Towards Mechanical Verification of Parts of the IEEE P1394 Serial Bus*

Lars Kühne† Jozef Hooman‡ Willem-Paul de Roever§

Abstract

The IEEE P1394 Serial Bus standard provides high performance connections for data transfer between hardware components and is especially well suited for connecting multimedia devices. To achieve its mechanical verification, a high level specification is developed for the asynchronous part of the P1394 Link layer, using the verification tool PVS. We derive a formal framework which closely resembles the state machine approach used in the standard document. In this framework, a new parallel combinator characterizes synchronous message passing between transitions for which a set of messages is exchanged atomically. The combinator is mechanically checked for being commutative and associative. As expected, unclarities, ambiguities and unforseen properties need to be resolved in the specification of the Link layer. To date, important Link Layer properties have been handproved using linear time temporal logic; the next aim of the authors is the mechanization of these proofs in PVS.

1 Introduction

The IEEE P1394 Serial Bus [3], also known as "Firewire", is a standard for high performance connections between (computer) components. The P1394 protocol is organized in three layers, the Transaction layer, the Link layer and the Physical layer. It supports two basic data transfer services: asynchronous data transfer and isochronous data transfer. The isochronous service guarantees a minimum bandwidth for data transmission, which makes the IEEE P1394 Serial Bus especially well suited for multimedia applications.

The mechanical verification of a realistic protocol such as IEEE P1394 is one of the major challenges in the application of formal methods. As a case study, based on a proposal by Dr. Ron Koymans from Philips Research Lab., our aim is the mechanical verification of properties of the asynchronous part of the Link layer. This paper describes the first step in this direction, namely the formalization of the Link layer protocol. To achieve mechanical support, the formalization is given in the specification language of the Prototype Verification System (PVS) [7, 8].

PVS[1] is a specification language integrated with support tools and a theorem prover. The specification language of PVS is a higher-order typed logic with many predefined types, including booleans, integers, sets, sequences, etc. New types may be constructed using, for example, functions, tuples, records, predicate subtypes, or abstract datatypes, such as binary trees. The interactive theorem prover contains powerful commands ("strategies" in PVS terminology) for, e.g., inductive reasoning, automatic rewriting and the use of decision procedures for propositional and arithmetical reasoning.

In section 2 we briefly summarize the protocol architecture and the interface of the Link layer. In [3] the Link layer operation is specified in a non-standard and informal state machine approach. We point out the general problems that arise when trying to use these state machines as a basis for formal verification. These problems are solved in section 3 by providing a framework for Communicating Transition Systems which formalizes the state machine notation in [3]. In this framework a new parallel combinator is derived such that that parallel processes are able to atomically synchronize on a set of messages instead of a single one. To increase confidence in the definition it is mechanically checked that the parallel combinator is commutative and associative. In section 4 the framework is instantiated with a PVS translation of the original Link layer specification where we abstract from time, transmission failures on the bus media and consider sending a data packet to be an

*This work has been supported by the Ministerium fuer Bildung, Wissenschaft, Forschung und Kultur of Schleswig-Holstein under grant X 200 b, 0621.415-96, Hochschulsonderprogramm II

†Christian Albrechts University, Institut für Informatik, Preusserstraße 1-9, 24105 Kiel, Germany, lku@informatik.uni-kiel.de

‡Eindhoven University of Technology, P.O. Box 513, NL-5600 MB Eindhoven, The Netherlands, wsinjh@win.tue.nl

§Christian Albrechts University, Institut für Informatik, Preusserstraße 1-9, 24105 Kiel, Germany, wpr@informatik.uni-kiel.de

[1]PVS is freely available, see HTTP://www.csl.sri.com/sri-csl-pvs.html

atomic action. Section 5 contains a summary of our experiences and unsolved problems with the specification. We use linear time temporal logic to state the main hand proved properties of the protocol; its response properties are weaker than expected.

2 The Protocol

2.1 Protocol architecture

The P1394 serial bus protocol is organized in three layers. The communication between higher layers of different nodes is implemented by using services of the lower layers. For the case of three nodes this is illustrated in figure 1.

2.2 Link Layer Interface

The Link layer provides an acknowledged datagram service (a one way data transfer with confirmation of request) to the Transaction layer [3, p. 22]. It provides four services to the Transaction layer:

- The "Link Data request" (LK_DATA.req) service is used to initiate transmission of data either to a single dedicated receiver or to all nodes attached to the bus (broadcast transmission).
- The "Link Data indication" (LK_DATA.ind) service is used to inform the Transaction layer of the arrival of data.
- In case of a non broadcast data transmission the reception of data must be acknowledged by the receiving Transaction layer using a "Link Data confirmation" (LK_DATA.conf).
- With the "Link Data response" (LK_DATA.-resp) service a successful data transmission is confirmed to the originating Transaction layer.

In turn, the Link layer uses four services of the Physical layer:

- Before sending data packets, the Link layer must ask for access to the bus by using the "Physical Arbitration request" (PH_ARB.req) service.
- The Physical layer responds to a PH_ARB.req by communicating the arbitration result (either WON or LOST) with the "Physical Arbitration confirmation" (PH_ARB.conf) service.
- The Link layer instructs the Physical layer to put data on the bus media by using the "Physical Data request" (PH_DATA.req) service.
- The Link layer is informed of data on the bus media with the "Physical Data indication" (PH_-DATA.ind) service. All data on the bus are indicated to the Link layers of all nodes using this service. It is the responsibility of each Link layer to forward only the data intended for the particular node to its Transaction layer.

Note that the communication between layers can be regarded as synchronous message passing between pairs of layers.

2.3 Specification Starting Point

The P1394 standard utilizes three levels of description for operations: State machines, C++ code fragments and natural language (English text), in this order of priority. The Link layer operation is specified in the state machine depicted in [3, Figure 6.19]. For the reader's convenience, the part relevant for this work is reproduced in figure 2.

State machines are a well-known tool in formal methods, so at first sight the part of the state machine that specifies asynchronous operation (States L0 to L7) seems to be an excellent starting point for our verification efforts. There are several problems, though, connected with the use of state machines in the Firewire standard as a basis for a formal verification task:

- Actions are possible during transitions *and* in discrete states, e.g. in L2. This is in contrast to most other widely accepted formalisms related to communicating state machines.
- The description of actions is not included in the state machines but given in natural language and therefore not precise.
- The proof of any nontrivial property of the link layer will necessarily have to rely on the communication behavior of its environment. Thus, parallel execution of state machines must be considered. But in [3] the semantics of parallel execution of state machines is not described. In particular the underlying communication mechanism is neither mentioned nor explained.

For formal verification it is necessary to translate the original state machine into a formal framework where these points are clarified. To keep the probability of translation errors small, it is desirable to use a framework which closely resembles the state machines of [3].

