Using E-LOTOS to Pick a Leader

Carron Shankland*
Department of Computing Science and Mathematics,
University of Stirling
ces@cs.stir.ac.uk

Abstract

The paper presents a case study using formal methods to describe a bus protocol. Specifically, the proposed ISO standard formal description technique E-LOTOS is used to describe a leader election protocol (the tree identify protocol of the IEEE 1394 serial multimedia bus) in several different ways, allowing illustration of the new aspects of the language. Comparisons are drawn with its predecessor LOTOS, and also with other formalisms used to describe the same protocol.

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

1 Introduction

The ISO formal description technique LOTOS [ISO88] has been used for many varied systems, for example OSI protocols, avionics software, telephone systems, service feature interactions and even children's games. Such practical experience has shown that the language has several failings: the abstract data type language has long been criticised for being rather cumbersome, the lack of real time makes it difficult to model timing accurately, and the language lacks modularisation. Additionally, in practice, software engineers desire features familiar from imperative languages such as loops, variables, exceptions. In order to address these failings (and perceived omissions) a new version of LOTOS, known as Enhancements to LOTOS or E-LOTOS, has been developed and is currently undergoing standardisation [ISO98].

An important part of standardisation (and post-standardisation acceptance of a formal description technique) is to demonstrate the capabilities of the language. In this paper a case study is presented which illustrates the use of some of the new elements of the E-LOTOS language, and demonstrates its suitability for describing a relatively complex protocol at different levels of abstraction. The particular case study chosen is the tree identify protocol (a leader election protocol) of the IEEE 1394 serial multimedia bus [IEE95]. The IEEE standard

*This work was carried out under the British Council project EASEL.

uses a mixture of techniques (state transition diagrams, C++ implementations and informal text) to describe the bus. Although originally developed by Apple as the FireWire, many other companies including Sun, Microsoft and Philips have participated in the development of 1394 and associated systems.

Formal methods were not used in the development of the 1394 standard, but various parts of it have been the subject of case studies in the application of formal methods. The particular part addressed here has been described and verified using I/O automata [DGRV97, Rom99] and μCRL [SvdZ98]. It is therefore useful to compare E-LOTOS as a specification technique with these other techniques, i.e. to use the 1394 standard as a benchmarking exercise.

Our experience with E-LOTOS has been positive. The description of this reasonably complex protocol can be made in a modular fashion, separating the data type specifications from the process specifications, and dividing the processes into more easily understandable components. 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. This compares favourably with the performance of LOTOS, I/O automata and μCRL.

In the remainder of the paper familiarity is assumed with LOTOS [ISO88][1]. The main aim here is to draw out the improvements of E-LOTOS for the LOTOS user, but it is not our intention to provide a thorough tutorial to E-LOTOS; tutorials can be found elsewhere, for example in an appendix to the committee draft standard [ISO98] or in [Ver99].

In the next section the 1394 tree identify protocol at its most basic level is introduced informally. More information is added in each of the E-LOTOS description sections as the level of abstraction becomes lower. Section 3 compares the general features of our E-LOTOS descriptions, and experience of using E-LOTOS, with the 1394 descriptions in other formalisms mentioned above. Section 4 contains general comments about the specification strategy and overall design decisions and specification of the data types and functions. There follow four E-LOTOS descriptions at varying levels of abstraction of this protocol, beginning with a very simple description capturing the basic properties of the system (`FireWireSpec`), moving through two untimed descriptions, one with synchronous message passing (`FireWireUTS`), the other with asynchronous message passing (`FireWireUTA`), to end with a timed description of the system (`FireWireTA`). Finally some conclusions are drawn about the E-LOTOS experience.

2 Informal Overview of 1394

First of all, what is the system being described in this paper? The serial multimedia bus IEEE 1394 [IEE95] connects together a collection of systems and

[1]For those unfamiliar with LOTOS, the language comprises two parts: an abstract data type specification language based on ACT ONE, and a process algebra part which is similar to CCS and CSP. A tutorial on LOTOS may be found in [LFHH92].

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 temporal phases. In this paper only the tree identify protocol of the physical layer is described.

Informally, the tree identification protocol is a leader election which takes place after a bus reset in the system (i.e. when a node is added to, or removed from, the network). At this point all nodes in the system have equal status and are equally likely to be elected leader (or root). Each node carries out a series of negotiations with its neighbours in order to establish the direction of the parent-child relationship between them. When all negotiations are concluded, the node which has established that it is the parent of *all* connected nodes must be the root node of a spanning tree of the network, or the leader. The chosen leader serves as bus manager in other phases of the protocol.

