdest (s2) <> n)) then
            LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                    (n, id, buffer, n)
         else
            dest := getdest (s2);
            if dest = id then
               PAreq (id, immediate)
            end if;
            -- here, the LOTOS process Link4RH was expanded (called only once)
            PDind (id, ?s3);
            if not (valid_hpart (s3)) then
               -- here, the LOTOS process Link4RD was expanded (called only once)
               LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                       (n, id, buffer, dest)
            else
               PDind (id, ?s4);
               if not (is_data (s4)) then
                  LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                          PCind] (n, id, buffer, dest)
               else
                  -- here, the LOTOS process Link4RE was expanded (called only once)
                  PDind (id, ?s5);
                  if (s5 <> End) and (s5 <> Prefix) then
                     LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                             PCind] (n, id, buffer, dest)
                  elsif dest <> id then
                     -- here, the LOTOS process Link4BRec was expanded (called only once)
                     if getdcrc (s4) = check then
                        LDind (id, broadrec (gethead (s3), getdata (s4)))
                     end if;
                     Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                            PCind] (n, id, buffer)
                  else
                     -- here, the LOTOS process Link4DRec was expanded (called only once)
                     if getdcrc (s4) = check then
                        LDind (id, good (gethead (s3), getdata (s4)))
                     else
                        LDind (id, dcrc_err (gethead (s3)))
                     end if;
                     PAcon (id, won);
                     -- here, the LOTOS process Link5 was expanded (called only once)
                     loop L in
                        select
                          PCind (id);
                          PDreq (id, Prefix)
                        []
                          break L
                        end select
                     end loop;
                     var a: ACK, b: BOC, p: SIGNAL in
                        LDres (id, ?a, ?b);
                        p := acksig (a, crc (a));
                        -- here, the LOTOS process Link6 was expanded (called only once)
                        PCind (id);
                        PDreq (id, Start);
                        PCind (id);
                        PDreq (id, p);
                        PCind (id);
                        if b = release then
                          PDreq (id, End);
                          Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon,
                                 PCind] (n, id, buffer)
                        else
                          PDreq (id, Prefix);
                          -- here, the LOTOS process Link7 was expanded (called only once)
                          loop L in
                             select
                                PCind (id);
                                PDreq (id, Prefix)
                             []
                                break L
                             end select
                          end loop;
                          var dest: Nat, h: HEADER, d: DATA, t: SIG_TUPLE in
                             LDreq (id, ?dest, ?h, ?d);
                             t := quadruple (dhead,
                                             destsig (dest),
                                             headsig (h, crc (h)),
                                             datasig (d, crc (d)));
                             -- here, Link2 represents the LOTOS process Link2resp
```

```
                        Link2 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq,
                               PAcon, PCind] (n, id, buffer, t)
                     end var
                  end if
               end var
            end if
         end if
      end if
    end if
  end if
 end var
end process

-----------------------------------------------------------------

process LinkWSA [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
   PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, buffer: SIG_TUPLE, dest: Nat) is
   select
      var p: SIGNAL in
         PDind (id, ?p);
         if p = subactgap then
               Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                     (n, id, buffer)
         else
               LinkWSA [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                       (n, id, buffer, dest)
         end if
      end var
   []
      only if dest = id then
            PAcon (id, won);
            PCind (id);
            PDreq (id, End);
            Link0 [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
                  (n, id, buffer)
      end if
   end select
end process

end module
```

D.4 The BUS process in LNT

```
module BUS (DATA, CHANNELS) is

process Bus [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
             arbresgap, losesignal: none] (n: Nat) is
   BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, init (n))
end process

-----------------------------------------------------------------
```

```
process BusIdle [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                 arbresgap, losesignal: none] (n: Nat, t: BoolTABLE) is
   select
      var id: Nat in
         PAreq (?id, ?any PHY_AREQ) where id < n;
         -- here, the LOTOS process DecideIdle was expanded in-line
         -- (see footnote 7 in the research report [Sighireanu-Mateescu-97])
         if get (id, t) = false then
            PAcon (id, won);
            BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                    (n, invert (id, t), init (n), init (n), id)
         else
            PAcon (id, lost);
            BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
         end if
      end var
   []
      only if not (zero (t)) then
         arbresgap;
         BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, init (n))
      end if
   end select
end process

