Time, E-LOTOS, and the FireWire

Carron Shankland [1] and Alberto Verdejo [2]

[1] Department of Computing Science and Mathematics,
University of Stirling
ces@cs.stir.ac.uk
[2] Departamento de Sistemas Informáticos y Programación
Universidad Complutense de Madrid
jalberto@eucmos.sim.ucm.es

Abstract. The proposed ISO standard formal description technique E-LOTOS is used to describe a leader election protocol (that of the IEEE 1394 serial multimedia bus), allowing illustration of the new aspects of the language, particularly time and parallelism.

Keywords: E-LOTOS (Enhancements to LOTOS), IEEE 1394, Protocols, Leader Election Algorithm, Formal Methods.

1 Introduction

The proposed ISO standard formal description technique E-LOTOS is used to give a timed description of the leader election protocol of the IEEE 1394 serial multimedia bus, demonstrating the capabilities of the new language for describing communications protocols.

The 1394 (FireWire) serial multimedia bus is an IEEE standard for multimedia communications. It connects together a collection of systems and devices in order to carry all forms of digitized video and audio quickly, reliably, and inexpensively. Its architecture is scalable, and it is "hot-pluggable", so a designer or user can add or subtract systems and peripherals easily at any time.

The 1394 as a whole is complex, comprising of several different subprotocols, each concerned with different tasks (e.g. data transfer between nodes in the network, bus arbitration, leader election). In the standard [IEE95] it is described in layers, in the style of OSI, and each layer is split into different phases. Our focus here is the tree identify phase of the physical layer.

Although formal methods were not used in the development of the 1394 standard, various aspects of the system have been described elsewhere using a variety of different techniques, including I/O automata [DGRV97,Rom99], μCRL [SvdZ98], and E-LOTOS [SM97]. Most of these do not address the real time aspects of the system, although these are an important part of the C++ implementations of the IEEE standard. Our main aim is to model the standard as closely as possible, particularly addressing the real time issues.

In this paper we present a specification in E-LOTOS of the Tree Identify protocol of the 1394. E-LOTOS (Enhancements to LOTOS) is a new formal description technique, currently undergoing the standardisation process [ISO98]. The ISO formal description technique LOTOS [ISO88] has been in use for over a decade, and although it has been shown to be useful in some areas, the language was perceived to have several failings, hence the development of E-LOTOS. Many new features have been added, the most important being better modularity, a more flexible data type specification language, and the capability to express real time.

A secondary aim of this case study was to present and evaluate features of E-LOTOS. Although not all new features have been used here, we feel that this paper will serve as an introduction to the main features of E-LOTOS for experienced LOTOS users and newcomers alike.

Our experience with E-LOTOS has been positive. The description of this reasonably complex protocol can be made in a modular fashion, breaking down phases into processes, and separating the data type specifications from the process specifications. The real time aspects can be introduced to the specification without detracting from the non-real time aspects, and synchronisation of pairs of processes in a network of many similar processes can be easily specified.

Familiarity is assumed with LOTOS or E-LOTOS. Tutorials to E-LOTOS can be found in an appendix to the committee draft standard [ISO98] and in [Ver99].

In the next section the 1394 tree identify protocol is introduced informally. There follows an E-LOTOS specification of the protocol, and finally some conclusions are drawn about our experience with E-LOTOS.

2 Informal Overview of the Protocol

The tree identification protocol of IEEE 1394 is a leader (root) election protocol which takes place after a bus reset in the network (i.e. when a node is added to, or removed from, the network). Immediately after a bus reset all nodes in the network have equal status, and know only to which other nodes they are connected. A leader (root) must be elected to serve as the bus manager for the other phases of the 1394. Figure 1(a) shows the initial state of a possible network. Connections between nodes are indicated by solid lines.

Each node carries out a series of negotiations with its neighbours in order to establish the direction of the parent-child relationship between them. More specifically, if a node has n connections then in the normal case it receives “be my parent” requests from all, or all but one, of its connections, up to a time limit `CONFIG_TIMEOUT`. If `CONFIG_TIMEOUT` is reached before $n-1$ requests are made this indicates that the network contains a loop, therefore it is not possible to configure the network as a tree, and an error is reported.