The basis for the descriptions is the state transition diagrams, C++ implementations, and text of the IEEE standard [IEE95].

3 Comparison with other approaches

As mentioned in the introduction two other formal methods have been used to describe this protocol. In the (untimed) I/O automata paper [DGRV97] the specifiers take the same view of the system as in `FireWireUTA` here, but the system is described in terms of predicates over a graph and hence describes the behaviour of the whole network simultaneously. Their description is shown to be correct with respect to the basic requirements (a leader is eventually chosen and only one leader is chosen).

A timed I/O automata description is presented in [Rom99]. This is closest to the timed description of Section 4.6, although the effect of the `FORCE_ROOT` parameter is ignored. Again this description is shown to be correct with respect to the basic requirements.

In the μCRL paper [SvdZ98] there are several descriptions, corresponding to `FireWireSpec`, `FireWireUTS` and `FireWireUTA`. Unlike the I/O automata version, nodes in the network are specified individually; however, to aid the reasoning process all phases of the protocol are combined in one process, and a state variable controls the choices available. The three descriptions are shown to be equivalent.

E-LOTOS compares well with these approaches. Largely the same expressivity is available in both I/O automata and μCRL as in E-LOTOS (aside from the problem of neatly describing n similar parallel processes occurring in the description of `FireWireUTS`). At this level the main criteria for choosing one particular specification language over another are familiarity, universality (there is more point in writing a specification if others will be able to read it), tool support, and ease of use. The main advantage E-LOTOS has over the others is its similarity to programming languages (which is a definite bonus for software engineers who may not be comfortable with the more mathematical formalisms). It cannot be denied that there is some tradeoff between the mathematical elegance and brevity of I/O automata and μCRL and the verbosity of E-LOTOS. For example, the μCRL description of processes corresponding to those of Section 4.5 takes just under a page, as do the I/O automata versions.

A further consideration is the availability of analytical support, both theoretical and automated. Clearly E-LOTOS lacks both of these at the moment, but that is to be expected for a relatively new technique. For example, it is desirable to prove that all four descriptions satisfy the basic behaviour that a single leader is chosen (regardless of the identity of that leader) (safety), and that a leader is eventually chosen (liveness). There is a large body of suitable theory for similar formalisms that can be adapted for E-LOTOS.

We may also consider how E-LOTOS compares to the IEEE standard itself. The standard uses a number of different techniques to explain the operation of different sections of the protocol, and this can be distracting. For example, C++ implementations describe the actions taken in particular states (roughly corresponding to the phases of `FireWireUTA` or `FireWireTA`) and state transition diagrams describe how to move from one state to another. The whole picture has to be pieced together with reference to several parts of the standard. In addition, much information is included about timing details, signals on the bus, construction of the cables and connectors and so on, and these are a distraction from understanding the basic operation of the protocol. E-LOTOS allows the state transition information and the state activity information to be coded in one language, and allows us to abstract away from non-essential details. This is more helpful, in the opinion of the author, in understanding the protocol.

4 E-LOTOS descriptions

Four different versions of the tree identify protocol are considered in Section 4.3 through to Section 4.6. These are *informal* refinements of the system; there is no formal refinement system for E-LOTOS at present. The aim is to build up increasingly accurate representations of the system, culminating in the timed description of Section 4.6. While our aim is to model the system described in the 1394 standard as closely as possible, some simplifications have been made. The more abstract descriptions of Sections 4.3 and 4.4 make many assumptions, while the more concrete descriptions of Section 4.5 and especially the timed description of Section 4.6 make fewer.

In the standard, each node in 1394 has a number of parameters, as does each port (via which a node is connected to another node). Rather than copying these parameters exactly, we choose the most appropriate representation of the data for our modelling technique. For example, each node has a parameter `root` which is set if the node is the current root and a parameter which indicates the current parent. To model this, the descriptions have a set of neighbours which is reduced through communications with connected nodes until either one element is left (which is the parent) or no elements are left (therefore the node is the root).

It is also necessary to add an identification number for each node in the system. This is used to control communication and to model network configuration, whereas in the real system the knowledge of node identity and network configuration is encoded in the physical wiring of the network.

