Description and Formal Specification of the Link Layer of P1394

S.P. Luttik

Software Engineering (SEN)

SEN-R9706 May 31, 1997

Report SEN-R9706
ISSN 1386-369X

CWI
P.O. Box 94079
1090 GB Amsterdam
The Netherlands

Description and Formal Specification of the Link Layer of P1394

S.P. Luttik
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

Programming Research Group, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands

Email: luttik@cwi.nl

ABSTRACT
We give a formal specification in μCRL of the Link Layer as described in the IEEE Standard P1394 [IEE95] that may serve as a starting point for further verification.

1991 Mathematics Subject Classification: 68M10 Computer networks; 68Q60 Specification and verification of programs
1991 Computing Reviews Classification System: C.2.2 Network Protocols; D.2.1 Requirements/Specifications; D.2.4 Program Verification
Keywords and Phrases: P1394, μCRL, Serial Bus Protocol.
Note: Research supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-33-008. Work carried out under project SEN 2.1 Process Specification and Analysis.

1. Introduction

P1394 is an IEEE Standard describing a "High Performance Serial Bus" [IEE95]. It deals both with the physical requirements and the protocol of the bus. The main feature of P1394 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 acknowledge.

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, 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 (for instance, audio/video data).

We give a specification of part of the P1394 protocol in μCRL [GP94], a formalism based on process algebra (see e.g. [Mil89, BW90]) that has the possibility of including algebraic data descriptions. Our specification serves as a case study in the application of formal methods and forms a starting point for further verification of the protocol.

We specify the behaviour in asynchronous mode of the Link Layer, the middle layer of the three layered protocol; 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. Moreover, we specify a Bus-process, describing the external behaviour of the underlying physical components according to P1394, in order to be able to simulate the situation where a number of Link Layers communicate asynchronously.

The rest of this section is devoted to an informative introduction to P1394. In Section 2 we will discuss our specification of the Link Layer protocol and in Section 3 we will explain our specification of a process describing the external behaviour of a number of P1394-compliant Physical Layers connected by a cable. The actual specifications are included as appendices. In Section 4 we will indicate a number of properties that should hold w.r.t. our specification.

1.1 The P1394 Protocol

The 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 line (to which we will refer by the term CABLE in the sequel).

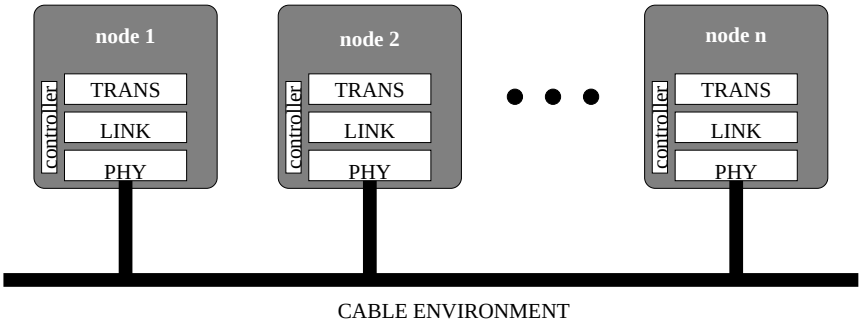

Figure 1: Serial Bus architecture.

The behaviour of each node in asynchronous mode is described by three layers:

1. The Transaction Layer (the upper layer; indicated by TRANS) offers three types of transactions to the application(s) running on the node: applications may want to perform *read transactions* (read data from another node), *write transactions* (write data to another node), and *lock transactions* (have some of its own data processed by another node after which it is transferred back). Such transactions consist of a *request* and a *response*; TRANS can both handle 'concatenated response' transactions (response follows request immediately) as well as 'split' transactions (response not necessarilly follows immediately on the request it belongs to).

2. The Link Layer (referred to as LINK) is the middle layer. It forms the interface between TRANS and the physical components of the bus (consisting of the 'Physical Layers' (PHYs) connected by a serial line (CABLE), together denoted by BUS). LINK provides two types of services to TRANS:

   Data request/response: By means of a 'LINK Data request' TRANS instructs LINK to send a packet to some particular node or to broadcast a packet to all other nodes. TRANS must react on a packet addressed to it by sending an acknowledge by means of a 'LINK Data response'.

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

   LINK divides the stream of packets that it sees on BUS into an alternating sequence of 'subactions' and 'subaction gaps'; the latter being time intervals having a specified minimal length during which 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; 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.

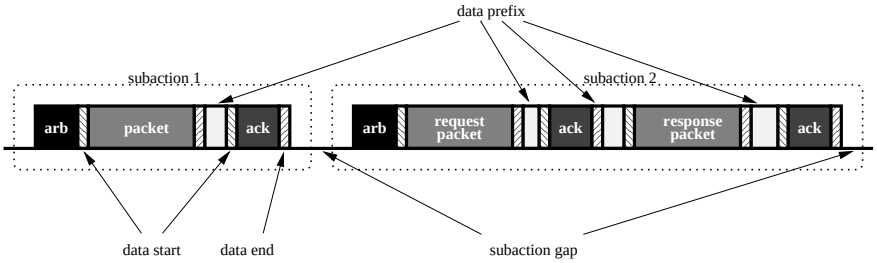

Figure 2: LINK operation

Before a packet can be put on BUS, LINK must first gain access by issueing an arbitration procedure. Moreover LINK must transform the requests of TRANS 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 put on BUS by any of the nodes is received by each LINK; however, only packets addressed to the node that LINK is a part of is forwarded to TRANS. LINK also directs the process of waiting for acknowledgements.

3. The physical connection between a node and the serial line is called the Physical Layer (PHY). It listens to and puts signals on CABLE, measures the lengths of the time intervals during which CABLE resides in an idle state and determines together with the other PHYs which node has control over CABLE (arbitration). It provides the following services to LINK:

   Arbitration request/confirmation: LINK instructs PHY to start an arbitration procedure by means of a 'PHY Arbitration request'. The result of this procedure (either 'won' or 'lost') is communicated to LINK by means of a 'PHY Arbitration confirmation'.

   Data request/indication: The Link Layer instructs PHY to put some signal on CABLE by means of a 'PHY Data request'. PHY indicates a signal on CABLE (or information about the status of CABLE) to LINK by means of a 'PHY Data indication'.

   Clock indication: To notify LINK that it can (and should!) put a signal on CABLE, PHY communicates a 'PHY Clock indication'.

According to [IEE95] there is also a so-called 'node controller' that can influence each of the three layers. Since in asynchronous mode the rôle of this node controller is restricted to the ability to reset each of the three layers (force them into their initial state), we have left it out of our specification.

2. Explanation of Link-Specification

In [IEE95], data descriptions of LINK mainly consist of a number of tables, that tables define the lengths of several different streams of bits and how to interpret them as packets.

The operational behaviour of LINK is described by means of an (incomplete) state-transition diagram and some informal text, meant to impose additional restrictions on the states and transitions of this diagram.

We apply some abstractions to the data in our specification and focus on the process part of LINK, which we try to describe as precise as possible. In this section, we will explain some details of our specification of LINK (appendix B). First, we introduce some elementary and auxiliary data types that we need in the specification. Then, we describe a single packet format that is used by LINK to communicate with the LINKs of other nodes, via BUS). Finally, we discuss the process part of the specification.