Assuming n or $n-1$ requests have been made the node then moves into an acknowledgement phase, where it sends acknowledgements “you are my child” to all the nodes which sent “be my parent” in the previous phase. When all acknowledgements have been sent either the node has n children and therefore is the root node, or the node sends a “be my parent” request on the so far unused connection and awaits an acknowledgement from the parent. Leaf nodes skip the

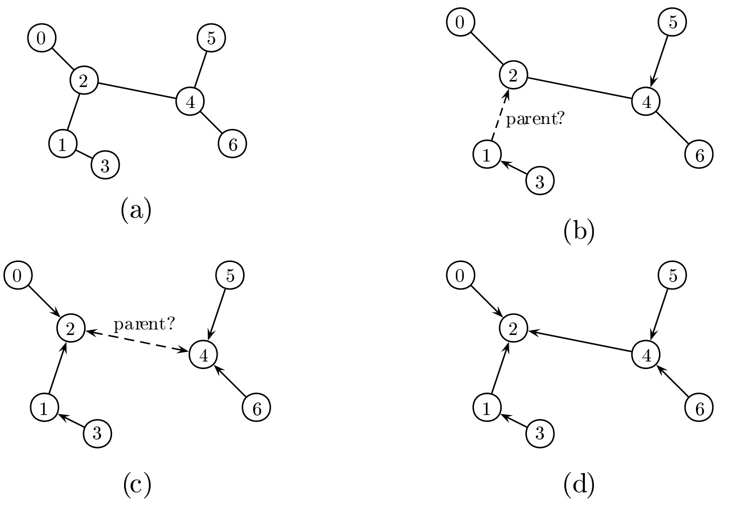

Fig. 1. Network Configurations during the Tree Identify Protocol

initial receive requests phase and move straight to this point; they have only one connection therefore it must be their parent. Figure 1(b) shows the instant when node 3 and 5 has their parent already decided (solid connections with arrows pointing to the parent), and node 1 is asking node 2 to be its parent (the queried relationship is shown by a dotted line and arrow).

Communication between nodes is asynchronous therefore it is possible that two nodes might simultaneously request that the other is their parent, leading to root contention (each wants the other to be the root, see Figure 1(c)). To resolve root contention, each node selects a random Boolean. The value of the Boolean specifies a long or short wait before resending the "be my parent" request. This may lead to contention again, but fairness guarantees that eventually one node will become the root.

When all negotiations are concluded, the node which has established that it is the parent of *all* its connected nodes must be the root node of a spanning tree of the network. See Figure 1(d) in which node 2 is the root node.

Setting a node's `FORCE_ROOT` parameter is intended to alter this basic pattern of communication by delaying the transition from the first phase (receiving "be my parent" requests) to the acknowledgement phase, therefore there is a higher probability that the node receives requests on all its connections, thus ensuring that it becomes the root of the tree.

There are two desirable properties for this system:

– A single leader is chosen (safety).

– A leader is eventually chosen (liveness).

In addition to describing the protocol of the standard [IEE95] the specification presented here should also satisfy these properties.

3 Timed E-LOTOS Specification of 1394

3.1 Overview: General Design Decisions

Our aim in the specification of the FireWire is to model the protocol as specified in the standard [IEE95] as closely as possible, using as a basis the state transition diagrams, C++ implementations, and text therein. The reasons for this are twofold. Firstly, this is a post-hoc specification exercise therefore the point is to capture the particular leader election algorithm described in the 1394 standard [IEE95], especially any errors in that description. Secondly, it is a test of the E-LOTOS specification language to confirm it is suited to description of such systems.

This decision to model the system as closely as possible has led to a more concrete specification than might otherwise be desirable if developing the specification from informal requirements. For example, the I/O automata specifications of [DGRV97] and the μ-CRL descriptions of [SvdZ98] are rather more abstract; possibly this is an effect of the specification languages chosen. Different levels of abstract are explored over a number of specifications in these papers, including ignoring all node negotiation, ignoring contention, having synchronous or asynchronous communication and ignoring time. The closest specification to that here in terms of level of abstraction is the recent timed I/O specification of [Rom99], which concentrates on verification of the correctness of the specification but ignores the effect of the `FORCE_ROOT` parameter.

A rather concrete style of specification also fits the general E-LOTOS philosophy. Although it is a specification language, the development committee felt that it was better to allow software engineers to describe systems in as convenient a way as possible, i.e. using constructs familiar from programming languages such as variables, loops, if statements, case statements and exceptions. The main benefit (over just using a programming language for the description) is that the language has a formal basis, and is amenable to analysis.