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 in the standard 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 description of Section 4.3 completely ignores these communications, the description of Section 4.4 (with synchronous message passing) ignores the two acknowledgements, and the descriptions of Section 4.4 and Section 4.6 both model only the "be my parent" request and the corresponding acknowledgement. The third message, checking the child has received an acknowledgement, is ignored since the message passing medium is assumed to be reliable (no messages are lost or corrupted).

In addition to the messages in the system regarding the parent-child relationship we add a "leader" message: this is sent by the elected leader to indicate that a leader has been chosen. This provides us with a means of checking the requirement that a single leader is eventually elected. In the real protocol, all nodes simply move on to the next stage of execution, self identification. This phase has been referenced as a process here (`SelfID`) but not described since only the tree identify protocol is of interest.

Other important assumptions concern the network configuration. For example, while the timed description of Section 4.6 checks for loops in the network, the two untimed descriptions (Section 4.4 and Section 4.5) assume that the network is without loops. All descriptions will only operate correctly if the network is a single connected component; the opportunity to describe networks with disconnected components is an artificial by-product of the formal description that does not arise in the standard.

Before the descriptions are presented, the module system of E-LOTOS and the data types required for this system are introduced.

4.1 Modules

A disadvantage of LOTOS is the inability to modularise descriptions. This has been corrected in E-LOTOS. The module system is illustrated here by describing the data types as one module (which is shared by all descriptions), and the processes as another module. To simplify the presentation here only the process definitions have been given in each of Section 4.4 to Section 4.6. An example is given below of how these should be wrapped up in a module for one description. The rest are similar.

```
module  FireWireProcsUTSynch is
  (* process definitions of Section 3.4 *)
endmod
```

In E-LOTOS, comments are enclosed within `(*` and `*)`.

With the above module description and that of `Types` (see Section 4.2) the full specification of the FireWire with no time and synchronous message passing is then

```
specification FireWire2 import Types, FireWireProcsUTSynch is
  gates read, leader
  behaviour
  var
    P:conntable
  in
    (* input the network configuration from an external source *)
    read (?P);
    FireWireUTS [leader](P)
  endvar
endspec
```

The `conntable` variable `P` is set up in the specification to model the initial configuration of the network. This 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 model a network with three nodes `P` might be `cons((0,{1,2}), cons((1,{0}), cons((2,{0}), nil)))`. Node 0 is connected to both node 1 and node 2, node 1 is only connected to node 0, as is node 2. It is convenient to assume that this list has no duplicates, as the auxiliary function `neighbours` does, but E-LOTOS allows more complex functions to be written to handle the case with duplicates. The `conntable` type would have been more appropriately described as an array, but arrays are not currently part of E-LOTOS.

The descriptions here do not greatly lend themselves to modularisation since they are largely disjoint; the facilities of abstraction and encapsulation which allow parameterisation and reuse of modules have therefore not been exploited here.

4.2 Data Types

One of the main drawbacks of using LOTOS is that the ACT ONE data language is fairly user-unfriendly. Each data type has to be described equationally, in the style of an abstract data type, which can be tedious, even for someone used to this style of specification. In E-LOTOS data types are described functionally and standard types such as lists and sets are part of the language.

The data types described in below are all very straightfoward, and make use of the built-in list, set, record and disjoint type constructors.

```
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 commtype is parent | ack endtype

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

The functions described here illustrate the use of pattern matching on list constructors. Note the use of `!` to denote a value, or used occurrence of a variable, and `?` to denote a binding occurrence of a variable. This can also be seen in the events of `FireWireUTS`, `FireWireUTA` and `FireWireTA`.

```
(* Function definitions for the FireWire descriptions. *)
  (* 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):bool 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
```

4.3 Abstract specification

The most basic description of the protocol merely announces that a leader has been chosen and terminates. The network is viewed as a whole, and no communication between nodes is described. This process captures the basic verification requirements that only one leader is chosen, and that a leader is eventually chosen. It is not important which node is chosen as a leader; each node is equally likely to become the leader (but see Section 4.6 where the `FORCE_ROOT` parameter is introduced).

The E-LOTOS description is given below. This should all be familiar to the LOTOS user, but note that E-LOTOS actions are given the same status as processes.

```
process FireWireSpec [leader] is leader endproc
```

4.4 Untimed description with synchronous communication

In this description, each node in the network is modelled by a separate process, and the whole is given by the parallel composition of these nodes. Communication between nodes is synchronous, i.e. a message is received and sent simultaneously, therefore there is no need for acknowledgements (unlike the standard and the next two descriptions).

Individual nodes in the system are described in E-LOTOS as follows.