2.1 Auxiliary Data Types of LINK

We need in our specification a sort $\mathbb{B}$ with constants $\mathtt{t}$ and $\mathtt{f}$, the logical connectives not, or and and and a function $\mathrm{if}(x, y, z)$ that chooses between its second and third argument upon the value of x, and an equality relation $eq(x, y)$. Moreover, we have a sort $\mathbb{N}$ of natural numbers, with constructors 0 and $\mathrm{succ}(n)$, a 'less than'-relation $\mathrm{lt}(n, m)$, and a predicate $eq(x, y)$ reflecting the standard equality on naturals.

Furthermore, we have five enumerated types that encode a number of attributes used in the communication between LINK and TRANS and between LINK and PHY:

- The sort BOC (for 'Bus Occupancy Control') contains two elements: `release`, and `hold`. It offers TRANS the possibility to indicate LINK whether a 'split' transaction (`release`) or a 'concatenated response' transaction (`hold`) is requested. Namely, in the first case LINK must release the bus after sending acknowledge packet, while in the second case the bus must be held to allow the sending of a response packet.
- The sort LDI contains the attributes LINK must attach to an incoming packet when it forwards it to TRANS.
- The sort LDC contains the values by which LINK informs TRANS of the success or failure of a transaction.
- The sort PAR contains the two possible arbitration modes[1] by which LINK can request PHY to gain access to CABLE.
- After PHY has performed the arbitration procedure, a result of either `won` or `lost` (sort PAC) is communicated back to LINK.

2.2 Packet Formats

The sorts DATA, HEADER, and ACK specify the actual contents of packets. In our specification these three sorts each contain two elements, but this could have been any number of distinct elements, given that there is a function crc, that maps each element to some checksum in CHECK. It is not really important how such a checksum is computed, but the function crc should, of course, have an inverse. We have specified two checksum values: $\top$ for 'good crc' and $\bot$ for 'bad crc'. We abstracted here a little bit from the original descriptions in [IEE95]. There, the sorts DATA, HEADER, and ACK contain sequences of 128, 112, and 4 bits respectively. Data elements and headers have a 32 bit checksum; acknowledgements have a 4 bit checksum.

In [IEE95] a number of asynchronous packet formats are defined for each kind of transaction. These formats are all in line with a general format for asynchronous packets, but differ in length, specific values of some fields of the header, and the presence of one or more data elements. We have specified just a single asynchronous packet format, consisting of a *destination* (from $\mathbb{N}$), a *header part* (an element from HEADER together with its crc) and one *data part* (an element from DATA and its crc. The destination and the header part together form the header as described in [IEE95]. Actually, the header part should contain al kinds of control fields that are not relevant on Link Layer level.

Every node that is connected to the CABLE knows its own identification number and the total number of nodes that are connected to the CABLE. If there are n nodes, then it is assumed that they have identification numbers 0 through $n-1$. Now, if the destination field of a packet contains either of the values 0 through $n-1$, then this is interpreted as the address of the node with this value as identification number. If the destination field has value n, then the packet is considered a broadcast packet, i.e. a packet that is addressed to all nodes (except for the sending node).

As in [IEE95], acknowledge packets consist of an *acknowledge (code)* and an *acknowledge checksum* (also called 'parity'), together having a fixed length of 1 byte (i.e. 8 bits). By means of a 'LINK Data

[1] a third possibility is mentioned in [IEE95], but that one is not relevant in asynchronous mode.

response' TRANS communicates the acknowledge code that should be put on the bus, as well as an instruction for LINK to either `release`, or `hold` control over bus.

According to [IEE95], LINK should communicate packets via BUS to other LINKs serially, that is, LINK must send one bit as soon as PHY indicates that it is ready to put a new bit on CABLE. Instead, we chose divide packets into *signals* (from $\mathbb{S}$). In this way we obtain a more concise specification, avoiding a number of complications that do not influence LINK's behaviour. We do make sure that our specification still reflects naturally the serial mechanisms as described in the standard. A signal is either a *control signal* or a *data signal*. We distinguish four types of data signals, which are embedded in the sort $\mathbb{S}$ by means of the function `sig`:

destination signals: constructed by applying `sig` to a natural number — 'getdest' yields the destination from such a signal;

header signals: constructed by `sig` from a header and a checksum — 'gethead' yields the header from such a signal;

data signals: constructed by `sig` from a data part and a checksum — 'getdata' and 'getdcrc' respectively yield the data element and the checksum of such a signal;

acknowledge signals: constructed by `sig` from an acknowledge and a checksum – 'getack' yields the acknowledge part of such a signal.

The predicates dest?(s), header?(s), data?(s) and ack?(s) respectively yield `t` if s is a destination, a header, a data element, or an acknowledge. Furthermore, valid_hpart(s) and valid_ack(s) equal `t` if s is a header or an acknowledge signal (respectively) with a checksum value of $\top$.

To simulate the situation of [IEE95] where incoming destinations (consisting of 2 bytes) and acknowledgements (consisting of 1 byte) are distinguished by their length, we have an additional signal that is considered a data signal: `dhead`. Thus, destinations must be transmitted by means of *two* signals, `dhead` and the 'real' destination, whereas acknowledgements consist of *one* signal.

The sort $\mathbb{S}$ contains four control signals:

`start`: used to mark the beginning of a packet;

`end`, `Prefix`: both used to mark the end of a packet; `end` indicates PHY to release control over CABLE afterwards; `Prefix` tells PHY not to release control over CABLE;

`subactgap`: is used by PHY to notice LINK that it has detected that CABLE has been idle for some specific amount of time.

The predicate physig? holds when a signal is one of the four control signals; terminator? determines it is `start`, `Prefix`, or `end`; and hda? is `t` if the signal is either a header signal, a data signal or an acknowledgement signal.

Within LINK, asynchronous packets reside in a buffer that is either empty (`void`) or contains a quadruple of four signals: a destination header, a destination signal, a header signal and a data signal (see SIG_TUPLE). Acknowledge packets will be represented just by their signal.

2.3 Link Layer Operation

Our specification of the process behaviour of LINK is based on the state machine as depicted in [IEE95, Figure 6-19, Page 174] and the informal explanation it is accompanied by. We have derived process names of the names of the states in the state machine as follows: process LINKn for $0 \leq n \leq 7$ corresponds to state Ln. Each process receives at least three parameters: the total number of nodes that are connected to CABLE, the identification number of the particular node, and a buffer to contain a pending request of TRANS. Each action receives the identification number as its first parameter.

LINK starts as process LINK0 with an buffer that is `void`; it can either receive a data request from TRANS ($\mathsf{LDreq}(id, dest, h, d)$) or a data indiation from PHY ($\mathsf{rPDind}(id, p)$).

At a data request, a packet s (`quadruple`) is constructed from the parameters put into the buffer. Since the buffer then is not `void` anymore LINK starts a `fair` arbitration procedure to gain access to BUS (sPAreq(id, `fair`)). If this results in `won` then the underlying PHY controls CABLE, so LINK enters 'send mode' (LINK2_{req}, see below). However, it can also happen that PHY indicates a the arrival of data: the packet is stored in *buffer* and the data is received first (LINK4).

At a data indiction, it must be checked whether the received signal is a `start`-signal. If it is, then this indicates that some node has control over CABLE and is sending a packet. The incoming packet must be received in 'receive mode' (LINK4). However, if the signal was not a `start`-signal, it may be ignored.