That said, the specification language does encourage abstraction of certain aspects, e.g. network configuration and messages between nodes. Nodes in the real system have a number of ports which are connected or unconnected. The physical cable connections describe the network implicitly. In the specification the nodes have identifiers and the configuration of the network is modelled by a list of pairs (node identifier, list of identifiers of nodes to which that node is connected).

In the standard, nodes sample the ports to detect wire voltages which are interpreted as messages from the connected nodes. In the specification, messages are modelled as process events; however, not all messages used in the standard have been modelled here. Typically, a node receives a “be my parent” request, sends an acknowledgement to that request, and checks that acknowledgement has been received (all by setting or sampling wire voltages). The first two are modelled as process events; obviously the initial request must be part of the specification, and the acknowledgement is necessary to detect contention. The third message, checking the child has received an acknowledgement, is ignored. The message passing medium is assumed to be reliable (no messages are lost or corrupted).

Abstraction can introduce problems as well as simplify them. The protocol only operates correctly if the network can be configured as a tree (i.e. there are no loops and no disconnected components). The first is checked by the protocol of the standard itself. The second is not checked by the protocol because disconnected nodes are merely a different network. However, the representation chosen in the specification makes it a problem here because once a node is in the network list then it is part of the network, whether or not it is connected to any other nodes in the network. Therefore a check for disconnected nodes is added to the specification.

Communication between nodes is asynchronous and modelled by one place buffers. One place buffers are sufficient since only one message is sent between two nodes in a particular direction at any given time.

Presentation of the E-LOTOS specification of the 1394 follows in three main sections: modules, data types and processes (which is where the timing details are discussed).

3.2 Modules

One of the enhancements of E-LOTOS is *modularity*, which allows the definition of types, functions, and processes in separate modules, control of their visibility by means of module interfaces, and the definition of generic modules, useful for code reuse.

In this paper the module system is illustrated by describing the data types as one module, the processes specifying the behaviour of the different phases of the tree identification protocol as another, and the time constants as a third. All modules are used (imported) by the main specification of Figure 2.

E-LOTOS also offers further parameterisation and reuse facilities via abstraction and encapsulation, but the specifications here do not lend themselves to such further modularisation.

The variable `P:conntable` read from the environment in the specification of Figure 2 models the initial configuration of the network. `P` is a list of pairs, where the first element indicates the node identifier and the second element is the set of nodes to which it is connected. For example, to match the configuration of the network as shown in Figure 1 the value of `P` would be `[(0,{2}), (1,{2,3}), (2,{0,1,4}), (3,{1}), (4,{2,5,6}), (5,{4}), (6,{4})]`. In this example the `conntable` is ordered, but this is for clarity and is not essential to correct operation of the specification.

```
specification TimedFireWire import Types, FireWireProcs, Constants is
  gates read, leader, error
  behaviour
  var
    P:conntable
  in
    trap
      exception LoopException is error endexn
      exception ConnectionError is error endexn
    in
      (* input the network configuration from an external source *)
      read (?P);
      if (sizeof(P)=1 and not connected(P)) then
        leader; SelfIDT[...](id, p) (* we're done *)
      else if connected(P) then ImpT[leader](P)[LoopException]
                           else raise ConnectionError
           endif
      endif
    endtrap
  endvar
endspec
```

Fig. 2. Top Level of the FireWire Specification

Exceptions have been added to E-LOTOS, hence the `conntable` (network) is checked for disconnected components and if there are any an exception is raised (since the algorithm is not designed to operate correctly in that case).

A new notation has been added to E-LOTOS, one which was in common informal use among LOTOS users. This is the use of `[...]` to denote the gate parameters of a process in an instantiation where actual gates identifiers are equal to the formal ones. Although seemingly trivial, this is actually a big bonus!

3.3 DataTypes

The part regarding the declaration and use of data types is one of those that has been changed more in E-LOTOS with respect to its predecessor LOTOS. In LOTOS the abstract data type specification language ACT ONE [EM85] is used to declare new data types and to represent their values. This language is fairly user-unfriendly and suffers from limitations such as the semi-decidability of equational specifications, the lack of modularity, and the inability to define partial operations. In E-LOTOS, ACT ONE has been substituted by a new language in which data types are declared in a similar way to that in functional languages (ML, Haskell), and where some facilities for value use and manipulation are given.