```
process TreeID [leader, c:comm] (id:iden, p:connections) is
loop
var
  j : iden
in
  case true is
    !isempty(p)   -> leader; break
    !singleton(p) -> ?j := any id [isin(j, p)];
                     (   (* branch node, so send a parent request to j. *)
                         c (!id, !j, !parent);
                         break
                      [] (* root node, so receive a parent request from j *)
                         c (!j, !id, !parent);
                         ?p := {};
                         break )
    any:bool      -> (* accept any incoming "be my parent" request *)
                     c (?j, !id, !parent)[isin(j, p)];
                     ?p := diff(p,{j})
  endcase
endvar
endloop;
SelfIDA [leader,c](id,p)
endproc
```

In each node there are three possible actions (captured in a case statement, a new feature of E-LOTOS). Here the cases match against the value `true`, which gives the effect of guarded commands. Starting with the last case (the "any", or default case), typically a node communicates with its neighbours to say "you are my child" and the parameters are adjusted accordingly. Again, note the use of `?` to bind a variable (i.e. an "input") and `!` to denote a value bound previously (an "output"). In the middle case, there is only one neighbour left with whom to communicate therefore the node may make either a "you are my child" or a "be my parent" communication. The `any` wildcard is usually used in an assignment to non-deterministically choose an element of `p`, but it is known that there is only one element therefore the choice is deterministic here. Non-determinism is given by the use of the `[]` operator where both actions are on the same gate. Finally, in the top case, if a node has communicated on all its ports to establish children, then it must be the leader. In E-LOTOS, the order of evaluation of cases is strictly top to bottom, which is why the singleton case has to include a non-deterministic choice between sending and receiving a message, otherwise the system would deadlock. This is also the reason for putting the `any` case last.

The case statement is enclosed within a loop construct, broken by the `break` command. This is another imperative feature added to E-LOTOS, as is the inclusion of local variables and assignment statements, both of which can be seen in the above specification. These are intended to make the language more appealing to programmers. They are extremely useful in the specification here, although a different approach (a more LOTOS style approach, using recursion) is illustrated in `FireWireUTA`.

The individual nodes as specified are connected in a network of three nodes by the following behaviour expression.

```
process FireWireUTS [leader] (P:conntable) is
  hide c:comm in (
    par c#2 in
        [c] -> TreeID [leader,c](0, neighbours(P,0))
     || [c] -> TreeID [leader,c](1, neighbours(P,1))
     || [c] -> TreeID [leader,c](2, neighbours(P,2))
    endpar )
  endhide
endproc
```

The most important E-LOTOS change illustrated by this description is the ability to specify communication of some processes amongst a network of many processes. For example, in the process `FireWireUTS`, communication on the gate `c` is specified as having degree 2. This means that exactly two processes communicate on this gate at any one time, although all the processes in the expression (three here) have the ability to transmit on this gate. It is not possible to describe this "*n from m*" (where $n < m$) style of synchronisation in standard LOTOS. Standard LOTOS would insist that either all three processes simultaneously communicate on the gate, or that no processes communicate on the gate (i.e. interleaving the actions).

Here the data communicated on the gate specifies exactly which two nodes are to synchronise (implementing the network configuration). The inability to express this sort of communication has long been recognised as a shortcoming of LOTOS [KT97, Bol90], which is addressed in other process algebras, e.g. μCRL [GP95].

Unfortunately in E-LOTOS it is not possible to neatly specify the combination of n nodes, where n is specified elsewhere. In this particular case the existence of three processes in the network must be hardwired into the description. This problem does not occur in the next description (because the `TreeID` processes there are interleaved).

4.5 Untimed description with asynchronous communication

Although the previous description should be easy to understand, it is not a completely accurate depiction of events in the real protocol because communication is actually asynchronous. It takes time for a message to pass from one node to another. In particular, two nodes might simultaneously request that the other is their parent, leading to root contention (each wants the other to be the root node).

Asynchronous communication is modelled by making `TreeID` processes talk to each other via one place buffers. Larger buffers are not necessary; for each pair of nodes and direction of message passing, only one message can be sent at any time. The nature of communication in E-LOTOS (`?` and `!` data can synchronise in any combination) means that the communication between nodes and buffers must be split into two actions (to stop the intended input of a buffer from comunicating with the receiving node rather than the sending node).

The `Buffer` itself describes receiving a message from node `id` to node `j` of type `m` (a communication is either a parent request or an acknowledgement) and then passing that message on to node `j` as coming from node `id`.