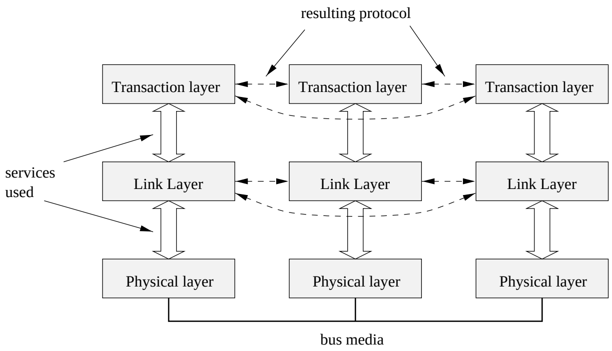

Figure 1: Protocol architecture for three nodes

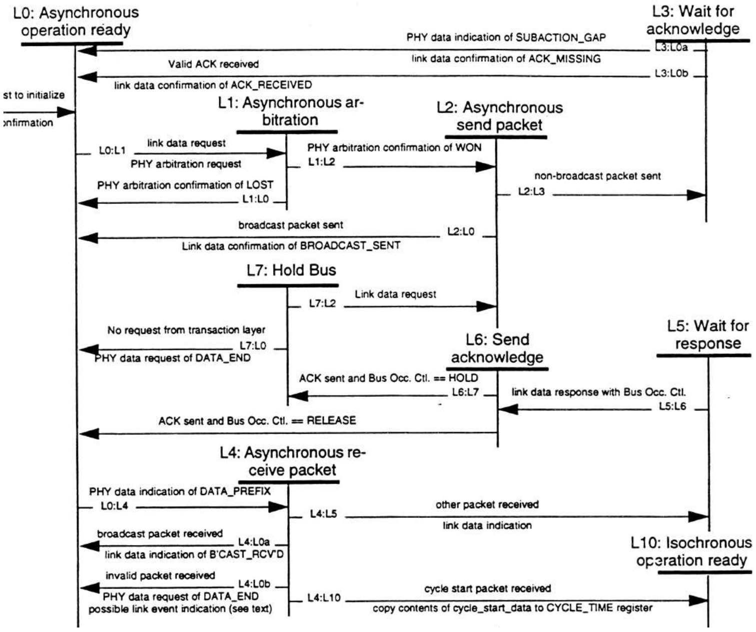

Figure 2: The asynchronous part of the Link layer state machine in [3, Figure 6.19]

3 Formal Framework

The development of the formal framework for the specification of the Link layer has been led by the following design goals:

- A Link layer formalization close to the original description should be possible.
- The framework should be similar to other well known formalisms.
- The framework should be flexible enough to be useful for other applications.
- The framework must facilitate mechanically checked proofs.

A natural choice with respect to the first three design goals is to model the Link layer as a labelled transition system. Systems working in parallel have no shared variables, they communicate by means of synchronous message passing. The label of a transition carries the set of messages received and the set of messages sent when the transition is taken. Messages have a direction, there can only be one sending and only one receiving process for each message (point to point communication).

Parallel processes communicating by means of synchronous message passing are well understood [6, 1, 2]. The distinguishing feature of our framework is that processes must synchronize on a *set* of messages in a single step instead of only a single message. Note that this extension is necessary to achieve the first design goal, since e.g. on the transition L0 $\rightarrow$ L1 more than one message occurs.

We call our systems "Communicating Transition Systems" (CTS). In this section we will formally define syntax and compositional semantics of a CTS in the specification language of PVS, to enable mechanical support.

3.1 Example

To give an informal introduction, a very simple CTS is presented in figure 3. The two processes P1 and P2 should synchronize on the messages a and b. Additionally, P2 must synchronize on the message c with the environment of P1 || P2. Instead of using a set notation as transition labels, here each message is augmented with "!" or "?" to indicate that the message is respectively sent or received during the transition.

Suppose P1 is in state A and P2 in state C. If the environment is never ready to receive the message c, then P1 || P2 can never take any transition. On the other hand, if the environment is ready to receive c, then P1 || P2 can take the transitions leading to a system state where P1 is in B and P2 is in C. From now on P2 can only take the ?a transition, since we have point to point messages and thus the b communication of P2 only synchronizes with P1.

3.2 Syntax

For types of *State*s and *Message*s, given as parameters of the PVS theory, the notion of a CTS will now be formalized in the specification language of PVS. A transition relation is a predicate on a quadruple which connects a start state with an end state by labels of sets of sent and received messages.

$\mathrm{TraRel}[\mathrm{State}: \textsc{type}, \mathrm{Message}: \textsc{type}]: \textsc{theory}$
$\textsc{begin}$
$\mathrm{TransitionRelation}: \textsc{type} \;=$
$\quad \mathrm{pred}[[\mathrm{State}, \mathrm{setof}[\mathrm{Message}], \mathrm{setof}[\mathrm{Message}], \mathrm{State}]]$
$\textsc{end}\ \mathrm{TraRel}$

A sequential *CTSSyntaxTree* consists of a process identifier *pid*, a set of initial states *init* and a transition relation τ. A parallel *CTSSyntaxTree* consists of its left and right subtree. Note that this definition resembles the use of a BNF in the definition of programming languages, a member of the *CTSSyntaxTree* type corresponds to a parsed program text.

$\mathrm{CTSSyntaxTree}[\mathrm{State}: \textsc{type}, \mathrm{Message}: \textsc{type}]: \textsc{datatype}$
$\textsc{begin}$
$\textsc{importing}\ \mathrm{TraRel}[\mathrm{State}, \mathrm{Message}]$
$\mathrm{seq}(\mathrm{pid}: \mathbb{N},$
$\quad \mathrm{init}: \mathrm{setof}[\mathrm{State}],$
$\quad \tau: \mathrm{TransitionRelation}): \mathrm{seq?}$
$||(\mathrm{left}: \mathrm{CTSSyntaxTree},$
$\quad \mathrm{right}: \mathrm{CTSSyntaxTree}): \mathrm{par?}$
$\textsc{end}\ \mathrm{CTSSyntaxTree}$

Note that this is a data type in PVS, typechecking generates, for instance, schemas for proofs by induction and recursive definitions on the data type.

In the theory *CTSSyntax* we define the notion of a *CTS*, a syntactically correct *CTSSyntaxTree*. Similar to the definition of programming languages, the semantics will only be defined for a subset of the possible syntax trees. Here it is required that each process identifier is unique in the system and that each message is point to point. Our aim is to define a predicate *good?* on *CTSSyntaxTree* which captures exactly these requirements. To this end, first some auxiliary functions are defined.

$\mathrm{CTSSyntax}[\mathrm{State}: \textsc{type+}, \mathrm{Message}: \textsc{type+}]: \textsc{theory}$
$\textsc{begin}$
$\textsc{importing}\ \mathrm{CTSSyntaxTree}[\mathrm{State}, \mathrm{Message}]$
$\mathrm{ctsst}: \textsc{var}\ \mathrm{CTSSyntaxTree}$
$\mu, \mu_1, \mu_2: \textsc{var}\ \mathrm{Message}$
$\mathrm{m}_1, \mathrm{m}_2: \textsc{var}\ \mathrm{setof}[\mathrm{Message}]$
$\mathrm{st}_1, \mathrm{st}_2: \textsc{var}\ \mathrm{State}$

In PVS all functions are total. To ensure termination of recursive functions (i.e. totality), PVS requires the statement of a well founded measure which decreases with every recursive application of the function.

For a *CTSSyntaxTree* the function *ConsMess* calculates the set of messages consumed by recursively descending to the leafs of the syntax tree and successively building the union of the results. The abstract data type mechanism automatically generates the $\ll$ relation which decreases with the "height" of the syntax tree. This relation is well suited as a measure for *ConsMess*.