Send Mode As soon as it has gained control over CABLE, PHY starts indicating clock signals (rPCind(id)) to notify LINK that it should send a signal. LINK is supposed to respond to every such clock indication and sends the entire packet —delimited by a `start`- and an `end`-signal—, one signal at a time (LINK2_{req}).

The `end` signal also notifies PHY that LINK has sent all of its packet; it will cease to send clock indications. Upon the value of the destination field, LINK either informs TRANS that a broadcast packet was sent (LDcon(id, `broadsent`)) properly (LINK0), or it must wait for an acknowledge packet (LINK3).

The acknowledge packet must arrive within some specific amount of time: if a `subactgap` occurs before an acknowledgement with valid checksum has been received entirely (i.e. up to and including the terminating `end` signal), then LINK will act as if the acknowledge is missing. Recall that an acknowledge packet can be identified by its length (1 signal). As a `start` signal arrives, LINK evolves into LINK3_{RA}, expecting an acknowledge signal. If the next signal is indeed a data signal, then LINK receives the terminating `end` signal (LINK3_{RE}), checks the validity of the received acknowledge signal and sends an 'acknowledgement received' (`ackrec`) to TRANS. On the other hand, if anything goes wrong (for instance, another data signal arrives, there is no terminating `end`, or the acknowledge packet is not valid), then LINK sends 'acknowledgement missing' (`ackmiss`) to TRANS. Both in case of failure and in case of success, LINK does wait for an indication of PHY that a 'subaction gap' (`subactgap`) has occurred, before it returns the initial state (LINKWSA). Of course, if a subaction gap interferes in the above described behaviour, LINK should immediately send an `ackmiss` and return to its initial state.

Receive Mode If LINK receives a `start` signal is indicated, it enters 'receive mode' (LINK4) and expects to see a packet being put on BUS by some other LINK. Asynchronous packets consist of 4 signals, and LINK will 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` (LINK4_{DH}), then it has received an acknowledge packet, which it should ignore: it will wait for the next subaction gap and return to the initial state (LINKWSA). If the second signal is indeed a destination signal, then it must check whether the incoming packet is either (1) a packet addressed to it, (2) a broadcast packet or (3) a packet destinated to some other node.

In the first case it must notify PHY that it wants access to BUS as soon as the packet has been received entirely, in anticipation of sending an acknowledge. It does this by means of an `immediate` arbitration request sPAreq(id, `immediate`). Broadcast packets, however, should not be acknowledged, so in the second case no such request is needed. In both cases LINK evolves into state LINK4_{RH}; by means of the parameter *dest*, it still can decide whether the incoming packet was broadcast or not. In the third case LINK should completely ignore the packet; it returns to LINK0 at the next subaction gap (LINKWSA).

The third signal is expected to be a header signal (LINK4_{RH}), and the fourth signal should be a data signal (LINK4_{RD}). If the packet is correctly terminated by either an `end` signal or a `Prefix` signal, then the packet is indicated to TRANS as either a broadcast packet (LINK4_{BRec}), or as a packet that was addressed to this node (LINK4_{DRec}). In both cases the data checksum is verified. Observe that in

the broadcast-case, a packet with invalid data checksum is ignored. In the other case, the packet will have to be acknowledged, so upon a 'Phy Arbitration confirmation' of `won`, Link evolves into 'send acknowledge mode' (Link5).

Any deviation of the above described procedure will cause Link to ignore the packet; it will wait for a subaction gap to return to the initial state (LinkWsa). However, an `immediate` arbitration request will always be followed by a 'Phy Arbitration confirmation' of `won`. In such a case, Link is granted access to Cable, although it does not need to send an acknowledgement. Therefore, if the destination signal indicated that the packet was meant for this Link, the arbitration confirmation must be received and Cable-control must be terminated immediately by sending an `end` signal (LinkWsa).

Send Acknowledge Mode While Link is waiting for Trans to respond to a data indication with the proper acknowledgement code, it must keep Cable 'busy' by sending a `Prefix` signal on every clock indication; this is to avoid the occurrence of a `subactgap`. Depending on the type of the received packet, Trans may need to issue a so-called 'concatenated response' (for instance, the packet was a read request and Trans immediately wants to send the requested data to the requesting node). By means of a data response ($\mathsf{LDreq}(id, dest, h, d)$) Trans 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 acknowledge (*Link*6), Link may release Cable and go to its initial state. The latter means that a concatenated response is requested and that Link should hold Cable after sending the acknowledgement packet by responding to clock indications by `Prefix` signals (Link7). Now, Link already has control over Cable, so upon a data request it may evolve into 'send mode' ($\text{Link2}_{\text{resp}}$) immediately.

3. The Auxiliary Bus Protocol

In order to simulate the interaction of n Link Layers, we have also specified the external the behaviour of n Phys, connected by Cable (see appendix C) in the process Bus.

3.1 Tables over $\mathbb{B}$

As an auxiliary data type we need a sort $\mathbb{B}$table of tables over booleans. It is constructed from $\emptyset$ and $\mathtt{btable}(n, b, B)$, where b from $\mathbb{B}$, n from $\mathbb{N}$ and B from $\mathbb{B}$table. For example,

$$\mathtt{btable}(\mathrm{succ}(0), \mathsf{t}, \mathtt{btable}(0, \mathsf{f}, \emptyset))$$

denotes a table over booleans with 3 entries, numbered 0 through succ(succ(0)), and with entries 0 and succ(succ(0)) set to `t` and succ(0) set to `f`.

Additionally, a function init(n) that creates a table with n entries (number 0 through $n-1$), all initalized to `f`; a function get(n, B) yields the nth value of B; a function invert(n, B) that updates the nth value of B to not(get(n, B)); a function if(b, B_1, B_2) that chooses between B_1 and B_2, and functions zero(B), one(B), and more(B) that return `t` if either *none of the entries*, *precisely one entry*, or *more than one* entry (respectively) of B has/have the value `t`, and `f` otherwise.

Most notably, this sort is used to store information about the specific nodes connected to Bus.

3.2 Bus *Operation*

Initially, Bus is idle (BusIdle); it knows the number of Link processes that should communicate with it (n), and administrates in a $\mathbb{B}$table which Links have had control during a 'fairness interval' (t).

During each fairness interval, each Link process is allowed to gain control over Bus by means of a `fair` arbitration request at most once; it may access Bus more than once as a consequence of an `immediate` arbitration request. As soon as Bus has been idle for some specific amount of time and some Link has had access during the running fairness interval (not(zero(t))), an 'arbitration reset gap' (arbresgap) occurs, to indicate that every Link may again be granted access by `fair` arbitration.

The time interval that BUS must be idle before such an arbresgap may occur should be longer than that of a subactgap.

As soon as some LINK requests arbitration (rPAreq($id, astat$)), BUS enters 'decision mode': it checks in its table whether the requesting node has or has not had access during the present fairness interval; if not, then BUSconfirms by won and evolves into a 'busy state' (BUSBUSY); otherwise, BUS just responds lost (DECIDEIDLE).

In a busy state, it also has to be administrated which nodes have requested immediate arbitration (*next*), and which node has control over BUS (*busy*). The fourth parameter keeps track of which nodes have received a bad destination field, and is only present to be able to simulate loss and corruption of data (see the discussion of the DISTRIBUTE process).