```
process Buffer[ip:comm,op:comm](id:iden, j:iden) is
loop
  var
    m:commtype
  in
    ip (!id, !j, ?m);
    op (!id, !j, !m)
  endvar
endloop
endproc
```

The combination of n^2 parallel instantiations of `Buffer` is more novel:

```
process Buffers[ip:comm,op:comm](n:nat) is
var
  l:list of nat
in
  ?l := (0 upto (n-1));
  par ?id in l
    ||| par ?j in l
          ||| Buffer[ip,op](id,j)
        endpar
    endpar
endvar
endproc
```

The `Buffers` process uses parallel over values to specify concisely a network of n^2 `Buffer` processes (one for each combination of the values of `id` and `j`). The actions of these processes are interleaved (the buffers do not talk to each other). This kind of expression is not possible in LOTOS (but is a feature of μCRL). Note that this will generate too many buffers, because we know that not all nodes are connected to all other nodes. Describing exactly the buffers needed would be too cumbersome and confusing.

The parallel over values can also be used to describe n interleaved instantiations of the nodes `TreeID`, and the buffers communicate with the nodes with degree 2. This is exactly what we want here. Data is used to constrain communication (so that the *right* buffer talks to the *right* node). This is the reason that the extra buffers generated above will simply remain idle.

This is a much more elegant way of describing the system than having to hardwire the number of processes into the behaviour expression (as in `FireWireUTS` and as normally required in LOTOS).

```
process FireWireUTA [leader] (P:conntable) is
var
  n:nat; l:list of nat
in
  ?n := sizeof(P);
  ?l := (0 upto (n-1));
  hide r:comm,s:comm in (
    par r:comm#2,s:comm#2 in
         [r,s] -> (par ?x in l
                   ||| TreeID [...] (x, neighbours(P,x),{})
                 endpar)
      || [r,s] -> Buffers[r,s](n)
    endpar)
  endhide
endproc
```

Lastly, the TreeID process is described. The protocol is considerably more complicated here because of the need to send acknowledgements to children, and the possibility of contention, therefore it is divided up into phases. Each phase of the protocol is described as a separate process. A new parameter ch records the children to whom acknowledgements must be sent.

In the first phase, TreeID, the node receives parent requests from children up to all but one of its connections.

```
process TreeID [leader, r:comm, s:comm]
                (id:iden, p:connections, ch:connections) is
var j : iden in
  case true is
      !isempty(p)   -> leader; SelfIDB [...](id,p,ch)
    | !singleton(p) ->
           (   (* branch node, so either send first child ack to
                  some j in ch, or if no children send parent request. *)
               case true is
                   !isempty(ch) -> s (!id, ?j, !parent)[isin(j, p)];
                                   WaitParent [...](id,p,ch)
                 | any:bool     -> s (!id, ?j, !ack)[isin(j,ch)];
                                   SendAcks [...](id,p,diff(ch,{j}))
               endcase
            [] (* root node, so receive a parent request from j,
                  and then go into sending ack mode *)
               r (?j, !id, !parent)[isin(j, p)];
               SendAcks [...](id, diff(p,{j}),union(ch,{j})) )
    | any:bool   -> (* accept any incoming "be my parent" request *)
                     r (?j, !id, !parent)[isin(j, p)];
                     TreeID[...](id,diff(p,{j}),union(ch,{j})
  endcase
endvar
endproc
```

Next the node enters the acknowledgement phase, SendAcks, where it sends acknowledgements to all its children.

```
process SendAcks [leader, r:comm, s: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).          *)
                         s (!id, ?k, !parent)[isin(k, p)];
                         WaitParent [...] (id, p)
    | !singleton(ch)  -> (* make last ack and if empty(p) send leader *)
                         s (!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 *)
                         s (!id, ?j, !ack)[isin(j,ch)];
                         SendAcks[...](id, p, diff(ch,{j}))
  endcase
endvar
endproc
```

Lastly, the node sends a parent request to the remaining connection, if there is one, and awaits an acknowledgement, the process `WaitParent`. Should a parent request be received instead then both nodes involved move into contention. Contention is resolved (eventually) through non-determinism in this description (a different solution appears in the timed description).