$\mathrm{ConsMess}(\mathrm{ctsst}): \textsc{recursive}\ \mathrm{setof}[\mathrm{Message}] \;=$
$\quad \textsc{cases}\ \mathrm{ctsst}\ \textsc{of}$
$\quad\quad \mathrm{seq}(\mathrm{pid}, \mathrm{init}, \tau):$
$\quad\quad\quad \{\mu \mid \exists\ \mathrm{st}_1, \mathrm{m}_1, \mathrm{m}_2, \mathrm{st}_2 : \tau(\mathrm{st}_1, \mathrm{m}_1, \mathrm{m}_2, \mathrm{st}_2) \wedge (\mu \in \mathrm{m}_1)\},$
$\quad\quad ||(\mathrm{ctsst1}, \mathrm{ctsst2}):$
$\quad\quad\quad (\mathrm{ConsMess}(\mathrm{ctsst1}) \cup \mathrm{ConsMess}(\mathrm{ctsst2}))$
$\quad \textsc{endcases}$
$\textsc{measure}\ \mathrm{ctsst}\ \textsc{by} \ll$

The definition of *ProdMess*, which determines the produced messages of a syntax tree, and the definition of *Pids* which collects the occurring process identifiers, is analogous. The *Alphabet* of a syntax tree is defined as the union of the consumed and produced messages. The PVS definitions are not presented here for brevity.

The *good?* predicate filters out those syntax trees for which we want to give a semantics.

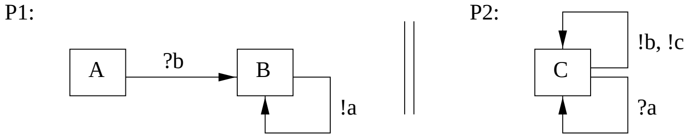

Figure 3: A simple CTS, consisting of two sequential components

good?(ctsst) : RECURSIVE $\mathbb{B}$ =
  COND
    seq?(ctsst) $\rightarrow$
      ConsMess(ctsst) $\cap$ ProdMess(ctsst) = $\emptyset$,
    par?(ctsst) $\rightarrow$
      good?(left(ctsst))$\wedge$
      good?(right(ctsst))$\wedge$
      ConsMess(left(ctsst))$\cap$ConsMess(right(ctsst)) = $\emptyset\wedge$
      ProdMess(left(ctsst))$\cap$ProdMess(right(ctsst)) = $\emptyset\wedge$
      Pids(left(ctsst))$\cap$Pids(right(ctsst)) = $\emptyset$
  ENDCOND
MEASURE ctsst BY $\ll$

CTS : TYPE = {ctsst | good?(ctsst)}

END CTSSyntax

To increase the confidence in definitions, a recommended method is to try to prove so called "putative" lemmas, that is, formulate properties that ought to hold. Here we prove, using the interactive theorem prover of PVS, that for a *CTS* the syntactic constraints which are imposed by *good?* indeed guarantee point to point communication.

Message_in_three_Alphabets_not_good : LEMMA
  good?(((ctsst1||ctsst2)||ctsst3)) $\Rightarrow$
    $\forall \mu$:
      $\neg$(Alphabet(ctsst1)(μ)$\wedge$
        Alphabet(ctsst2)(μ)$\wedge$
          Alphabet(ctsst3)(μ))

3.3 Example Revisited

The following PVS specifications formally define the CTS represented in figure 3. To instantiate the *CTS_Syntax* theory the types of states and messages must be defined. Once these types are available it is straightforward to define the transition relations for the two processes. Note that typechecking the declarations of *P1*, *P2* and *simplects* generates typecheck conditions (TCCs) which require the user to prove that indeed the predicate *good?* holds for these syntax trees[2].

[2] All three TCCs can be proved with one user interaction using the proof strategy *(grind)*.

examplects : THEORY
BEGIN
simpstate : TYPE = $\{A, B, C\}$
simpmesg : TYPE = $\{a, b, c\}$
IMPORTING CTSSyntax[simpstate, simpmesg]
$\mathrm{st}_1, \mathrm{st}_2$: VAR simpstate
cm, pm : VAR setof[simpmesg]
$\tau_1(\mathrm{st}_1, \mathrm{cm}, \mathrm{pm}, \mathrm{st}_2)$: $\mathbb{B}$ =
  $(\mathrm{st}_1 = A \wedge \mathrm{cm} = \{b\} \wedge \mathrm{pm} = \emptyset \wedge \mathrm{st}_2 = B) \vee$
  $(\mathrm{st}_1 = B \wedge \mathrm{cm} = \emptyset \wedge \mathrm{pm} = \{a\} \wedge \mathrm{st}_2 = B)$
$\tau_2(\mathrm{st}_1, \mathrm{cm}, \mathrm{pm}, \mathrm{st}_2)$: $\mathbb{B}$ =
  $(\mathrm{st}_1 = C \wedge \mathrm{cm} = \emptyset \wedge \mathrm{pm} = \{b, c\} \wedge \mathrm{st}_2 = C) \vee$
  $(\mathrm{st}_1 = C \wedge \mathrm{cm} = \{a\} \wedge \mathrm{pm} = \emptyset \wedge \mathrm{st}_2 = C)$
$P1$: CTS = seq$(1, \{A\}, \tau_1)$
$P2$: CTS = seq$(2, \{C\}, \tau_2)$
simplects : CTS = $(P1||P2)$
END examplects

3.4 Semantics

Basically we will define the semantics of a CTS as the set of sequences of the steps it can take. The type of a *Step* is defined dependent on the set of process identifiers *pidset* occurring in the CTS.

The start state component $\mathcal{S}$ of a *Step* is defined as a mapping of the process identifiers to the *State* of the component (every process gets its own "address space"). For example, in each step of process P2 $\mathcal{S}$ will be a mapping from $\{2\}$ to the current control location of P2.

A *Step* records the set of occurring messages in its $\mathcal{M}$ component.

From the example in section 3.1 it is clear that the communication behavior of the environment must be taken into account in order to define a compositional semantics. In the ready set component $\mathcal{R}$ of a *Step* we record which synchronization partners the environment could have made available to trigger additional transitions.

If $\mathcal{R}$ is nonempty, this means that there are (potentially many) components in the system which could not take any transition because the environment did not offer the right synchronization. Each member m of the Ready set represents one possibility the environment had to trigger at least one of the blocked transitions by offering all messages in m. An empty $\mathcal{R}$ set means that the environment has no chance to trigger additional transitions by offering more communication partners.

Our ready sets extend the idea of ready sets in [6] to capture synchronization on sets of messages, each element of a ready set represents a choice of the environment. Note that acceptance sets in [2], like our ready sets, are sets of sets of messages but their meaning is different - each member of an acceptance set represents an internal choice of the system.

$$
\begin{array}{l}
\mathrm{CTSSemantics[State} : \textsc{type+}, \\
\qquad\qquad \mathrm{Message} : \textsc{type+}] : \textsc{theory} \\
\textsc{begin} \\
\textsc{importing}\ \mathrm{CTSSyntax[State, Message]} \\
\mathrm{cts} : \textsc{var}\ \mathrm{CTS} \\
\mathrm{seqcts} : \textsc{var}\ \{\mathrm{cts} \mid \mathrm{seq?(cts)}\} \\
\mathrm{pid}, n : \textsc{var}\ \mathbb{N} \\
\mathrm{pidset, pidsetl, pidsetr} : \textsc{var}\ \mathrm{setof}[\mathbb{N}] \\
\mathrm{m}_c, \mathrm{m}_p, \mathrm{m}, \mathrm{m}_1, \mathrm{m}_2, \alpha : \textsc{var}\ \mathrm{setof[Message]}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{Step(pidset)} : \textsc{type} \ = \\
\quad [\#\ \mathcal{S} : [(\mathrm{pidset}) \to \mathrm{State}], \\
\qquad \mathcal{M} : \mathrm{setof[Message]}, \\
\qquad \mathcal{R} : \mathrm{setof[setof[Message]]}\ \#]
\end{array}
$$

$$
\mathrm{Run(pidset)} : \textsc{type} \ = \ \mathrm{sequence[Step(pidset)]}
$$

We start with defining the semantics of a system by defining the set of *runs* of a sequential system. The first start state $\mathcal{S}$ in such a run σ must map the process identifier to an initial state. For each step there are three possibilities:

- The start state has no successors in the transition relation (termination). Then this state is repeated in the next system step.
- The system takes a step according to the transition relation, assuming that its environment provides the necessary communication partners.
- There are successor states in the transition relation, yet the system stays in the current state, assuming that its environment does not provide any of the necessary communication counterparts. Note that in this case it should not be possible to leave the state without message passing.

$$
\begin{array}{l}
\mathrm{runs(seqcts)} : \mathrm{setof[Run(\{pid(seqcts)\})]} \ = \\
\quad \{\sigma : \mathrm{Run(\{pid(seqcts)\})} \ \mid \\
\quad\ (\mathcal{S}(\sigma(0))(\mathrm{pid(seqcts)}) \in \mathrm{init(seqcts)}) \wedge \\
\quad\ \forall (n : \mathbb{N}) : \\
\qquad \textsc{let} \\
\qquad\quad \mathrm{ssn0} \ = \ \mathcal{S}(\sigma(n))(\mathrm{pid(seqcts)}) \\
\qquad\quad \mathrm{ssn1} \ = \ \mathcal{S}(\sigma(n+1))(\mathrm{pid(seqcts)}) \\
\qquad \textsc{in} \\
\qquad (\forall\ \mathrm{m}_c, \mathrm{m}_p, \mathrm{st} : \neg\tau(\mathrm{seqcts})(\mathrm{ssn0}, \mathrm{m}_c, \mathrm{m}_p, \mathrm{st})) \wedge \\
\qquad \mathrm{ssn1} = \mathrm{ssn0} \wedge \\
\qquad \mathcal{M}(\sigma(n)) = \emptyset \wedge \\
\qquad \mathcal{R}(\sigma(n)) = \emptyset \\
\qquad \vee \\
\qquad (\exists\ \mathrm{m}_c, \mathrm{m}_p : \\
\qquad\quad \tau(\mathrm{seqcts})(\mathrm{ssn0}, \mathrm{m}_c, \mathrm{m}_p, \mathrm{ssn1}) \wedge \\
\qquad\quad \mathcal{M}(\sigma(n)) = (\mathrm{m}_c \cup \mathrm{m}_p) \wedge \\
\qquad\quad \mathcal{R}(\sigma(n)) = \emptyset) \\
\qquad \vee \\
\qquad (\exists\ \mathrm{m}_c, \mathrm{m}_p, \mathrm{st} : \tau(\mathrm{seqcts})(\mathrm{ssn0}, \mathrm{m}_c, \mathrm{m}_p, \mathrm{st})) \wedge \\
\qquad (\forall\ \mathrm{st} : \neg\tau(\mathrm{seqcts})(\mathrm{ssn0}, \emptyset, \emptyset, \mathrm{st})) \wedge \\
\qquad \mathrm{ssn1} = \mathrm{ssn0} \wedge \\
\qquad \mathcal{M}(\sigma(n)) = \emptyset \wedge \\
\qquad \mathcal{R}(\sigma(n)) = \\
\qquad\quad \{\mathrm{m} \mid \\
\qquad\qquad \exists\ \mathrm{m}_1, \mathrm{m}_2, \mathrm{st} : \\
\qquad\qquad \mathrm{m} = (\mathrm{m}_1 \cup \mathrm{m}_2) \wedge \\
\qquad\qquad \tau(\mathrm{seqcts})(\mathrm{ssn0}, \mathrm{m}_1, \mathrm{m}_2, \mathrm{st})\}\ \}
\end{array}
$$

Consider as an example the *runs* of process P2. From the type information we see that its *runs* are a set of infinite sequences of *Steps*, where the $\mathcal{S}$ component of a step is a mapping from {2} to {A, B, C}. In the initial step, 2 must be mapped to C, since {C} is the set of initial states of P2. As there is a successor of C in the transition relation of P2, there are three possibilities for the first step of a run of P2:

- $\mathcal{M}$={b,c} and $\mathcal{R}=\emptyset$, corresponding to taking the upper transition.
- $\mathcal{M}$={a} and $\mathcal{R}=\emptyset$, corresponding to taking the lower transition.
- $\mathcal{M}=\emptyset$ and $\mathcal{R}$={{b,c},{a}}, corresponding to taking no transition. In the $\mathcal{R}$ set we record that the environment neither does communicate a nor b and c together.

Similarly, in the first step of a run of P1, the start state always maps 1 to A, as {A} is the set of initial states of P1. For the $\mathcal{M}$ and $\mathcal{R}$ components there are two possibilities:

- $\mathcal{M}$={b} and $\mathcal{R}=\emptyset$, corresponding to taking the transition that exits A.
- $\mathcal{M}=\emptyset$ and $\mathcal{R}$={{b}}, corresponding to not being able to take the transition because the environment does not generate b in this step.

To abstract from time, the *stutterclosure* operation inserts arbitrary but finite delays into each member of a set of runs. The definition is rather technical and therefore omitted here for brevity. We only present the type declaration.

$\text{stutterclosure}(\text{pid}, (\text{runs}: \text{setof}[\text{Run}(\{\text{pid}\})])):$
$\quad \text{setof}[\text{Run}(\{\text{pid}\})] = \ldots$

The parallel composition of two components is based on merging the steps of two parallel components into a step of the combined component. The predicate *step_merged_from* below states the relation between steps *ls* and *rs* of the components and step *cs* of the parallel composition:

- The start state $\mathcal{S}$ of *cs* is obtained from the start states of the components for the relevant process identifiers.
- The set of messages occurring in the combined step is the union of the message sets occurring in the component steps[3]. As a side condition, the $\mathcal{M}$ components of the component steps have to coincide on their synchronization alphabet α.
- To make sure that the two components are not waiting for each other their ready sets must be disjoint.
- Each member m of the Ready set of a component step represents one or more transitions that cannot be taken. The environment of the combined system has a chance to trigger the same transitions if no messages of the synchronization alphabet are involved. If waiting occurs in both components, then additional possibilities m of the environment of the combined step to trigger additional transitions are given by taking the symmetric difference of all possibilities m_1 and m_2 to trigger transitions in the components. In this case we must make sure that m_1 and m_2 coincide on the synchronization alphabet, since otherwise one component would not be able to synchronize on the internal messages required by the other component.
- Finally we have to ensure that no set of messages that would have triggered a transition is available in the $\mathcal{M}$ component of the combined step.