The data types are given in Figure 3 and the function definitions in Figure 4. Comments are enclosed within `(*` and `*)`. The novel features of E-LOTOS are described in some detail below.

```
module Types is
  (* Node identifiers *)
  type iden renames nat endtype

  (* Set of connections *)
  type connections is set of iden endtype

  (* Node plus the set of nodes to which it is connected *)
  type pair is (iden, connections) endtype

  (* Network description *)
  type conntable is list of pair endtype

  (* Two types of message can be sent *)
  type conntype is parent | ack endtype

  (* Standard form of communications *)
  type comm is (iden, iden, conntype) endtype
```

Fig. 3. Data Types for the FireWire Specification

Standard types such as integers, Booleans, lists and sets are part of the language. E-LOTOS allows synonymous types to be defined to facilitate more meaningful data type names, e.g. `iden` which is the set of natural numbers.

The type of `connections` is specified as a set of identifiers. E-LOTOS provides standard functions to manipulate sets that are used in the processes below (Section 3.4). Similarly list, as in `conntable`, is a built-in type and E-LOTOS provides several functions to manipulate lists.

New user defined data types can be declared by enumeration of all the constructors, as in `conntype`. Each constructor may also have data arguments, but in this case there are none.

The record type `(iden, iden, conntype)` is used to model communications in the system by identifying the source and destination nodes, and also the message type. Such anonymous type cannot be used directly in E-LOTOS, so it is given a name: `comm`.

Of course, user-defined data types are of limited use without user-defined functions. Those for the FireWire are given in Figure 4. Although several functions are provided with the built-in sets, the function to check if a given set has only one member is not provided, so it is declared here.

The next three functions are all concerned with manipulating the `conntable` (network configuration). The first, `connected`, returns true if all nodes are associated with a non-empty set of connections. A `case` statement is used to pattern match on list constructors. This technique is also used in `neighbours`, which returns the connections associated with a particular node, and `sizeof`, which returns the number of nodes in the network. In both of these functions it is

```
  (* True if the current set is a singleton, false otherwise. *)
  function singleton (c:connections):bool is
      card(c) = 1
  endfun

  (* Checks that all nodes are connected to at least one other node *)
  function connected(ct:conntable):boolean is
    case ct is
        nil -> true
      | cons ((?n, ?c), ?cs) -> if empty(c) then false
                                 else connected(cs) endif
    endcase
  endfun

  (* Returns the length of the conntable *)
  function sizeof(ct:conntable):nat is
    case ct is
        nil -> 0
      | cons ((?n, ?c), ?cs) -> 1 + sizeof(cs)
    endcase
  endfun

  (* Extracts the list of connections associated with index x *)
  function neighbours(ct:conntable,x:nat):connections is
    case ct is
        nil-> {}
      | cons ((?n,?c), ?cs)  -> if x = n then c
                                 else neighbours(cs,x) endif
    endcase
  endfun

  (* Produce a list of naturals from x to y *)
  function infix upto (x:nat, y:nat):List is
    if y < x then nil else cons(x, (x + 1) upto y) endif
  endfun
endmod
```

Fig. 4. Function Definitions for the FireWire Specification

assumed that the list contains no duplicates. The `if` and `case` statements are new features of E-LOTOS.

Infix functions can also be defined, as in the `upto` function, to allow a more natural specification style.

3.4 Processes

The behaviour of the system is modelled by processes and events. Each node in the network is modelled by a separate process and the whole is given by the parallel composition of these nodes. The combination of these processes to give the whole network is described first, followed by the behaviour of individual nodes.

The whole network is described by:

```
module FireWireProcs is
process ImpT [leader, error] (P:conntable) raises [LoopException] is
var
  id:iden, n:nat, l:list of nat
in
  ?n := sizeof(P);
  ?l := (0 upto (n-1));
  hide c:comm in
    par c#2 in
         [c] -> par ?id in l
                   ||| TreeIDT [...](id, neighbours(P,id),
                                          {}, false)[LoopException]
                endpar
      || [c] -> Buffers[...](n)
    endpar
  endhide
endvar
endproc
```

Two novel forms of parallelism are introduced in E-LOTOS, and both are used in the `ImpT` specification. The first is *parallel over values*, seen in the line

```
par ?id in l
  ||| TreeIDT [...](id, neighbours(P,id), {}, false)[LoopException]
endpar
```

The variable `id` is instantiated from a list of n values, producing n instantiations of the `TreeIDT` process, each with parameters dependent on the current value of `id`. `TreeIDT` describes the behaviour of an individual node in the system. Parallel over values therefore allows a general description of a group of processes (run in parallel), without specifying the data parameters for each process in the network individually, or specifying the number of processes in the system. In LOTOS such systems had to have the values and the number of processes hardwired into the specification.