-------------------------------------------------------------------------------

process BusBusy [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                 arbresgap, losesignal: none] (n: Nat, t: BoolTABLE,
                 in var next: BoolTABLE, destfault: BoolTABLE, busy: Nat) is
   select
      var j: Nat in
         PAreq (?j, fair) where j < n;
         PAcon (j, lost);
         BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, next, destfault, busy)
      end var
   []
      var j: Nat in
         PAreq (?j, immediate) where not (get (j, next)) and (j < n);
         BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                 (n, t, invert (j, next), destfault, busy)
      end var
   []
      if busy < n then
         var p: SIGNAL in
            PCind (busy);
            PDreq (busy, ?p);
            Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                       (n, t, next, destfault, busy, p)
         end var
```

```
    elsif zero (next) then
      SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
    else
      -- here, the LOTOS process Resolve was expanded (called only once)
      var j: Nat, p: SIGNAL in
        for j := 0 while j < n by j := j + 1 loop
          if get (j, next) then
            PAcon (j, won);
            PCind (j)
          end if
        end loop;
        -- here, the LOTOS process Resolve2 was expanded (called only once)
        while more (next) loop
          PDreq (?j, End) where get (j, next) and (j < n);
          next := invert (j, next)
        end loop;
        PDreq (?j, ?p) where j < n;
        if p = End then
          SubactionGap [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                       (n, t)
        else
          Distribute [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
                     (n, t, init (n), init (n), j, p)
        end if
      end var
    end if
  end select
end process

-----------------------------------------------------------------------------

process SubactionGap [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                      arbresgap, losesignal: none] (n: Nat, t: BoolTABLE) is
  var j: Nat in
    for j := 0 while j < n by j := j + 1 loop
      PDind (j, subactgap)
    end loop;
    BusIdle [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (n, t)
  end var
end process

-----------------------------------------------------------------------------

process Distribute [PAreq: Areq, PDreq, PDind: Sig, PAcon: Acon, PCind: Id,
                    arbresgap, losesignal: none] (n: Nat, t: BoolTABLE,
                    in var next, destfault: BoolTABLE, busy: Nat, p: SIGNAL) is
  var j, incr: Nat in
    for j := 0 while j < n by j := j + incr loop
      incr := 1;
      if j <> busy then
        select
          only if not (is_header (p) and get (j, destfault)) then
```

```
               PDind (j, p)
            end if
         []
            only if is_dest (p) then
               var dest: Nat in
                  dest := any Nat;
                  PDind (j, destsig (dest));
                  destfault := invert (j, destfault)
               end var
            end if
         []
            only if is_header (p) or is_data (p) or is_ack (p) then
               select
                  PDind (j, corrupt (p))
               []
                  losesignal
               end select
            end if
         []
            only if is_data (p) then
               PDind (j, p);
               PDind (j, Dummy)
            end if
         []
            PAreq (j, immediate) where not (get (j, next));
            incr := 0; -- instead of 1, here
            next := invert (j, next)
         end select
      end if
   end loop;
   if p = End then
      j := n
   else
      j := busy
   end if;
   BusBusy [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal]
           (n, t, next, destfault, j)
 end var
end process

end module
```

D.5 The TRANS process in LNT

```
module TRANS (DATA, CHANNELS) is

process Trans [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, TDreq: Dreq]
             (n, id: Nat, v: Version) is
 hide TX0: none in
   par TX0 in
      TransReq [LDreq, LDcon, TDreq, TX0] (n, id)
   ||
      TransRes [LDind, LDres, TX0] (id, v)
   end par
 end hide
end process