$$
\begin{aligned}
&\text{step\_merged\_from}(\text{pidsetl}, \text{pidsetr}, \alpha,\\
&\qquad (\text{ls}: \text{Step}(\text{pidsetl})),\\
&\qquad (\text{rs}: \text{Step}(\text{pidsetr})),\\
&\qquad (\text{cs}: \text{Step}((\text{pidsetl} \cup \text{pidsetr})))): \mathbb{B} =\\
&\quad (\forall\ (\text{pid}: (\text{pidsetl})): \mathcal{S}(\text{cs})(\text{pid}) = \mathcal{S}(\text{ls})(\text{pid})) \wedge\\
&\quad (\forall\ (\text{pid}: (\text{pidsetr})): \mathcal{S}(\text{cs})(\text{pid}) = \mathcal{S}(\text{rs})(\text{pid})) \wedge\\
&\quad \mathcal{M}(\text{cs}) = (\mathcal{M}(\text{ls}) \cup \mathcal{M}(\text{rs})) \wedge\\
&\quad (\mathcal{M}(\text{ls}) \cap \alpha) = (\mathcal{M}(\text{rs}) \cap \alpha) \wedge\\
&\quad \mathcal{R}(\text{ls}) \cap \mathcal{R}(\text{rs}) = \emptyset \wedge\\
&\quad \mathcal{R}(\text{cs}) =\\
&\qquad \{\text{m} \mid\\
&\qquad\quad (\text{m} \in (\mathcal{R}(\text{ls}) \cup \mathcal{R}(\text{rs}))) \wedge \text{m} \cap \alpha = \emptyset \vee\\
&\qquad\quad \exists \text{m}_1, \text{m}_2:\\
&\qquad\qquad (\text{m}_1 \in \mathcal{R}(\text{ls})) \wedge\\
&\qquad\qquad (\text{m}_2 \in \mathcal{R}(\text{rs})) \wedge\\
&\qquad\qquad (\text{m}_1 \cap \alpha) = (\text{m}_2 \cap \alpha) \wedge\\
&\qquad\qquad \text{m} = (\text{m}_1 \cup \text{m}_2) \setminus (\text{m}_1 \cap \text{m}_2)\} \wedge\\
&\quad \forall \text{m}: (\text{m} \in \mathcal{R}(\text{cs})) \Rightarrow \neg(\text{m} \subseteq \mathcal{M}(\text{cs}))
\end{aligned}
$$

[3] Note that internal messages are not hidden

As an example, consider again figure 3 and the merge of steps of P1 and P2 into a combined step of P1 || P2. These two components have the synchronization alphabet {a,b}. P1 is initially in state A, so the $\mathcal{S}$ of the combined first step maps 1 to A and 2 to C. The component steps must coincide on {a,b}, i.e. if P1 takes the "?b" transition in the first step then P2 must take the "!b, !c" transition. In the case where in the first step both components are ready but not taking a transition, the ready set $\mathcal{R}$ of the combined step is {{c}}, stating that transitions in P1 || P2 were blocked because the environment could not synchronize on "c".

The semantics of a parallel system is the set of runs obtained from the point-wise merge of the runs of its components. This results in the following definition of the semantic operator $[\![\cdot]\!]$:

$$
\begin{aligned}
&[\![\text{cts}]\!]: \text{RECURSIVE setof}[\text{Run}(\text{Pids}(\text{cts}))] =\\
&\quad \text{IF seq?}(\text{cts}) \text{ THEN}\\
&\qquad \text{stutterclosure}(\text{pid}(\text{cts}), \text{runs}(\text{cts}))\\
&\quad \text{ELSE}\\
&\qquad \{\sigma_c: \text{Run}(\text{Pids}(\text{cts})) \mid\\
&\qquad\quad \exists\ (\sigma_l: \text{Run}(\text{Pids}(\text{left}(\text{cts})))),\\
&\qquad\qquad (\sigma_r: \text{Run}(\text{Pids}(\text{right}(\text{cts})))):\\
&\qquad\qquad (\sigma_l \in [\![\text{left}(\text{cts})]\!]) \wedge\\
&\qquad\qquad (\sigma_r \in [\![\text{right}(\text{cts})]\!]) \wedge\\
&\qquad\qquad \forall n:\\
&\qquad\qquad\quad \text{step\_merged\_from}(\text{Pids}(\text{left}(\text{cts})), \text{Pids}(\text{right}(\text{cts})),\\
&\qquad\qquad\qquad (\text{Alphabet}(\text{left}(\text{cts})) \cap \text{Alphabet}(\text{right}(\text{cts}))),\\
&\qquad\qquad\qquad \sigma_l(n), \sigma_r(n), \sigma_c(n))\}\\
&\quad \text{ENDIF}\\
&\text{MEASURE cts BY} <
\end{aligned}
$$

END CTSSemantics

Intuitively one expects from the semantics that it should be commutative and associative w.r.t. the parallel operator. The following lemmas show that this is indeed the case:

$$
\begin{aligned}
&\text{semantics\_commutativity}: \text{LEMMA}\\
&\quad \text{good?}((\text{cts1} || \text{cts2})) \Rightarrow\\
&\qquad [\![(\text{cts1} || \text{cts2})]\!] = [\![(\text{cts2} || \text{cts1})]\!]
\end{aligned}
$$

$$
\begin{aligned}
&\text{semantics\_associativity}: \text{LEMMA}\\
&\quad \text{good?}(((\text{cts1} || \text{cts2}) || \text{cts3})) \Rightarrow\\
&\qquad [\![(\text{cts1} || (\text{cts2} || \text{cts3}))]\!] = [\![((\text{cts1} || \text{cts2}) || \text{cts3})]\!]
\end{aligned}
$$

It is interesting to note that for a preliminary version of the semantics definition the associativity lemma did not hold. Trying to prove it isolated the problem, and led to the required correction.

4 Formalization of the Link Layer

4.1 Abstractions and Changes

Looking through the IEEE 1394 standard document, it becomes immediately clear that the description focuses very much on technical details, e.g. the exact shape of plug connectors or the bit-description of data packets sent via the serial bus. This is very useful for an implementation of the Firewire protocol and explains why in industry [3] is considered a well written document. At the same time a very detailed description aggravates any correctness argument since the wealth of information hides why the protocol basically works. We therefore abstract from the specification in [3] in the following ways, resulting in the description given in figure 4:

- We do not use a quantitative notion of time (i.e. only the order of events is considered, not the distance in time between events). The only exception is a time-out which is modeled as a message.
- Transmission failures are not considered. Hence, we need not model CRC checksum values, packet format errors, etc.
- Transmission of data via the Physical layer is performed on the level of packets instead of the level of bits. Packet delimiters such as DATA_END are integrated into the packets.

The last point has some consequences for the resulting CTS for the Link layer. In the original specification, transmission of a packet is often done in a state (e.g. in states L2, L4 or L6). In our formalism actions can only take place during execution of transitions. Transforming the state machine in [3, Figure 6.19] into our formalism makes the state L6 superfluous. Furthermore it is necessary to introduce a new location L2' from which the transmission of a data packet (originally inside L2) can be performed.

When sending a data packet, the Link layer always appends a DATA_END delimiter to the packet. In the case of sending an acknowledgment packet, the Link layer must decide whether to release the bus according to the parameter of the LK_DATA.resp parameter (HOLD or RELEASE). In case of a HOLD parameter the Transaction layer intends to do a so called concatenated response, a new data packet shall be appended directly to the acknowledgment packet without re-arbitration for the bus. In this case a DATA_END symbol must not be sent, indicating that control over the bus is not yet released. To model these possible behaviors, we introduce two functions ASSEMBLE_ACK_1 and ASSEMBLE_ACK_2 which construct an acknowledge packet from an acknowledgment code with and without a DATA_END delimiter respectively.

It is not clear from [3, Figure 6.19] how it is ensured that a Link layer does not miss incoming data packets while arbitrating in L1. To solve this problem, here an incoming data packet is identified with a LOST arbitration, as indicated in [3, p. 175].