The second novelty is that synchronisation can take place on an n from m basis, here exemplified by the 2 from n synchronisation between `Buffers` and `TreeIDT` processes. In LOTOS, multiway synchronisation and the particular parallel operators available mean that if all processes in a network are specified to synchronise on a particular gate then every communication on that gate has to be participated in by *all* processes. To describe the sort of network presented here means hardwiring the number of nodes, and defining special communication channels for pairs of connected nodes.

Here, although potentially each node can communicate with all buffers simultaneously (because they all use the gate `c` to synchronise), it is forced by use of the n from m parallel construct to communicate with only one buffer (i.e. synchronisation involves two processes from a possible n processes). Additionally, data parameters to the events are used to make sure that the node communicates with the *right* buffer and hence the right destination node.

A LOTOS notational device which has been preserved in E-LOTOS is the use of `?` to denote a binding occurrence of a variable. Notice that here it is used in an assignment statement; another of the features drawn from imperative programming. Use of a `!` denotes a value, or used occurrence of a variable.

Hiding is also inherited from LOTOS. Here all the events on gate `c` will be hidden from the environment. This means that the only events observable in the environment are `leader` and `error`. This would be useful if a formal analysis, such as a proof that the specification satisfies a formal version of the properties stated in Section 2, were to be carried out. Additionally, hiding is used to force the evolution of the system, since events on hidden gates are *urgent*, that is, they have to be performed as soon as possible unless another event occurs without consuming time.

Communication between nodes occurs via `Buffers` (therefore communication is asynchronous). Messages in the 1394 are sent along wires of variable length (up to 4.5m), therefore message passing is also subject to delay. The `Buffer` processes here introduce randomly chosen delays (specific wire lengths are not used for particular connections). `Buffer[c](j, k)` represents the wire between nodes `j` and `k`.

```
process Buffer[c:comm](j:iden, k:iden) is
var
  m:conntype, t:time
in
  loop
    c(!j, !k, ?m);
    ?t := any time [(lower_buffer < t) and (t < upper_buffer)];
    wait(t);
    c(!j, !k, !m)
  endloop
endvar
endproc
```

The `Buffer` is the first process presented here which uses time. Real time may be added to a specification in one of two ways: by defining the exact time at which events occur (by adding annotations to actions), or by defining the time that behaviours take (by using `wait` statements). We only use the second method in this specification.

Buffers receive a message, choose a delay time, wait for that length of time, pass the message on, then loop (another feature introduced to E-LOTOS from programming languages). The choice of delay time is nondeterministic; this is signified by the `any` wildcard. The `wait` statement is used to force time to pass, necessary because all `c` actions are hidden and therefore urgent.

Each `TreeIDT` process uses two buffers to communicate with another node; one for each direction of communication. A network of n^2 buffers parameterised by $(0,0),(0,1),\ldots,(1,0),(1,1),\ldots,(n-1,n-1)$ is created using the parallel over values construct. This produces more buffers than are necessary (since the actual network should not be fully connected), but the specification is simple, and the extra buffers do not interfere with other events. Buffers are interleaved; they do not communicate with each other. Figure 5 shows the buffers between nodes 1, 2 and 3 of Figure 1. Those buffers between nodes 2 and 3 are shown in a different typeface to indicate that they are not used (they do not model a connection in the real network).