----------------------------------------------------------------

process TransReq [LDreq: Dreq, LDcon: Dcon, TDreq: Dreq, TX0: none] (n, id: Nat) is
 var dest: Nat, h: HEADER, d: DATA, a: ACK in
   loop
      TDreq (id, ?dest, ?h, ?d) where dest <= n;
      select
         TX0
      []
         null
      end select;
      i; -- this "i" corresponds to the ">>" operator in the LOTOS specification
      LDreq (id, dest, h, d);
      select
         if dest = n then
            LDcon (id, broadsent)
         else
            a := any ACK;
            LDcon (id, ackrec (a))
         end if
      []
         LDcon (id, ackmiss)
      end select
   end loop
 end var
end process

----------------------------------------------------------------

process TransRes [LDind: Dind, LDres: Ack, TX0: none] (id: Nat, v: Version) is
 var l: LIN_DIND, a: ACK in
   loop
      LDind (id, ?l);
      if is_broadrec (l) then
         case v in
            ko ->
               -- original (incorrect) specification
               LDres (id, a1, no_op)
         | ok ->
               -- correct specification
               null
         end case
      else
         a := any ACK;
         select
            -- concatenated response = lock transaction
            TX0;
            LDres (id, a, hold)
         []
            -- split response
            LDres (id, a, release)
         end select
      end if
   end loop
 end var
end process

end module
```

D.6 The APPLI process in LNT

```
module APPLI (DATA, CHANNELS) is

process Application [TDreq: Dreq] (n: Nat, id: Nat, s: Scenario) is
   var dest: Nat, h: HEADER, d: DATA, r: Nat in
      case s in
         scenario_1 ->
            only if id == 0 then
               -- send a request for transaction with a *different* node
               dest := any Nat where (dest <= n) and (dest <> id);
               h := any HEADER;
               d := any DATA;
               TDreq (id, dest, h, d);
               stop
            end if
      | scenario_2 ->
            -- send a request for transaction with a *different* node
            dest := any Nat where (dest <= n) and (dest <> id);
            h := any HEADER;
            d := any DATA;
            TDreq (id, dest, h, d);
            stop
      | scenario_3_2 | scenario_3_3 | scenario_3_4 ->
            only if id == 0 then
               h := any HEADER;
               d := any DATA;
               for r := requests (s) while r > 0 by r := r - 1 loop
                  TDreq (id, n, h, d)
               end loop;
               stop
            end if
      end case
   end var
end process

end module
```

D.7 The NODE process in LNT

```
module NODE (DATA, CHANNELS, APPLI, TRANS, LINK) is

process Node [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind: Sig,
          PAreq: Areq, PAcon: Acon, PCind: Id] (n, id: Nat, v: Version, s: Scenario) is
   hide TDreq: Dreq in
      par
         TDreq ->
            Application [TDreq] (n, id, s)
      ||
         TDreq, LDreq, LDcon, LDind, LDres ->
            Trans [LDreq, LDcon, LDind, LDres, TDreq] (n, id, v)
      ||
         LDreq, LDcon, LDind, LDres ->
            Link [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind] (n, id)
      end par
   end hide
end process

end module
```

D.8 The MAIN process in LNT

```
module scen3_orig_2_4 (APPLI, TRANS, LINK, NODE, BUS) is

!nat_sup 2

process MAIN [LDreq: Dreq, LDcon: Dcon, LDind: Dind, LDres: Ack, PDreq, PDind:
   Sig, PAreq: Areq, PAcon: Acon, PCind: Id, arbresgap, losesignal: none] is
   par PDreq, PDind, PAreq, PAcon, PCind in
      par
         Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (2, 0, ko, scenario_3_4)
      ||
         Node [LDreq, LDcon, LDind, LDres, PDreq, PDind, PAreq, PAcon, PCind]
              (2, 1, ko, scenario_3_4)
      end par
   ||
      Bus [PAreq, PDreq, PDind, PAcon, PCind, arbresgap, losesignal] (2)
   end par
end process

end module
```