4.2 Problems with the Formalization

In addition to the more general problems mentioned in section 2.3, the original specification in [3] is imprecise at many points in the description of the operation of the Link layer. For example the explaining text for state L4, where an asynchronous packet is received, includes [3, p. 176]:

> The link layer should communicate the PHY arbitration request (with an arbitration class of IMMEDIATE) as soon as possible after it verifies that the destination_ID in the packet header addresses this node [...], in anticipation of sending an acknowledge packet [...]. The PHY layer shall respond with a PHY arbitration confirmation of WON.

Already these two sentences raise many open questions: When is "as soon as possible"? When does the Physical layer respond with WON? Where in the Link layer state machine is the response received? *Obtaining appropriate decisions for questions like these was one of the most time consuming parts of the work presented here.*

4.3 Formal Definition of the Link layer

The formalized Link layer is illustrated in figure 4. To give the formal definition in PVS we need to define the data types of a state and a message with which the framework can be instantiated.

To define the type of a message (or service), we first define its parameter types, most prominently the type of a packet. A packet contains a destination address and some data. For N nodes on the bus the addresses 0 to N-1 are node addresses, address N is

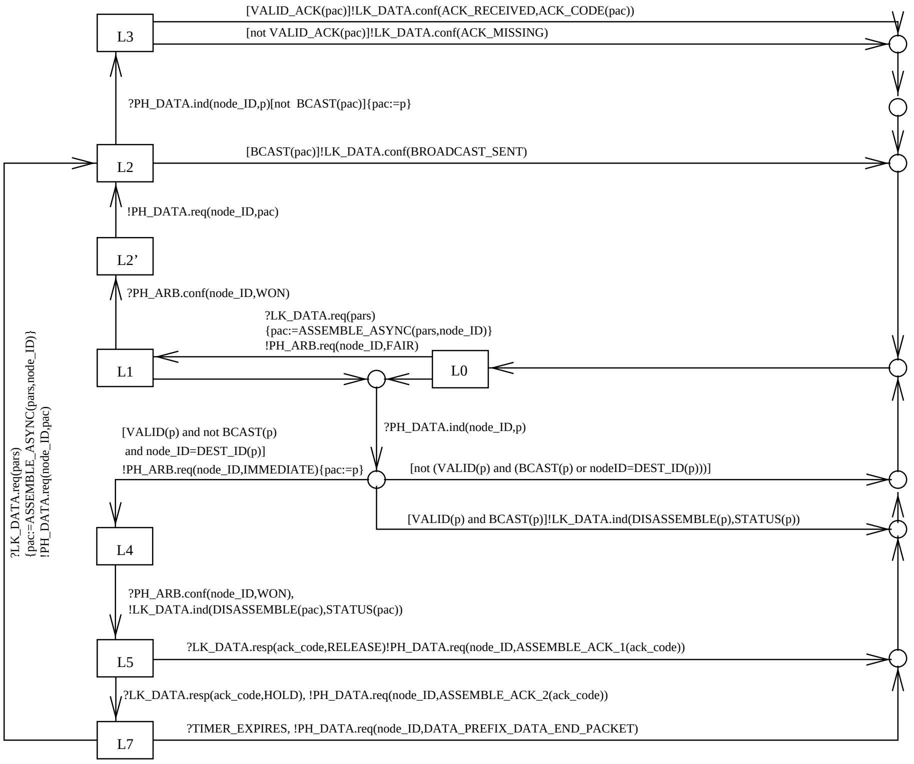

Figure 4: The Link layer formalization

used for broadcast transmissions. The representation of the packet data is not important here, therefore it is defined as an uninterpreted type which should not be empty.

Messages$[N : \mathbb{N}]$: THEORY
BEGIN
Address : TYPE $=$ upto(N)
NAddress : TYPE $= \{a :$ Address $\mid a \neq N\}$
Data : TYPE+

A packet is either an acknowledgment packet, an asynchronous data packet or a packet consisting only of data delimiters. To keep the data structure simple, only for acknowledgment packets the DATA_END symbol is modeled explicitly. The HASDATAEND predicate determines, which packets carry a DATA_END symbol. Predicates are defined to determine if a packet is a broadcast packet, a valid asynchronous packet, or a valid acknowledgment packet.

Packet : DATATYPE
BEGIN
AckPacket(ackdata : Data,
 hasdataend : $\mathbb{B}$) : ackpacket?
AsyncPacket(source : NAddress,
 dest : Address, data : Data) : asyncpacket?
DataPrefixDataEndPacket : pepacket?
END Packet

p : VAR Packet

HASDATAEND(p) : $\mathbb{B}$ $=$
 IF ackpacket?(p) THEN hasdataend(p)
 ELSE TRUE ENDIF
BCAST(p) : $\mathbb{B}$ $=$
 IF asyncpacket?(p) THEN dest$(p) = N$
 ELSE FALSE ENDIF
VALID(p) : $\mathbb{B}$ $=$ asyncpacket?(p)
VALID_ACK(p) : $\mathbb{B}$ $=$ ackpacket?(p)

Next we define the types of the other message parameters such as acknowledgment codes, Physical arbitration request classes etc.

AckCode : TYPE+
PacketStatus : TYPE $=$ {GOOD, BROADCAST}
lkdatareqpars : TYPE $=$
 [# dest : Address, data : Data#]

lkdataindpars : TYPE $=$
 [# source : NAddress,
  dest : Address,
  data : Data #]

lkdataconfpars : DATATYPE
BEGIN
 ACK_MISSING : ackmissing?
 ACK_RECEIVED(ackcode : AckCode) : ackreceived?
 BROADCAST_SENT : bcastsent?
END lkdataconfpars

$$\begin{aligned}
&\text{ArbitrationConfClass}: \textsc{type} = \{\text{WON}\}\\
&\text{ArbitrationReqClass}: \textsc{type} = \{\text{FAIR}, \text{IMMEDIATE}\}\\
&\text{BusOccCtl}: \textsc{type} = \{\text{HOLD}, \text{RELEASE}\}
\end{aligned}$$

$$\begin{aligned}
&\text{STATUS}(\text{asy}: (\text{asyncpacket?})): \text{PacketStatus} =\\
&\quad \textsc{if}\ \text{BCAST}(\text{asy})\ \textsc{then}\ \text{BROADCAST}\\
&\quad \textsc{else}\ \text{GOOD}\ \textsc{endif}
\end{aligned}$$

The functions *ASSEMBLE_ASYNC*, *ASSEMBLE_ACK_1* and *ASSEMBLE_ACK_2* are used by the Link layer to encode the information received from its Transaction layer into packets. Using the functions *DISASSEMBLE* and *ACK_CODE* the incoming packets can be decoded. The function definitions are omitted here for brevity; the following lemmas state the relation between the encoding and the decoding functions.

$$\begin{aligned}
&\text{pars}: \textsc{var}\ \text{lkdatareqpars}\\
&\text{nodeid}: \textsc{var}\ \text{NAddress}\\
&\text{ack}: \textsc{var}\ \text{AckCode}
\end{aligned}$$

$$\begin{aligned}
&\text{diassembleassemble}: \textsc{lemma}\\
&\text{DISASSEMBLE}(\text{ASSEMBLE_ASYNC}(\text{pars}, \text{nodeid})) =\\
&\quad (\#\ \text{source} := \text{nodeid},\\
&\qquad \text{dest} := \text{dest}(\text{pars}),\\
&\qquad \text{data} := \text{data}(\text{pars})\ \#)
\end{aligned}$$

$$\begin{aligned}
&\text{ackcodeassemble}: \textsc{lemma}\\
&\quad \text{ACK_CODE}(\text{ASSEMBLE_ACK_1}(\text{ack})) = \text{ack} \wedge\\
&\quad \text{ACK_CODE}(\text{ASSEMBLE_ACK_2}(\text{ack})) = \text{ack}
\end{aligned}$$