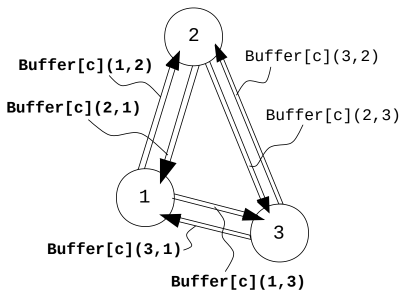

Fig. 5. Buffers

In E-LOTOS, the network of buffers is constructed by:

```
process Buffers[c:comm](n:nat) is
var
  j:iden, k:iden, l:list of nat
in
  ?l := (o upto (n-1));
  par ?j in l
    ||| par ?k in l
          ||| Buffer[c](j, k)
        endpar
  endpar
endvar
endproc
```

A single node in the system is modelled by the process `TreeIDT`; this process includes timing details in the separate process `AlarmClock` and the first phase of the protocol proper, receive "be my parent" requests, in `ReceiveReqs`. The other phases are described in separate processes: `SendAcks`, `WaitPar`, and `Contention`. A further phase, `SelfIDT`, denotes the part of the 1394 which follows the tree identification protocol. `SelfIDT` is not described in this case study.

```
process TreeIDT[leader, c:comm]
              (id:iden, p:connections, ch:connections, fr:bool)
              raises [ErrorExc] is
 hide alarmct, alarmfr, stopAlarmct, stopAlarmfr in
    ReceiveReqs[...](id,p,ch,card(p),fr)[ErrorExc]
  |[alarmct, alarmfr, stopAlarmct, stopAlarmfr]|
    ( AlarmClock[alarmct, stopAlarmct](CONFIG_TIMEOUT)
    |||
      if fr then AlarmClock[alarmfr, stopAlarmfr](FRTIME) endif)
  endhide
endproc
```

The parameters used here are:

- `id`, the identification number of the node,
- `p`, the set of communications still to make (initially the set of nodes connected to `id`), and
- `ch`, the children to be acknowledged.

The main use of `p` and `ch` is in the `ReceiveReqs` phase and the `SendAcks` phase. In `ReceiveReqs`, as each request is made the node making the request is removed from the set `p` and added to the set `ch`. The `ReceiveReqs` phase terminates and `SendAcks` is called when `p` is empty, or is a singleton set. In `SendAcks` as each child is acknowledged that child is removed from the set `ch`. When all negotiations are complete, `p` will be empty (indicating that the node `id` is the leader), or will have one member (the parent of `id`).

Timing concerns are separated from the procedure of negotiating the parent relationship with other nodes. There are two timing considerations for each node. The first is whether or not the `CONFIG_TIMEOUT` has been exceeded. This indicates that the network has been set up incorrectly (i.e. it includes a loop) and an exception is raised. In the specification a separate "alarm clock" is used which waits for `CONFIG_TIMEOUT` time units. The alarm is run in parallel with `ReceiveReqs` and notifies this process with an `alarmct` event when `CONFIG_TIMEOUT` expires.

The second timing consideration concerns the force root parameter `fr`. Normally it is possible for a node to move to the `SendAcks` phase when $n-1$ communications have been made (where n is the initial size of the set `p`, that is, the neighbours of the node). Setting `fr` forces the node to wait a bit longer, in the hope that all n communications will be made (and the node becomes the leader). To model this we set another alarm clock with timeout value of `FRTIME`. If `fr` is set then `stopAlarmct` is only enabled if n communications have been made, until the second alarm (`alarmfr`) goes off after which `stopAlarmct` is enabled when $n-1$ communications have been made. The second alarm is only needed when `fr` is set.

The alarm clock process is

```
  process AlarmClock[alarm, stopAlarm](TO:time) is
       stopAlarm
    []
       wait(TO); alarm
  endproc
```

and the main message passing part of the node is

```
process ReceiveReqs
        [leader,c:comm,alarmct,alarmfr,stopAlarmct,stopAlarmfr]
        (id:iden, p:connections, ch:connections, n:nat, fr:bool)
        raises [LoopExc] is
var j:iden in
     (* receive a "be my parent" request *)
     c (?j, !id, !parent)[isin(j, p)];
     ReceiveReqs[...](id, diff(p,{j}), union(ch,{j}), n, fr)[LoopExc]
  []
     (* Stop receiving requests at either n or n-1.
        The n-1 test is delayed when fr is set. *)
     stopAlarmct [if fr then (card(ch)=n) else (card(ch) >= n-1) endif];
     (* FRTIME alarm may still be running, so stop it too *)
     if fr then stopAlarmfr endif;
     SendAcks[...](id,p,ch)
  []
     (* Timeout on FRTIME, therefore start testing for n-1 connections.
         Set fr to false to denote this. *)
     alarmfr;
     ReceiveReqs[...](id, p, ch, n, false)[LoopExc]
  []
     (* Timeout on CONFIG_TIMEOUT, therefore stop receiving requests. *)
     alarmct;
     (* FRTIME alarm may still be running, so stop it *)
     if fr then stopAlarmfr endif;
     if (card(ch) < n-1) then
      (* timeout and not enough communications *)
        raise LoopExc
     else
      (* although timeout, enough communications, so continue *)
        SendAcks[...](id, p, ch)
     endif
endvar
endproc
```

Note that although `stopAlarmct` is enabled when $n-1$ communications have been made (when `fr` is not set), a further communication may happen before the alarm is stopped (if that communication is ready). This satisfies the text of the standard which specifies that a node moves on to the next phase on all or all but one communications.