```
process WaitParent [leader, r:comm, s: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)];
  (  r (!j, !id, !ack); SelfIDT[...](id, p)
  [] r (!j, !id, !parent); Contention[...](id, p))
endvar
endproc

process Contention [leader, r:comm, s:comm]
                   (id:iden, p:connections, ch:connections) is
var j:iden in
  (* Note there is only one j in p *)
  ?j := any iden [isin(j,p)];
  (   s (!id, !j, !parent); SendAcks[...](id, diff(p,{j}), union(ch,{j}))
   [] r (!j, !id, !parent); WaitParent[...](id,p,ch))
endvar
endproc
```

The bodies of the processes are rather similar in style to `TreeID` of the previous description although clearly this time there are more phases to deal with and different choices to make. The case behaviour could be replaced by an equivalent `if _ then _ else` behaviour but that is really a matter of stylistic choice for the specifier. Similarly, the use of recursion rather than iteration is also a stylistic choice; it is felt to give a simpler description in this case because several processes are involved (there is not the same clean loop break and call of `SelfID` as in `FireWireUTS`).

E-LOTOS allows the shorthand `[...]` for gate parameters, where the actual parameters are the same as the formal parameters. This is a great convenience over LOTOS.

4.6 Timed description with asynchronous communication

One of the major changes to E-LOTOS is the introduction of real time. LOTOS has a notion of ordering in time, but no quantitative time. E-LOTOS adds the ability to specify delay and communication which is sensitive to delay. This allows for the most realistic description of the protocol of the four presented here.

The problem with the earlier descriptions is that they cannot model the first stage of the protocol accurately because it depends on some time values. Whereas the previous two descriptions had a non-deterministic choice between sending or receiving a "be my parent" request when the number of connections is one, the actual 1394 normally moves onto the `SendAcks` stage when $n-1$ children have communicated "be my parent" (where n is the number of neighbours). Abnormal cases are a timeout (`CONFIG_TIMEOUT`), which means the network contains a loop, or when the parameter `fr` (`FORCE_ROOT` of the standard) is set. `FORCE_ROOT` makes the process wait for a longer time before checking the $n-1$ condition, thereby allowing a greater chance of receiving all child communications (so the node is more likely to be the root).

In the following several time constants are used. The particular choice of values is discussed later in this section.

The `Buffers` process is the same as in `FireWireUTA`, but a delay is added to the `Buffer` process.

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

Delay is modelled by the addition of a non-deterministically chosen wait, using the wildcard `any` with a selection predicate to restrict the size of the delay. E-LOTOS allows that either dense or discrete time can be chosen. Here `time` is a synonym for `nat` and is measured in nanoseconds.

The main combining process is also similar to that of `FireWireUTA`, but calling `TreeID` with the new parameter `fr` which is the force root flag. Here all values of `fr` are set to false for convenience. An initial configuration for the `fr` parameters could be set up in the same way as the initial network configuration is set up, and that would allow one of the `fr` to be set to true in the top level specification to make that node be the root.

```
process FireWireTA [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 r:comm,s:comm in
    par r:comm#2,s:comm#2 in
         [r,s] -> par ?id in l
                   ||| TreeID [...](id, neighbours(P,id), {}, false)
                                   [LoopException]
                  endpar
      || [r,s] -> Buffers[r,s](n)
    endpar
  endhide
endvar
endproc
```

Another feature familiar from programming languages is used here: the exception. `LoopException` is raised inside `TreeID` when a loop is detected in the network, and trapped by a main specification slightly modified from the general one given in Section 4.1.

```
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 FireWireTA[leader](P)[LoopException]
                           else raise ConnectionError endif
      endif
    endtrap
  endvar
endspec
```

The TreeID process is also modified by the inclusion of time. We add two alarm clocks to deal with the timing constraints FRTIME and CONFIG_TIMEOUT.

```
process TreeID[leader, r:comm, s: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

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

ReceiveReqs then uses these alarms to control communication.

```
process ReceiveReqs
        [leader,r:comm,s: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 *)
     r (?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
```

The condition on the cases in `TreeID` are quite complicated. The basic case, the first one, describes normal inter-node communication. In the second case the node stops receiving communications and moves to the `SendAcks` phase when the number of children reaches `n` or `n-1`. The alarms are deactivated. Note that when `fr` is set the check is for `card(ch) = n`, i.e. we can only move on to `SendAcks` if *all* connected nodes are children. The third case says that if we've waited long enough (by the definition of the standard) then we allow that even a node with `fr` set initially can move on with `card(ch) = n-1`, so we reset `fr` to false. Lastly, `CONFIG_TIMEOUT` may be reached, in which case there is a loop. The extra `if` statement takes care of the case when the alarm goes off just after receiving `n-1` communications (so the node was about to move into the next phase).