Being in a busy state, BUS may still receive fair arbitration requests, but they will be confirmed by lost. A potential candidate for a response on the packet that is put on BUS must make itself known by an immediate arbitration request, this will be recorded in *next*. No confirmation is sent, however, until the 'busy node' releases its control.

Furthermore, as long as $busy < n$ some LINK process is still wanting to send signals, so the apropriate clock indications must be generated (sPCind($busy$)), and signals must be distributed (DISTRIBUTE).

The DISTRIBUTE process recurses over all nodes (as long as lt(i, n)) delivering the requested signal (except if $eq(i, busy)$). However, to obtain a realistic simulation, we also modelled the potential loss or corruption of signals. We have added a function corrupt to $\mathbb{S}$ which changes the checksum field of header signals, data signals, and acknowledge signals to $\perp$. Moreover, an extra dummy is used to simulate the situation in which packets with a wrong length are delivered.

- If the signal is a destination signal, then this signal may get invalid in BUS (second summand of DISTRIBUTE). However, if this happens, then the header checksum (which comes with the next signal) is no longer valid. Therefore, it is recorded in the table *destfault* to which nodes invalid destinations have been distributed.

- Any signal, except for header signals which should have a corrupted checksum according to the above, may be delivered correctly (the first summand of DISTRIBUTE).

- If the signal to be delivered is either a header signal, a data signal or an acknowledge signal, then it may be delivered corrupted (third summand) or it may not be delivered at all (fourth summand).

- 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.

After DISTRIBUTE has distributed the signal to every node, it checks whether the distributed signal was a terminating end. Namely, if it is then the present 'busy node' no longer requires access to BUS and BUSBUSY is called with n, instead of *busy*. It should be tested whether there is some node that has requested immediate arbitration request by examining *next*. If this is not the case then a subactgap is distributed to all nodes (SUBACTIONGAP) and BUS returns to idle (BUSIDLE). However, if some of the entries of *next* have value t, then control over the BUS must go to some other node (RESOLVE).

The RESOLVE process sends arbitration confirmations (won) and a clock indication to all nodes that requested immediate arbitration. As long as there is more than one node (more(*Next*)) having control over BUS there is a conflict situation. Terminating end signals must be received from LINK processes (RESOLVE2), until there is only one LINK having access to BUS. Then, a data request is received from this node and if it is not an end signal, this signal is distributed to all other nodes; this particular node becomes the 'busy node'. However, if the received signal is indeed an end signal, then no LINK has control over the BUS anymore, so a subactgap is distributed to all LINK processes, after which BUS becomes idle again (BUSIDLE).

4. Correctness Properties

In this section we formulate some properties we expect to hold for our specifications of Link and Bus (see also appendix D).

We will denote by P1394(n) the process consisting of n Link Layers numbered 0 through $n-1$ running in parallel with the specification of Bus, assuming that the 'send actions' $\mathsf{sPDind}, \mathsf{sPDreq}, \ldots$ can only happen when they communicate with their corresponding 'read actions' $\mathsf{rPDind}, \mathsf{rPDreq}, \ldots$ and that only the services provided by the Link Layers to the respective Transaction Layers are visible. That is, if we assume that I and H represent the following sets of actions:

$$I \stackrel{\text{def.}}{=} \left\{ \begin{array}{c} \mathsf{PDind}, \mathsf{PDreq}, \mathsf{PAcon}, \mathsf{PAreq}, \\ \mathsf{PCind}, \mathsf{arbresgap}, \mathsf{losesignal} \end{array} \right\}$$

$$H \stackrel{\text{def.}}{=} \left\{ \begin{array}{c} \mathsf{rPDind}, \mathsf{sPDind}, \mathsf{rPDreq}, \mathsf{sPDreq}, \mathsf{rPAcon}, \\ \mathsf{sPAcon}, \mathsf{rPAreq}, \mathsf{sPAreq}, \mathsf{rPCind}, \mathsf{sPCind} \end{array} \right\}$$

then P1394(n) $\stackrel{\text{def.}}{=} \tau_I \circ \partial_H(\text{Bus}\,(n) \| \text{Link}\,(n,0))$.

The following requirements should hold in our specification for every $n \geq 1$:

- "P1394(n) is deadlock free". That is, it has no trace that ends in a deadlock. Of course, as soon as one of the Links exhibits a deadlock, then P1394(n) will also eventually enter a deadlock situation, since `subactgap`s can no longer be communicated to this process. In real implementations such deadlocks can be resolved by the node controllers: they can reset every layer.
- "Between the occurrence of two 'subaction gaps' at most two asynchronous packets have travelled over the Bus". This can be verified by checking that in any trace of $\partial_H(\text{Bus}\,(n) \| \text{Link}\,(n,0))$ at most two LDcon actions take place.
- "If a LDreq action at node $0 < i < n$ occurred and node i communicates a PAreq each time it receives a `subactgap` signal (and before an arbresgap occurs), it also eventually does a LDcon". Since timing is absent in our specification we need the extra proviso that the PAreq occurs before the arbresgap. In real implementations timers must make sure that this is the case.
- "Every PAreq with parameter `immediate` is followed by a a matching PAcon with parameter `won`" in any trace of $\partial_H(\text{Nodes}(n))$".
- "Between two arbitration reset gaps no node receives a PAcon with parameter `won` upon a PAreq with parameter `fair` more than once."

A. Link Data

sort $\mathbb{B}$
func $\mathsf{t}, \mathsf{f} : \rightarrow \mathbb{B}$

func $\mathrm{and} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

var $b : \mathbb{B}$
rew $\mathrm{and}(\mathsf{t},b) = b$
$\mathrm{and}(b,\mathsf{t}) = b$
$\mathrm{and}(b,\mathsf{f}) = \mathsf{f}$
$\mathrm{and}(\mathsf{f},b) = \mathsf{f}$

func $\mathrm{or} : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

var $b : \mathbb{B}$
rew $\mathrm{or}(\mathsf{t},b) = \mathsf{t}$
$\mathrm{or}(b,\mathsf{t}) = \mathsf{t}$
$\mathrm{or}(b,\mathsf{f}) = b$
$\mathrm{or}(\mathsf{f},b) = b$

func $\mathrm{not} : \mathbb{B} \rightarrow \mathbb{B}$
$\mathrm{if} : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

var $b1, b2 : \mathbb{B}$
rew $\mathrm{not}(\mathsf{f}) = \mathsf{t}$
$\mathrm{not}(\mathsf{t}) = \mathsf{f}$
$\mathrm{if}(\mathsf{t},b1,b2) = b1$
$\mathrm{if}(\mathsf{f},b1,b2) = b2$

sort $\mathbb{N}$
func $0 : \rightarrow \mathbb{N}$
$\mathrm{succ} : \mathbb{N} \rightarrow \mathbb{N}$

func $\mathrm{eq} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
var $n, m : \mathbb{N}$
rew $\mathrm{eq}(0,0) = \mathsf{t}$
$\mathrm{eq}(\mathrm{succ}(n),0) = \mathsf{f}$
$\mathrm{eq}(0,\mathrm{succ}(n)) = \mathsf{f}$
$\mathrm{eq}(\mathrm{succ}(n),\mathrm{succ}(m)) = \mathrm{eq}(n,m)$