The next two phases are not affected by timing considerations.

```
process SendAcks [leader, c:comm]
                 (id:iden, p:connections, ch:connections) is
(* From previous phase it is known that p = {} or {k}, where k is the
   potential parent of this node. *)
var j:iden, k:iden in
  case true is
      !isempty(ch)    -> (* No more acks to make, so make a parent
                            request (to the only k in p).           *)
                         c(!id, ?k, !parent)[isin(k, p)];
                         WaitParent [...] (id, p)
    | !singleton(ch)  -> (* make last ack and if empty(p) send leader *)
                         c(!id, ?j, !ack)[isin(j, ch)];
                         if isempty(p) then
                            leader; SelfIDT[...](id, p)
                         else
                            SendAcks[...](id, p, {})
                         endif
    | any:bool        -> (* while more children, send an ack *)
                         c(!id, ?j, !ack)[isin(j,ch)];
                         SendAcks[...](id, p, diff(ch,{j}))
  endcase
endvar
endproc
```

Here the case statement is used with `true` which has the effect of making statements guarded by the conditions of the case. Note that the guards are evaluated in top to bottom order (so it is important to put the wildcard last).

This phase sends acknowledgements to all the children of the current node, and when finished either sends a “be my parent request” to its parent (if there is an element in `p`) or declares itself leader (if the set `p` is empty). Here the case statement guards match against values; they may also match against constructors (as was seen in the function definitions).

If a parent request has been sent, then a node waits for an acknowledgement. If a parent request arrives instead, then the two nodes are in contention for leader.

```
process WaitParent [leader, c:comm] (id:iden, p:connections) is
var j:iden in
  (* There will be only one member of p - the parent.
     Either get an ack and are done, or a contentious parent req *)
  ?j := any iden [isin(j,p)];
  (  c(!j, !id, !ack); SelfIDT[...](id, p)
  [] c(!j, !id, !parent); Contention[...](id, p))
endvar
endproc
```

In the standard, contention is resolved by choosing a random Boolean `b` and waiting for a short or long time depending on `b` before sampling the relevant port to check for a “be my parent” request from the other node. If the request is there then this node should agree to be the root and send an acknowledgement to the other. If the message is not present, then this node will resend its own “be my parent” request.

The specification presented here differs from the standard because of the nature of synchronisation in E-LOTOS; it is not possible to check for a message. Instead, the Boolean is chosen and a wait time selected. If a “be my parent” request arrives during the wait then the wait aborts and the request is dealt with. If the wait time expires then the node resends its own “be my parent” request.

```
process Contention [leader, c:comm]
                   (id:iden, p:connections) is
(* There will be only one j in p.
   This means that both nodes (i and j) have sent a parent request. *)
var j:iden, b:bool, t:time in
  ?b := any bool;
  ?j := any iden [isin(j,p)];
  if b then ?t := ROOT_CONTEND_SLOW
      else ?t := ROOT_CONTEND_FAST
  endif;
  (  wait(t); c(!id, !j, !parent); WaitParent[...](id, p)
  []
     c(!j,!id,!parent); SendAcks[...](id, {}, {j}) )
endvar
endproc
endmod
```

To complete the specification it is also required to declare some constants of type time (Figure 6). The time values are taken from the standard [IEE95]:

- `upper_buffer` set to 22.725 ns models the maximum buffer delay time (assuming a maximum cable length of 4.5m and propagation delay of 5.05 ns, both of these are specified in the standard),
- `lower_buffer` set to 0 ns models the minimum buffer delay time (no minimum cable length is given in the standard),
- `CONFIG_TIMEOUT` set to a value in the range 166.6 – 166.9 μs, determines how long a node waits for parent requests in the normal case,
- `FRTIME` set to a value in the range 83.3 – `CONFIG_TIMEOUT` μs, determines how long a node delays the $n-1$ communications check when force root is set,
- `ROOT_CONTEND_FAST` set to a value in the range 0.24 – 0.26 μs is the small delay in contention resolution, and
- `ROOT_CONTEND_SLOW` set to a value in the range 0.57 – 0.60 μs is the longer delay in contention resolution.

```
module Constants is
  value upper_buffer:time is 23 endval
  value lower_buffer:time is 0 endval
  value CONFIG_TIMEOUT:time is 166600 endval
  value FRTIME:time is 84000 endval
  value ROOT_CONTEND_FAST:time is 250 endval
  value ROOT_CONTEND_SLOW:time is 580 endval
endmod
```

Fig. 6. Time constants

At one point we considered inserting a delay in the `ReceiveReqs` process, to model the length of time a node takes to sample a port. While this might make the passing of time more realistic, it adds nothing else to the specification. The only time critical part concerns the `CONFIG_TIMEOUT`, and in the case of a loop this will always be reached. It is also important not to report a loop if there is none. The size of the buffer delays, the maximum number of nodes in the network (64), the maximum number of cable hops between nodes (16) guarantees that any communications between nodes will arrive before the `CONFIG_TIMEOUT` value. Also, it seems difficult choose appropriate values for these limits. The choice requires knowledge about the kind of processors used in the nodes, the other processing tasks that might slow down execution, the particular implementation of 1394, and the translation of that implementation to machine code.

4 Conclusions

We have presented the specification of a case study in the use of E-LOTOS for a reasonably complex protocol. In the case study we particularly aimed to express the synchronisation and real time aspects of the protocol. The features of E-LOTOS make this task reasonably straightforward, producing a compact, and easy to understand specification (particularly for those familiar with programming languages). This gives us confidence that E-LOTOS will be a suitable language for communications protocols and telecommunications in general.

The specification behaves correctly if it satisfies the properties mentioned in Section 2. While we did verify the correctness of the E-LOTOS specification by hand-execution (which is not particularly reliable) we were unable to carry out any automated checking, validation or verification, because at the moment tools for E-LOTOS are still under development. Although desirable, verification in this case was not essential since the tree identify protocol has already been verified using μCRL and I/O automata, and no errors were found. The same cannot be said for other parts of the protocol. The study of [SM97] describes the LINK and TRANS layer of this standard using E-LOTOS and uncovers an error in the state machines of the LINK layer during verification. (Verification was by translation into LOTOS, for which several analysis tools exist. This method was only possible because a restricted subset of E-LOTOS (without time) was used.)

Discovery of such errors strengthens the case for the use of formal methods in the development of standards and in all critical software systems.

Acknowledgements

Thanks to the British Council and the Ministerio de Educación y Ciencia for supporting this work under the EASEL project (project number 2246), and to Ken Turner of the University of Stirling, whose idea the EASEL project was. Thanks also to Luis Llana of the Universidad Complutense de Madrid for discussions about time in E-LOTOS, and Frank Kelly of the University of Stirling for discussions regarding the timing aspects of IEEE 1394.

References

[DGRV97] M.C.A. Devillers, W.O.D. Griffioen, J. Romijn, and F. Vaandrager. Verification of a Leader Election Protocol - Formal Methods Applied to IEEE 1394. Technical Report CSI-R9728, Computing Science Institute, University of Nijmegen, 1997.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[IEE95] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus. Std 1394-1995*, August 1995.

[ISO88] International Organisation for Standardisation. *Information Processing Systems — Open Systems Interconnection — LOTOS — A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour*, 1988.

[ISO98] International Organisation for Standardisation. *ISO/IEC JTC1/SC21 WG7: Enhancements to LOTOS*, May 1998. Final committee draft.

[Rom99] J.M.T Romijn. A Timed Verification of the IEEE 1394 Leader Election Protocol. In *Fourth International Workshop on Formal Methods for Industrial Critical Systems*, 1999.

[SM97] M. Sighireanu and R. Mateescu. Validation of the Link Layer Protocol of the IEEE-1394 Serial Bus (FireWire): an Experiment with E-LOTOS. Technical Report 3172, INRIA, 1997.

[SvdZ98] C. Shankland and M. van der Zwaag. The Tree Identify Protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10:509–531, 1998.

[Ver99] A. Verdejo. E-LOTOS: Tutorial and semantics. Master's thesis, 1999.