The type of a *Message* is defined as an abstract data type. To obtain point to point messages, the *nodeid* of the involved Link layer is necessary as a parameter for every message.

$$\begin{aligned}
&\text{Message}[N: \mathbb{N}]: \textsc{datatype}\\
&\quad \textsc{begin}\\
&\quad \text{PH_DATAind}(\text{nodeid}: \text{NAddress},\\
&\qquad p: \text{Packet}): \text{phdataind?}\\
&\quad \text{PH_ARBconf}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{class}: \text{ArbitrationConfClass}): \text{phdataconf?}\\
&\quad \text{PH_DATAreq}(\text{nodeid}: \text{NAddress},\\
&\qquad p: \text{Packet}): \text{phdatareq?}\\
&\quad \text{PH_ARBreq}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{class}: \text{ArbitrationReqClass}): \text{pharbreq?}\\
&\quad \text{LK_DATAreq}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{pars}: \text{lkdatareqpars}): \text{lkdatareq?}\\
&\quad \text{LK_DATAresp}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{ack_code}: \text{AckCode},\\
&\qquad \text{occ_ctl}: \text{BusOccCtl}): \text{lkdataresp?}\\
&\quad \text{LK_DATAind}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{pars}: \text{lkdataindpars},\\
&\qquad \text{status}: \text{PacketStatus}): \text{lkdataind?}\\
&\quad \text{LK_DATAconf}(\text{nodeid}: \text{NAddress},\\
&\qquad \text{pars}: \text{lkdataconfpars}): \text{lkdataconf?}\\
&\quad \text{TIMER_EXPIRES}(\text{nodeid}: \text{NAddress}): \text{timerex?}\\
&\quad \textsc{end}\ \text{Message}\quad \%\ \text{end of datatype definition}
\end{aligned}$$

$$\textsc{end}\ \text{Messages}\quad \%\ \text{end of theory}$$

The state of a Link layer CTS consists of the current control location *loc* and a local variable *pac* which records the currently processed packet.

$$\begin{aligned}
&\text{componentstate}[N: \mathbb{N}]: \textsc{theory}\\
&\textsc{begin}\\
&\textsc{importing}\ \text{Messages}[N]\\
&\text{Linklocation}: \textsc{type} =\\
&\quad \{L0, L1, L2', L2, L3, L4, L5, L7\}\\
&\text{Linkstate}: \textsc{type} =\\
&\quad [\#\ \text{loc}: \text{Linklocation}, \text{pac}: \text{Packet}\ \#]\\
&\textsc{end}\ \text{componentstate}
\end{aligned}$$

Similar to the example in section 3.3, it is now straightforward to give the PVS definition of the transition relation for the Link layer of node *nodeid* on a bus consisting of $N > 0$ nodes. Due to space limitations, only a small part of the PVS translation of figure 4 is presented here.

$$\begin{aligned}
&\text{linklayer}[N: \mathbb{N}_{>0}]: \textsc{theory}\\
&\textsc{begin}\\
&\textsc{importing}\ \text{componentstate}[N]\\
&\textsc{importing}\ \text{Messages}[N]\\
&\textsc{importing}\ \text{CTSSemantics}[\text{Linkstate}, \text{Message}]
\end{aligned}$$

$$\begin{aligned}
&\text{nodeid}: \textsc{var}\ \text{NAddress}\\
&\text{st}, \text{sst}, \text{est}: \textsc{var}\ \text{Linkstate}
\end{aligned}$$

$$\begin{aligned}
&\text{LinkTraRel}(\text{nodeid}): \text{TransitionRelation} =\\
&\{(\text{sst}, \text{m}_c, \text{m}_p, \text{est})\ |\\
&\quad (\exists\ (\text{pars}: \text{lkdatareqpars}):\\
&\qquad \text{loc}(\text{sst}) = L0 \wedge\\
&\qquad \text{m}_c = \{\text{LK_DATAreq}(\text{nodeid}, \text{pars})\} \wedge\\
&\qquad \text{m}_p = \{\text{PH_ARBreq}(\text{nodeid}, \text{FAIR})\} \wedge\\
&\qquad \text{pac}(\text{est}) = \text{ASSEMBLE_ASYNC}(\text{pars}, \text{nodeid}) \wedge\\
&\qquad \text{loc}(\text{est}) = L1) \vee\\
&\quad (\exists\ (p: \text{Packet}):\\
&\qquad \text{loc}(\text{sst}) = L0 \wedge\\
&\qquad \text{m}_c = \{\text{PH_DATAind}(\text{nodeid}, p)\} \wedge\\
&\qquad \text{VALID}(p) \wedge (\neg \text{BCAST}(p)) \wedge \text{nodeid} = \text{dest}(p) \wedge\\
&\qquad \text{m}_p = \{\text{PH_ARBreq}(\text{nodeid}, \text{IMMEDIATE})\} \wedge\\
&\qquad \text{pac}(\text{est}) = p \wedge\\
&\qquad \text{loc}(\text{est}) = L4) \vee\\
&\quad \ldots\}
\end{aligned}$$

Using the definition of a single *LinkLayer*, the parallel composition of all Link layers on the bus is defined by recursive application of the parallel composition operator:

$\text{LinkLayer}(\text{nodeid}):\ \text{CTS} =$
$\quad \text{seq}(\text{nodeid},\ \{\text{st} \mid \text{loc}(\text{st}) = L0\}, \text{LinkTraRel}(\text{nodeid}))$

$\text{LinkLayersUpto}(\text{nodeid}) : \textsc{recursive}\ \text{CTS} =$
$\quad \textsc{if}\ \text{nodeid} = 0\ \textsc{then}\ \text{LinkLayer}(0)$
$\quad \textsc{else}\ (\text{LinkLayer}(\text{nodeid}) || \text{LinkLayersUpto}(\text{nodeid} - 1))$
$\quad \textsc{endif}$
$\textsc{measure}\ \text{nodeid}$

$\text{AllLinkLayers} :\ \text{CTS} = \text{LinkLayersUpto}(N - 1)$

$\textsc{end}\ \text{linklayer}$

The operation of the Link layers on a bus is highly dependent on the operation of their environment (the parallel composition of all Transaction layers and Physical layers). We have therefore defined two CTS's to model the environment behaviour *as far as necessary to understand the operation of the Link layer*. One CTS models the operation of one Transaction layer, an other CTS models the operation of the parallel composition of all Physical layers. Their graphical representations are given in figures 5 and 6 below.

5 Conclusion and Future Work

As a first step towards the mechanical verification of the asynchronous part of the Link layer of the IEEE P1394 serial bus, we have formalized the specification given in [3], using the specification language of PVS. To keep the probability of introducing modeling errors small, we have not tried to model the protocol in an existing state machine framework, but instead tried to give a formal model close to the original description.

We consider the techniques for the formal analysis of the Link layer protocol, in particular the development of generic PVS theories for Communicating Transition Systems, to be the main contribution of our work. Using PVS for the formalization of the framework and the protocol additionally proved useful in the detection of errors by type check conditions and putative lemmas.