func $\mathrm{lt} : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{B}$
var $n, m : \mathbb{N}$
rew $\mathrm{lt}(0,0) = \mathsf{f}$
$\mathrm{lt}(\mathrm{succ}(n),0) = \mathsf{f}$
$\mathrm{lt}(0,\mathrm{succ}(n)) = \mathsf{t}$
$\mathrm{lt}(\mathrm{succ}(n),\mathrm{succ}(m)) = \mathrm{lt}(n,m)$

sort DATA
func $\mathrm{d1}, \mathrm{d2} : \rightarrow \mathrm{DATA}$
$\mathrm{crc} : \mathrm{DATA} \rightarrow \mathrm{CHECK}$
rew $\mathrm{crc}(\mathrm{d1}) = \top$
$\mathrm{crc}(\mathrm{d2}) = \top$

sort HEADER
func $\mathrm{h1}, \mathrm{h2} : \rightarrow \mathrm{HEADER}$
$\mathrm{crc} : \mathrm{HEADER} \rightarrow \mathrm{CHECK}$
rew $\mathrm{crc}(\mathrm{h1}) = \top$
$\mathrm{crc}(\mathrm{h2}) = \top$

sort ACK
func $\mathrm{a1}, \mathrm{a2} : \rightarrow \mathrm{ACK}$
$\mathrm{crc} : \mathrm{ACK} \rightarrow \mathrm{CHECK}$
rew $\mathrm{crc}(\mathrm{a1}) = \top$
$\mathrm{crc}(\mathrm{a2}) = \top$

sort CHECK
func $\bot, \top : \rightarrow \mathrm{CHECK}$
$\mathrm{eq} : \mathrm{CHECK} \times \mathrm{CHECK} \rightarrow \mathbb{B}$
rew $\mathrm{eq}(\bot,\bot) = \mathsf{t}$
$\mathrm{eq}(\top,\top) = \mathsf{t}$
$\mathrm{eq}(\top,\bot) = \mathsf{f}$
$\mathrm{eq}(\bot,\top) = \mathsf{f}$

sort $\mathbb{S}$
func `sig`: $\mathbb{N} \rightarrow \mathbb{S}$
`sig`: $\mathrm{HEADER} \times \mathrm{CHECK} \rightarrow \mathbb{S}$
`sig`: $\mathrm{DATA} \times \mathrm{CHECK} \rightarrow \mathbb{S}$
`sig`: $\mathrm{ACK} \times \mathrm{CHECK} \rightarrow \mathbb{S}$
`start`, `end`: $\rightarrow \mathbb{S}$
`Prefix`, subactgap: $\rightarrow \mathbb{S}$
`dhead`, `dummy`: $\rightarrow \mathbb{S}$

func start?, end? : $\mathbb{S} \to \mathbb{B}$
prefix?, sagap? : $\mathbb{S} \to \mathbb{B}$
var n : $\mathbb{N}$
h : HEADER
d : DATA
a : ACK
c : CHECK

rew

start?(`start`) = t	end?(`end`) = t	prefix?(`Prefix`) = t	sagap?(subactgap) = t
start?(`end`) = f	end?(`start`) = f	prefix?(`start`) = f	sagap?(`start`) = f
start?(`Prefix`) = f	end?(`Prefix`) = f	prefix?(`end`) = f	sagap?(`end`) = f
start?(subactgap) = f	end?(subactgap) = f	prefix?(subactgap) = f	sagap?(`Prefix`) = f
start?(`dhead`) = f	end?(`dhead`) = f	prefix?(`dhead`) = f	sagap?(`dhead`) = f
start?(`dummy`) = f	end?(`dummy`) = f	prefix?(`dummy`) = f	sagap?(`dummy`) = f
start?(`sig`(n)) = f	end?(`sig`(n)) = f	prefix?(`sig`(n)) = f	sagap?(`sig`(n)) = f
start?(`sig`(h,c)) = f	end?(`sig`(h,c)) = f	prefix?(`sig`(h,c)) = f	sagap?(`sig`(h,c)) = f
start?(`sig`(d,c)) = f	end?(`sig`(d,c)) = f	prefix?(`sig`(d,c)) = f	sagap?(`sig`(d,c)) = f
start?(`sig`(a,c)) = f	end?(`sig`(a,c)) = f	prefix?(`sig`(a,c)) = f	sagap?(`sig`(a,c)) = f

func dest?, header? : $\mathbb{S} \to \mathbb{B}$
data?, ack? : $\mathbb{S} \to \mathbb{B}$
var n : $\mathbb{N}$
h : HEADER
d : DATA
a : ACK
c : CHECK

rew

dest?(`sig`(n)) = t	header?(`sig`(h,c)) = t	data?(`sig`(d,c)) = t	ack?(`sig`(a,c)) = t
dest?(`sig`(h,c)) = f	header?(`sig`(n)) = f	data?(`sig`(n)) = f	ack?(`sig`(n)) = f
dest?(`sig`(d,c)) = f	header?(`sig`(d,c)) = f	data?(`sig`(h,c)) = f	ack?(`sig`(h,c)) = f
dest?(`sig`(a,c)) = f	header?(`sig`(a,c)) = f	data?(`sig`(a,c)) = f	ack?(`sig`(d,c)) = f
dest?(`start`) = f	header?(`start`) = f	data?(`start`) = f	ack?(`start`) = f
dest?(`end`) = f	header?(`end`) = f	data?(`end`) = f	ack?(`end`) = f
dest?(`Prefix`) = f	header?(`Prefix`) = f	data?(`Prefix`) = f	ack?(`Prefix`) = f
dest?(subactgap) = f	header?(subactgap) = f	data?(subactgap) = f	ack?(subactgap) = f
dest?(`dhead`) = f	header?(`dhead`) = f	data?(`dhead`) = f	ack?(`dhead`) = f
dest?(`dummy`) = f	header?(`dummy`) = f	data?(`dummy`) = f	ack?(`dummy`) = f

func physig?, terminator? : $\mathbb{S} \to \mathbb{B}$
var s : $\mathbb{S}$
rew physig?(s) =
or(start?(s),
or(end?(s),or(prefix?(s),sagap?(s))))
terminator?(s) = or(end?(s),prefix?(s))

func hda? : $\mathbb{S} \to \mathbb{B}$
var s : $\mathbb{S}$
rew hda?(s) =
or(header?(s),or(data?(s),ack?(s)))

func valid_hpart, valid_ack: $\mathbb{S} \to \mathbb{B}$
var n: $\mathbb{N}$
h: HEADER
d: DATA
a: ACK
c: CHECK

rew
valid_ack(sig(a,c)) = eq(c,$\top$)
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,$\top$)
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

func getdest: $\mathbb{S} \to \mathbb{N}$
getdcrc: $\mathbb{S} \to$ CHECK
getdata: $\mathbb{S} \to$ DATA
gethead: $\mathbb{S} \to$ HEADER
getack: $\mathbb{S} \to$ ACK
corrupt: $\mathbb{S} \to \mathbb{S}$
var n: $\mathbb{N}$
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,$\bot$)
corrupt(sig(d,c)) = sig(d,$\bot$)
corrupt(sig(a,c)) = sig(a,$\bot$)

sort SIG_TUPLE
func quadruple: $\mathbb{S} \times \mathbb{S} \times \mathbb{S} \times \mathbb{S} \to$ SIG_TUPLE
void: $\to$ SIG_TUPLE
π_1, π_2, π_3, π_4: SIG_TUPLE $\to \mathbb{S}$
void?: SIG_TUPLE $\to \mathbb{B}$