The `SendAcks` and `WaitParent` phases in `TreeID` are exactly the same as in `FireWireUTA` and so are omitted in the presentation. Only `TreeID` and `Contention` are affected by the introduction of time.

In the standard contention is resolved by examining a random Boolean. The value of the Boolean determines how long the process waits before trying to communicate with the other node.

```
process Contention [leader, r:comm, s:comm]
                   (id:iden, p:connections) is
(* There will be only one j in p.
   This means that both nodes (id 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); s (!id, !j, !parent); WaitParent[...](id, p)
  [] r (!j,!id,!parent); SendAcks[...](id, {}, {j}) )
endvar
endproc
```

To complete the description it would also be required to declare some constants of type time. 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.

When putting these definitions all together we come across a problem: in E-LOTOS, internal actions are urgent, which means that all hidden actions are urgent. Since the gates used for communication here are hidden, this means that effectively the only delays in the specification are those which occur because of the length of time it takes to send a message down a wire. These delays are small enough to be negligible when compared with the `CONFIG_TIMEOUT` value, so what is missing? In real life there will be other delays related to the length of time each node takes to inspect the state of its ports, and the time taken to execute the implementation. 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.

5 Conclusions

We have presented a case study in the use of E-LOTOS. We described a system at several different levels of abstraction and exploited some of the new features of E-LOTOS (in particular, the new synchronisation constructs, and the addition of time). Of course not all elements of E-LOTOS appeared in this case study. We did not exploit the new data type features to a great extent, because we required fairly simple data types. Neither did we exploit the full module system; again because it was not particularly appropriate in this case study.

The specification exercise is useful in itself because it shows some of the potential of E-LOTOS, shows it is possible to use it for a relatively complex communications protocol (which is to be expected since that was why it and LOTOS were developed), and demonstrates some facets of the language to new users. It fulfills the usual goals of providing a system description which is clear and unambiguous. The specification exercise forces the specifier to think about the system in completely formal way, which should lead to greater understanding of the system, and reduce the likelihood of errors.

We may ask something more of a specification, particularly if it is to be incorporated in an international standard, namely that it be easily *understandble* to the majority of users. This is a more subjective assessment of the specification and of a formal description technique. From the comments of the previous section it is clear that E-LOTOS is more like a programming language while other formalisms such as μCRL are more like mathematics. Similarity to mathematics can lead to shorter, more easily analysed specifications, but they may not be so understandable to the general user, and we would hope that automated tools would hide the complexity of analysis whatever the specification language. Therefore we would argue that similarity to programming languages is a desirable trait of a formal description technique which is to be used by a wide community.

While we did attempt to validate the correctness of the E-LOTOS descriptions 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 the language 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 for supporting this work, and to Ken Turner of the University of Stirling, whose idea the EASEL project was. Thanks also to Alberto Verdejo of the Universidad Complutense de Madrid for discussions regarding E-LOTOS, and to Frank Kelly of the University of Stirling for discussions regarding the timing aspects of IEEE 1394.

References

[Bol90] T. Bolognesi. A Graphical Composition Theorem for Networks of LOTOS Processes. In *Proc. of the 10th International Conference on Distributed Computing Systems, Washington, USA*, pages 88–95. IEEE, 1990.

[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.

[GP95] J.F. Groote and A. Ponse. The Syntax and Semantics of μ-CRL. In *Proceedings of Algebra of Communicating Processes, Utrecht 1994*, Workshops in Computing. Springer-Verlag, 1995.

[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.

[KT97] C. Kirkwood and M. Thomas. Towards a Symbolic Modal Logic for Full LOTOS. In A. Evans and D. Duke, editors, *Proceedings of the Northern Formal Methods Workshop*, Electronic Workshops in Computing, ISBN 3-540-76117-9. Springer-Verlag, 1997.

[LFHH92] L. Logrippo, M. Faci, and M. Haj-Hussein. An Introduction to LOTOS: Learning by Examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[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.