In general, it turned out to be rather complicated to give an abstract description of the protocol. For instance, the abstraction from bits led to discussions about which actions are considered to be atomic. It was also not immediately clear how to model the few points in the protocol where timing is explicitly mentioned. However, most time was spent on the formalization of the framework, and for dealing with the vagueness of the natural language explanations during the formalization of the protocol.

An important result of such a formalization is a list of ambiguities, unclarities, etc, about the original specification. The most prominent questions which could not be clarified are:

- How is re-arbitration handled in the case of a LOST Physical arbitration confirmation?
- How is a situation prevented where two nodes keep sending each other concatenated responses? This would imply that both nodes use only IMMEDIATE arbitration. Hence, all other nodes trying to gain access by means of a FAIR arbitration will never succeed.

As a continuation of this work, we have formulated a number of assumptions about the environment of the protocol, such as relevant properties of the services offered by the Physical layer and the behaviour of the Transaction layer. They have been used to give a manual proof, using linear time temporal logic [5], of some important properties of the protocol. In the proof the CTS's from figures 5 and 6 are not used directly but only serve to show that the assumptions about the environment have a model.

Using the abbreviations $L_i \stackrel{\text{def}}{=} (\text{loc}(\mathcal{S}(i)) = L)$ and $p_i \stackrel{\text{def}}{=} \text{pac}(\mathcal{S}(i))$ we can state the most interesting safety properties of the protocol:

- At any time only one node's Link layer is at L2', L4, L5 or L7. This implies that it is never the case that two Link layers are simultaneously in a position where they can put packets on the bus. Furthermore only one node at a time can receive a non broadcast packet.

  $\forall i \neq j : \Box\neg((L2'_i \vee L4_i \vee L5_i \vee L7_i) \wedge (L2'_j \vee L4_i \vee L5_j \vee L7_j))$
- If one node has received an asynchronous data packet and is at location L5 (about to send the acknowledgment) then the sender of the packet is waiting for the acknowledgment in L2.

  $\forall i : \Box(L5_i \rightarrow L2_{\text{source}(p_i)})$

As a consequence of the above mentioned unclarities, we are only able to prove response properties of the protocol which are weaker than intuitively expected. Only if access to the bus is achieved then the packet reaches all intended Transaction layers and a confirmation is sent back to the sending Transaction layer.

- $\forall i, j, p : \Box($
  $\text{BCAST}(p) \wedge p = p_i \wedge L2'_i \rightarrow$
  $\Diamond\text{LK\_DATA.ind}(j,$
  $\quad \text{DISASSEMBLE}(p), \text{STATUS}(p)) \wedge$
  $\Diamond\text{LK\_DATA.conf}(i, \text{BROADCAST\_SENT})\quad)$

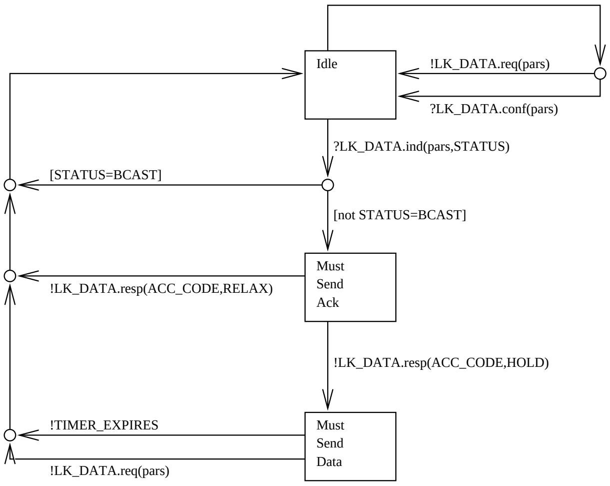

Figure 5: CTS describing the interaction of one Transaction layer with one Link layer

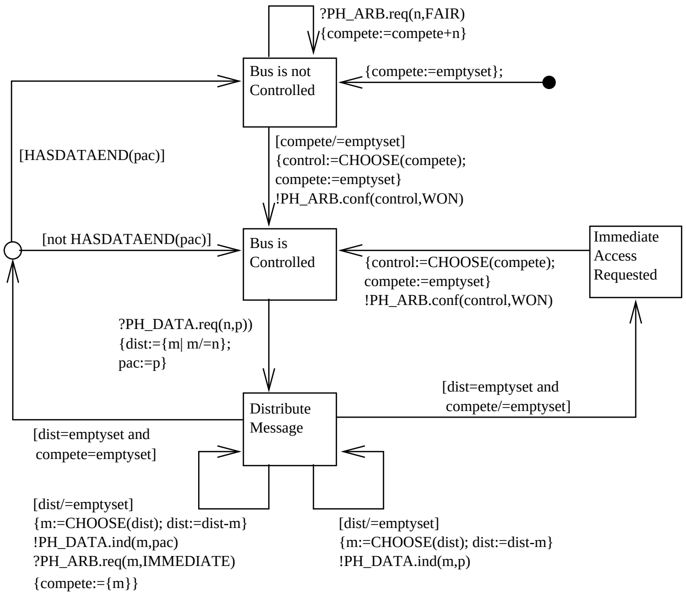

Figure 6: CTS describing the interaction of all Physical layers with all Link layers. The function CHOOSE returns an arbitrary element from a nonempty set. "compete" is a set variable. The operators + and - respectively add and subtract an element to/from a set

- $\forall i, j, p : \Box($
  $\neg\mathrm{BCAST}(p) \wedge p = p_i \wedge L2'_i \wedge j = \mathrm{dest}(p) \rightarrow$
  $\Diamond\mathrm{LK\_DATA.ind}(j,$
  $\mathrm{DISASSEMBLE}(p), \mathrm{STATUS}(p)) \wedge$
  $\Diamond\exists c : \mathrm{LK\_DATA.conf}(i, c) \qquad)$

Note in particular that it is not the case that each LK_DATA.req results in a corresponding LK_DATA.-ind. This is especially problematic since the Transaction layer is not informed of LOST Physical Arbitration confirmations, so it does not have the chance to repeat the LK_DATA.req.

The hand proofs of all mentioned properties will be published in [4]. Currently we are working on the integration of linear time temporal logic into PVS. With this at hand, we plan to use PVS to check the proofs of the protocol properties mechanically.

Acknowledgements

We would like to thank Kai Engelhardt for his support, and the Ministerium für Bildung, Wissenschaft, Forschung und Kultur of Schleswig-Holstein for paying the travel costs involved in this international collaboration. We are also grateful to the anonymous referees for useful suggestions for improvement.

References

[1] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[2] R. DeNicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34:83–133, 1984.

[3] IEEE. *P1394 Standard for a High Performance Serial Bus, Draft 8.0v2*. The Institute of Electrical and Electronic Engeneers, July 1995.

[4] L. Kühne. Master's thesis, Christian-Albrechts Universität Kiel, 1997. To appear.

[5] Z. Manna and A. Pnueli. *The temporal logic of reactive systems*. Springer-Verlag, 1991.

[6] E. R. Olderog and C. A. R. Hoare. Specification-oriented semantics for communicating processes. *Acta Informatica*, 23:9–66, 1986.

[7] S. Owre, N. Shankar, and J. M. Rushby. *The PVS Specification Language*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993.

[8] N. Shankar, S. Owre, and J. M. Rushby. *PVS Tutorial*. Computer Science Laboratory, SRI International, Menlo Park, CA, February 1993. Also appears in Tutorial Notes, *Formal Methods Europe '93: Industrial-Strength Formal Methods*, pages 357–406, Odense, Denmark, April 1993.