var *x1*, *x2*, *x3*, *x4*: $\mathbb{S}$
rew
π_1(quadruple(*x1*,*x2*,*x3*,*x4*)) = *x1*
π_2(quadruple(*x1*,*x2*,*x3*,*x4*)) = *x2*
π_3(quadruple(*x1*,*x2*,*x3*,*x4*)) = *x3*
π_4(quadruple(*x1*,*x2*,*x3*,*x4*)) = *x4*
void?(void) = t
void?(quadruple(*x1*,*x2*,*x3*,*x4*)) = f

sort PAC
func won, `lost`: → PAC
eq: PAC × PAC → 𝔹
rew eq(won,won) = t
eq(`lost`,`lost`) = t
eq(won,`lost`) = f
eq(`lost`,won) = f

sort PAR
func `fair`, `immediate`: → PAR
eq: PAR × PAR → 𝔹
rew eq(`fair`,`fair`) = t
eq(`immediate`,`immediate`) = t
eq(`fair`,`immediate`) = f
eq(`immediate`,`fair`) = 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
eq: BOC × BOC → 𝔹
rew eq(release,release) = t
eq(`hold`,`hold`) = t
eq(release,`hold`) = f
eq(`hold`,release) = f

B. Link Layer

act LDreq: ℕ × ℕ × HEADER × DATA
LDcon: ℕ × LDC
LDind: ℕ × LDI
LDres: ℕ × ACK × BOC
sPDreq, rPDind: ℕ × 𝕊
sPAreq: ℕ × PAR
rPAcon: ℕ × PAC
rPCind: ℕ

proc Link0(n:ℕ, id:ℕ, *buffer*:SIG_TUPLE) =
$(\sum(dest{:}\mathbb{N},$
$\sum(h{:}\text{HEADER},$
$\sum(d{:}\text{DATA},$
LDreq(id,$dest$,h,d) ·
Link0(n,id,`quadruple`(`dhead`,`sig`($dest$),`sig`(h,crc(h)),`sig`(d,crc(d)))))))
◁ void?(*buffer*) ▷ sPAreq(id,`fair`) · Link1(n,id,*buffer*))
$+ \sum(p{:}\mathbb{S},$ rPDind(id,p) · (Link4(n,id,*buffer*) ◁ start?(p) ▷ Link0(n,id,*buffer*)))

Link1(n:ℕ, id:ℕ, p:SIG_TUPLE) =
rPAcon(id,won) · $\text{Link2}_{\text{req}}$($n$,$id$,$p$) + rPAcon($id$,`lost`) · Link0($n$,$id$,$p$)

$\text{Link2}_{\text{req}}$($n$:ℕ, id:ℕ, p:SIG_TUPLE) =
(rPCind(id) ·
sPDreq(id,`start`) · rPCind(id) · sPDreq(id,$\pi_1(p)$) · rPCind(id) · sPDreq(id,$\pi_2(p)$)) ·
(rPCind(id) ·
sPDreq(id,$\pi_3(p)$) · rPCind(id) · sPDreq(id,$\pi_4(p)$) · rPCind(id) · sPDreq(id,`end`)) ·
(LDcon(id,`broadsent`) · Link0(n,id,`void`) ◁ eq(getdest($\pi_2(p)$),n) ▷ Link3(n,id,`void`))

$\textsc{Link3}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\sum(p{:}\mathbb{S},$
$\mathsf{rPDind}(id,p)\cdot$
$(\textsc{Link3}(n,id,buffer) \lhd \text{prefix?}(p) \rhd$
$(\textsc{Link3}_{\text{RA}}(n,id,buffer) \lhd \text{start?}(p) \rhd$
$(\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{Link0}(n,id,buffer) \lhd \text{sagap?}(p) \rhd$
$\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{LinkWSA}(n,id,buffer,n)))))$

$\textsc{Link3}_{\text{RA}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\sum(a{:}\mathbb{S},$
$\mathsf{rPDind}(id,a)\cdot$
$((\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{Link0}(n,id,buffer) \lhd \text{sagap?}(a) \rhd$
$\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{LinkWSA}(n,id,buffer,n))$
$\lhd \text{physig?}(a) \rhd \textsc{Link3}_{\text{RE}}(n,id,buffer,a)))$

$\textsc{Link3}_{\text{RE}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ a{:}\mathbb{S}) =$
$\sum(e{:}\mathbb{S},$
$\mathsf{rPDind}(id,e)\cdot$
$(\mathsf{LDcon}(id,\texttt{ackrec}(\text{getack}(a)))\cdot \textsc{LinkWSA}(n,id,buffer,n)$
$\lhd \text{and}(\text{valid\_ack}(a),\text{terminator?}(e)) \rhd$
$(\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{Link0}(n,id,buffer) \lhd \text{sagap?}(e) \rhd$
$\mathsf{LDcon}(id,\texttt{ackmiss})\cdot \textsc{LinkWSA}(n,id,buffer,n))))$

$\textsc{Link4}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\sum(dh{:}\mathbb{S},$
$\mathsf{rPDind}(id,dh)\cdot$
$((\textsc{Link0}(n,id,buffer) \lhd \text{sagap?}(dh) \rhd \textsc{LinkWSA}(n,id,buffer,n)) \lhd \text{physig?}(dh) \rhd$
$\textsc{Link4}_{\text{DH}}(n,id,buffer)))$

$\textsc{Link4}_{\text{DH}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\sum(dest{:}\mathbb{S},$
$\mathsf{rPDind}(id,dest)\cdot$
$((\mathsf{sPAreq}(id,\texttt{immediate})\cdot \textsc{Link4}_{\text{RH}}(n,id,buffer,id) \lhd \text{eq}(\text{getdest}(dest),id) \rhd$
$(\textsc{Link4}_{\text{RH}}(n,id,buffer,n) \lhd \text{eq}(\text{getdest}(dest),n) \rhd \textsc{LinkWSA}(n,id,buffer,n)))$
$\lhd \text{dest?}(dest) \rhd (\textsc{Link0}(n,id,buffer) \lhd \text{sagap?}(dest) \rhd \textsc{LinkWSA}(n,id,buffer,n))))$

$\textsc{Link4}_{\text{RH}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ dest{:}\mathbb{N}) =$
$\sum(h{:}\mathbb{S},$
$\mathsf{rPDind}(id,h)\cdot$
$(\textsc{Link4}_{\text{RD}}(n,id,buffer,dest,h) \lhd \text{valid\_hpart}(h) \rhd \textsc{LinkWSA}(n,id,buffer,dest)))$

$\textsc{Link4}_{\text{RD}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ dest{:}\mathbb{N},\ h{:}\mathbb{S}) =$
$\sum(d{:}\mathbb{S},$
$\mathsf{rPDind}(id,d)\cdot(\textsc{Link4}_{\text{RE}}(n,id,buffer,dest,h,d) \lhd \text{data?}(d) \rhd \textsc{LinkWSA}(n,id,buffer,dest)))$

$\textsc{Link4}_{\text{RE}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ dest{:}\mathbb{N},\ h{:}\mathbb{S},\ d{:}\mathbb{S}) =$
$\sum(e{:}\mathbb{S},$
$\mathsf{rPDind}(id,e)\cdot$
$((\textsc{Link4}_{\text{DRec}}(n,id,buffer,h,d) \lhd \text{eq}(dest,id) \rhd \textsc{Link4}_{\text{BRec}}(n,id,buffer,h,d))$
$\lhd \text{terminator?}(e) \rhd \textsc{LinkWSA}(n,id,buffer,dest)))$

$\text{LINK4}_{\text{DRec}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ h{:}\mathbb{S},\ d{:}\mathbb{S}) =$
$\quad \mathsf{LDind}(id,\mathtt{good}(\text{gethead}(h),\text{getdata}(d))) \cdot \mathsf{rPAcon}(id,\text{won}) \cdot \text{LINK5}(n,id,buffer)$
$\quad\quad \triangleleft\ \text{eq}(\text{getdcrc}(d),\top)\ \triangleright$
$\quad\quad \mathsf{LDind}(id,\mathtt{dcrc\_err}(\text{gethead}(h))) \cdot \mathsf{rPAcon}(id,\text{won}) \cdot \text{LINK5}(n,id,buffer)$

$\text{LINK4}_{\text{BRec}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ h{:}\mathbb{S},\ d{:}\mathbb{S}) =$
$\quad \mathsf{LDind}(id,\mathtt{broadrec}(\text{gethead}(h),\text{getdata}(d))) \cdot \text{LINK0}(n,id,buffer) \triangleleft \text{eq}(\text{getdcrc}(d),\top) \triangleright$
$\quad\quad \text{LINK0}(n,id,buffer)$

$\text{LINK5}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\quad\quad \sum(a{:}\text{ACK},\ \sum(b{:}\text{BOC},\ \mathsf{LDres}(id,a,b) \cdot \text{LINK6}(n,id,buffer,\mathtt{sig}(a,\text{crc}(a)),b)))$
$\quad + \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\mathtt{Prefix}) \cdot \text{LINK5}(n,id,buffer)$

$\text{LINK6}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ p{:}\mathbb{S},\ b{:}\text{BOC}) =$
$\quad (\mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\mathtt{start}) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,p)) \cdot$
$\quad (\mathsf{rPCind}(id) \cdot$
$\quad\quad (\mathsf{sPDreq}(id,\mathtt{end}) \cdot \text{LINK0}(n,id,buffer) \triangleleft \text{eq}(b,\text{release}) \triangleright$
$\quad\quad\quad \mathsf{sPDreq}(id,\mathtt{Prefix}) \cdot \text{LINK7}(n,id,buffer)))$

$\text{LINK7}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE}) =$
$\quad\quad \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\mathtt{Prefix}) \cdot \text{LINK7}(n,id,buffer)$
$\quad + \sum(dest{:}\mathbb{N},$
$\quad\quad \sum(h{:}\text{HEADER},$
$\quad\quad\quad \sum(d{:}\text{DATA},$
$\quad\quad\quad\quad \mathsf{LDreq}(id,dest,h,d) \cdot$
$\quad\quad\quad\quad \text{LINK2}_{\text{resp}}(n,id,buffer,\mathtt{quadruple}(\mathtt{dhead},\mathtt{sig}(dest),\mathtt{sig}(h,\text{crc}(h)),\mathtt{sig}(d,\text{crc}(d))))))$

$\text{LINK2}_{\text{resp}}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ p{:}\text{SIG\_TUPLE}) =$
$\quad (\mathsf{rPCind}(id) \cdot$
$\quad\ \mathsf{sPDreq}(id,\mathtt{start}) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\pi_1(p)) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\pi_2(p))) \cdot$
$\quad (\mathsf{rPCind}(id) \cdot$
$\quad\ \mathsf{sPDreq}(id,\pi_3(p)) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\pi_4(p)) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\mathtt{end})) \cdot$
$\quad (\mathsf{LDcon}(id,\mathtt{broadsent}) \cdot \text{LINK0}(n,id,buffer) \triangleleft \text{eq}(\text{getdest}(\pi_2(p)),n) \triangleright \text{LINK3}(n,id,buffer))$

$\text{LINKWSA}(n{:}\mathbb{N},\ id{:}\mathbb{N},\ buffer{:}\text{SIG\_TUPLE},\ dest{:}\mathbb{N}) =$
$\quad\quad \sum(p{:}\mathbb{S},\ \mathsf{rPDind}(id,p) \cdot (\text{LINK0}(n,id,buffer) \triangleleft \text{sagap?}(p) \triangleright \text{LINKWSA}(n,id,buffer,dest)))$
$\quad + (\mathsf{rPAcon}(id,\text{won}) \cdot \mathsf{rPCind}(id) \cdot \mathsf{sPDreq}(id,\mathtt{end}) \cdot \text{LINK0}(n,id,buffer) \triangleleft \text{eq}(dest,id) \triangleright \delta)$

C. Bus

sort $\mathbb{B}\text{table}$
func $\emptyset$: $\to \mathbb{B}\text{table}$
$\quad \mathtt{btable}$: $\mathbb{N} \times \mathbb{B} \times \mathbb{B}\text{table} \to \mathbb{B}\text{table}$

func init: $\mathbb{N} \to \mathbb{B}$table
invert: $\mathbb{N} \times \mathbb{B}$table $\to \mathbb{B}$table
get: $\mathbb{N} \times \mathbb{B}$table $\to \mathbb{B}$
if: $\mathbb{B} \times \mathbb{B}$table $\times \mathbb{B}$table $\to \mathbb{B}$table

var $n, m: \mathbb{N}$
$b: \mathbb{B}$
$bt1, bt2: \mathbb{B}$table
rew init(0) = $\emptyset$
init(succ(n)) = btable(n,f,init(n))
invert(n,$\emptyset$) = $\emptyset$
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,$bt1$))
if(t,$bt1$,$bt2$) = $bt1$
if(f,$bt1$,$bt2$) = $bt2$

func zero, one, more: $\mathbb{B}$table $\to \mathbb{B}$

var $n: \mathbb{N}$
$bt: \mathbb{B}$table
rew zero($\emptyset$) = t
zero(btable(n,t,bt)) = f
zero(btable(n,f,bt)) = zero(bt)
one($\emptyset$) = 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: $\mathbb{N} \times$ PAR
rPDreq, sPDind: $\mathbb{N} \times \mathbb{S}$
sPAcon: $\mathbb{N} \times$ PAC
sPCind: $\mathbb{N}$
arbresgap
losesignal

proc BusIdle(n:$\mathbb{N}$, t:$\mathbb{B}$table) =
$\sum(id{:}\mathbb{N}, \sum(astat{:}\text{PAR}, \text{rPAreq}(id,astat) \cdot \text{DecideIdle}(n,t,id,astat)))$
$+$ arbresgap $\cdot$ BusIdle(n,init(n)) $\lhd$ not(zero(t)) $\rhd \delta$

DecideIdle(n:$\mathbb{N}$, t:$\mathbb{B}$table, id:$\mathbb{N}$, $astat$:PAR) =
(sPAcon(id,won) $\cdot$ BusBusy(n,invert(id,t),init(n),init(n),id)) $\lhd$ not(get(id,t)) $\rhd$
(sPAcon(id,lost) $\cdot$ BusIdle(n,t))

BusBusy(n:$\mathbb{N}$, t:$\mathbb{B}$table, $next$:$\mathbb{B}$table, $destfault$:$\mathbb{B}$table, $busy$:$\mathbb{N}$) =
((sPCind($busy$) $\cdot \sum(p{:}\mathbb{S}$, rPDreq($busy$,$p$) $\cdot$ Distribute(n,t,$next$,$destfault$,$busy$,p,0)))
$\lhd$ lt($busy$,n) $\rhd$ (SubactionGap(n,t,0) $\lhd$ zero($next$) $\rhd$ Resolve(n,t,$next$,0)))
$+ \sum(j{:}\mathbb{N}$, rPAreq(j,fair) $\cdot$ sPAcon(j,lost) $\cdot$ BusBusy(n,t,$next$,$destfault$,$busy$))
$+ \sum(j{:}\mathbb{N}$,
rPAreq(j,immediate) $\cdot$
(BusBusy(n,t,invert(j,$next$),$destfault$,$busy$) $\lhd$ not(get(j,$next$)) $\rhd \delta$))

SubactionGap(n:$\mathbb{N}$, t:$\mathbb{B}$table, i:$\mathbb{N}$) =
BusIdle(n,t) $\lhd$ eq(i,n) $\rhd$ sPDind(i,subactgap) $\cdot$ SubactionGap(n,t,succ(i))

Resolve(n:$\mathbb{N}$, t:$\mathbb{B}$table, $next$:$\mathbb{B}$table, i:$\mathbb{N}$) =
(((sPAcon(i,won) $\cdot$ sPCind(i) $\cdot$ Resolve(n,t,$next$,succ(i))) $\lhd$ get(i,$next$) $\rhd$
($Resolve$(n,t,$next$,succ(i))))
$\lhd$ lt(i,n) $\rhd$ Resolve2(n,t,$next$))

$\textsc{Resolve2}(n{:}\mathbb{N},\ t{:}\mathbb{B}\text{table},\ next{:}\mathbb{B}\text{table}) =$
$(\sum(j{:}\mathbb{N},\ \mathsf{rPDreq}(j,\mathtt{end}) \cdot (\textsc{Resolve2}(n,t,\text{invert}(j,next)) \lhd \text{get}(j,next) \rhd \delta)) \lhd \text{more}(next) \rhd$
$\sum(j{:}\mathbb{N},$
$\sum(p{:}\mathbb{S},$
$\mathsf{rPDreq}(j,p) \cdot$
$(\textsc{SubactionGap}(n,t,0) \lhd \text{end?}(p) \rhd \textsc{Distribute}(n,t,\text{init}(n),\text{init}(n),j,p,0)))))$

$\textsc{Distribute}(n{:}\mathbb{N},\ t{:}\mathbb{B}\text{table},\ next{:}\mathbb{B}\text{table},\ destfault{:}\mathbb{B}\text{table},\ busy{:}\mathbb{N},\ p{:}\mathbb{S},\ i{:}\mathbb{N}) =$
$(((\quad (\mathsf{sPDind}(i,p) \cdot \textsc{Distribute}(n,t,next,destfault,busy,p,\text{succ}(i))$
$\lhd \text{or}(\text{not}(\text{header?}(p)),\text{not}(\text{get}(i,destfault))) \rhd \delta)$
$+ (\sum(dest{:}\mathbb{N},$
$\mathsf{sPDind}(i,\mathtt{sig}(dest)) \cdot \textsc{Distribute}(n,t,next,\text{invert}(i,destfault),busy,p,\text{succ}(i)))$
$\lhd \text{dest?}(p) \rhd \delta)$
$+ (\mathsf{sPDind}(i,\text{corrupt}(p)) \cdot \textsc{Distribute}(n,t,next,destfault,busy,p,\text{succ}(i)) \lhd \text{hda?}(p) \rhd \delta)$
$+ (\mathsf{losesignal} \cdot \textsc{Distribute}(n,t,next,destfault,busy,p,\text{succ}(i)) \lhd \text{hda?}(p) \rhd \delta)$
$+ (\mathsf{sPDind}(i,p) \cdot \mathsf{sPDind}(i,\mathtt{dummy}) \cdot \textsc{Distribute}(n,t,next,destfault,busy,p,\text{succ}(i))$
$\lhd \text{data?}(p) \rhd \delta)$
$+ (\mathsf{rPAreq}(i,\mathtt{immediate}) \cdot$
$(\textsc{Distribute}(n,t,\text{invert}(i,next),destfault,busy,p,i) \lhd \text{not}(\text{get}(i,next)) \rhd \delta)))$
$\lhd \text{not}(\text{eq}(i,busy)) \rhd \textsc{Distribute}(n,t,next,destfault,busy,p,\text{succ}(i)))$
$\lhd \text{lt}(i,n) \rhd$
$(\textsc{BusBusy}(n,t,next,destfault,n) \lhd \text{end?}(p) \rhd \textsc{BusBusy}(n,t,next,destfault,busy)))$

D. P1394

act $\mathsf{PDind}, \mathsf{PDreq} : \mathbb{N} \times \mathbb{S}$
$\mathsf{PAcon}: \quad \mathbb{N} \times \text{PAC}$
$\mathsf{PAreq}: \quad \mathbb{N} \times \text{PAR}$
$\mathsf{PCind}: \quad \mathbb{N}$

comm $\mathsf{rPDind} \mid \mathsf{sPDind} = \mathsf{PDind}$
$\mathsf{rPDreq} \mid \mathsf{sPDreq} = \mathsf{PDreq}$
$\mathsf{rPAcon} \mid \mathsf{sPAcon} = \mathsf{PAcon}$
$\mathsf{rPAreq} \mid \mathsf{sPAreq} = \mathsf{PAreq}$
$\mathsf{rPCind} \mid \mathsf{sPCind} = \mathsf{PCind}$

proc $\textsc{Link}(n{:}\mathbb{N},\ i{:}\mathbb{N}) = (\textsc{Link0}(n,i,\mathtt{void}) \parallel \textsc{Link}(n,\text{succ}(i))) \lhd \text{lt}(\text{succ}(i),n) \rhd \textsc{Link0}(n,i,\mathtt{void})$

$\textsc{Bus}(n{:}\mathbb{N}) = \textsc{BusIdle}(n,\text{init}(n))$

$\text{P1394}(n{:}\mathbb{N}) =$
$\tau(\{\mathsf{PDind},\mathsf{PDreq},\mathsf{PAcon},\mathsf{PAreq},\mathsf{PCind},\mathsf{arbresgap},\mathsf{losesignal}\},$
$\partial(\{\mathsf{rPDind},\mathsf{sPDind},\mathsf{rPDreq},\mathsf{sPDreq},\mathsf{rPAcon},\mathsf{sPAcon},\mathsf{rPAreq},\mathsf{sPAreq},\mathsf{rPCind},\mathsf{sPCind}\},$
$\textsc{Bus}(n) \parallel \textsc{Link}(n,0)))$

Acknowledgements
Thanks to Hubert Garavel, Jan Friso Groote and Mihaela Sighireanu for a number of comments that led to improvements and simplifications of both the specification and its explanation. I would also like to thank Judi Romijn for useful discussions on the IEEE Standard.

References
[BW90] J.C.M. Baeten and W.P. Weijland. *Process Algebra.* Number 18 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[GP94] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, Workshops in Computing, pages 26–62, Utrecht, The Netherlands, 1994. Springer-Verlag.

[IEE95] IEEE. P1394 Standard for a high performance serial bus, 1995. Draft 8.0v2, July 7, 1995.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs, 1989.