A case study in abstraction using E-LOTOS and the FireWire

Carron Shankland

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

Alberto Verdejo

Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid
`alberto@sip.ucm.es`

Abstract

The proposed ISO standard E-LOTOS is used to describe the leader election protocol of the IEEE 1394 serial multimedia bus ("FireWire"). The E-LOTOS language facilitates descriptions at several levels of abstraction, therefore a broad understanding of the protocol at an abstract level can be gained before adding more complexity and concrete detail. A secondary aim is to illustrate some of the novel features of E-LOTOS, particularly those for describing real time and parallelism. The specification process is described in some detail, paying particular attention to the general applicability of E-LOTOS to protocol description. Verification techniques and tools for E-LOTOS are considered, and comparisons are drawn between E-LOTOS and other similar formal methods.

Key words: E-LOTOS (Enhancements to LOTOS), IEEE 1394, Leader Election Protocol, Formal Methods.

1 Introduction

Formal Methods are increasingly becoming part of the system development process. The advantages of using a precise, unambiguous formal description technique amenable to formal analysis are clear. As more use is made of formal approaches, their shortcomings when applied to larger and more realistic systems become evident, and new formalisms are developed to address these problems. With this ever-increasing variety of formalisms it is not easy to keep up with developments or to decide which to choose for a particular application. Case studies have an important role to play in assessing the capabilities of a new formalism, in disseminating information about it, and in illustrating its use in practice.

In that spirit, we present here a case study using *E-LOTOS* [13] to describe the leader election protocol of the IEEE 1394 standard for a serial multimedia bus [10]. The main emphasis is on specification; illustrating the operators provided by E-LOTOS, particularly parallelism and time, and the range of abstraction levels available to the specifier. Abstraction is a key element of specification, since usually an abstract specification is more easily understood, is smaller and therefore easier to analyse.

E-LOTOS [13] (Enhancements to LOTOS) is a new formal description technique which is reaching the end of the ISO standardisation process. It was developed from the ISO standard formal description technique LOTOS [12], which allows the specification of behaviour and data using process algebra and abstract data types. LOTOS has been applied to many varied systems over the last fifteen years; for example, OSI protocols, avionics software, plain old telephone systems, feature interactions and even childrens' games. The experience gained in these applications informed the development of E-LOTOS, which has additional features intended to make it more useful to software engineers describing larger, timed, computer systems.

The leader election protocol of the IEEE 1394 standard for a serial multimedia bus (aka FireWire) was chosen for three reasons. Firstly, it is a standard in communications and a number of companies including Apple, Microsoft, Philips and Sun were involved in its development. Secondly, this reasonably complex example tests the expressivity of E-LOTOS. E-LOTOS allows the description of this protocol to be made in a modular fashion, breaking down phases into processes, and separating the data type specifications from the process specifications. Further, the system can be described at various levels of abstraction, reflecting different design choices in the modelling process. This range of descriptions allows several of the novel features of E-LOTOS to be demonstrated, particularly those involving parallelism and real time. In this way the case study can serve as an introduction to the main features of E-LOTOS for experienced LOTOS users and newcomers alike. Thirdly, this example has been described elsewhere using a variety of different techniques and is becoming something of a benchmark for formal methods.

The structure of the paper is as follows. An overview of the development and main features of E-LOTOS is given in Section 2. The case study is presented in Section 3. An informal introduction to the example is given in Section 3.1, particularly concentrating on the main modelling requirements this example places on a formal description technique. This is followed by a discussion of the particular design decisions taken for specification of the example in E-LOTOS. The bulk of the paper comprises four E-LOTOS descriptions of the leader election protocol at varying levels of abstraction; beginning with a very simple specification, moving through increasingly complex descriptions which build understanding of how the protocol works, and which illustrate different aspects of specification using E-LOTOS. The final specification is a timed version which models the system of the IEEE standard reasonably closely. In Section 4 we consider issues of formal analysis of E-LOTOS specifications. The capabilities of E-LOTOS as a specification language are examined throughout, but are particularly compared with previous experience of LOTOS and with other approaches to this example in Section 5. Finally some conclusions are drawn about the E-LOTOS experience.

2 Introduction to E-LOTOS

E-LOTOS maintains the strong formal basis of its predecessor LOTOS (which was in turn based on CCS [15] and CSP [8] for the behavioural part, and ACT ONE [5] for the data part). LOTOS was developed to describe distributed systems and communications protocols. E-LOTOS retains that focus, adds greater expressivity and structuring power, and is more user-friendly. E-LOTOS has many of the features of LOTOS such as events, processes, and operators for sequencing and choice; we assume the reader is familiar with these concepts. A more in-depth introduction to LOTOS notation may be found in [14].

The most important enhancements introduced in E-LOTOS are:

- the notion of quantitative time: in E-LOTOS we can define the exact time at which events or behaviours may occur (illustrated in Sections 3.6 and 3.7);
- a new data language: data types similar to those of (functional) programming languages are provided (Section 3.3), maintaining formal support;
- modularity: allowing the definition of types, functions, and processes in separate modules, control of their visibility by means of module interfaces, and definition of generic modules, useful for code reuse (Sections 3.3, 3.4 and 3.7);
- new operators which improve the expressive power of the language: specifically, operators to deal with exceptions (Section 3.7), and ones to represent complex parallelism schemes in an easy way (Sections 3.5 and 3.6);
- some constructs from imperative programming languages (loops, if-then-else, case, etc): making the language useful for covering the last steps of the software life cycle, when implementations are developed, and making the job of specifying systems easier (Sections 3.3 to 3.7).

- *write-many* variables: that is, variables that can be assigned several times as in imperative programming languages (Sections 3.5 to 3.7).

Abstraction was already a key part of the LOTOS language; it is the basis of the idea of verification of a system by proving some relation (for example, bisimulation) between an abstract specification-oriented description ("what" the system does) and a more concrete implementation-oriented description ("how" the system achieves it). What has been added in E-LOTOS is the ability to take the concrete implementation level a step nearer to the considerations of the real world, through the addition of new imperative operators and timed operators.

The semantics of E-LOTOS is given in terms of labelled transition systems (as is common for process algebras). Two main concepts have influenced the semantics: variables and time. It is beyond the scope of this paper to present the formal semantics of E-LOTOS; see the tutorial material of [20] or that appended to the committee draft standard [13]. Instead, we try to give an informal understanding of the semantics, and of the way it affects the new operators of E-LOTOS, using the examples of the next section for illustration.

In a language with communicating processes, variables must be handled carefully, to ensure data consistency. The E-LOTOS static semantics ensures that variables cannot be read before a value has been assigned, and that behaviours running in parallel assign to disjoint sets of variables, that is, communication between processes must be made explicitly through actions. In other words, there is no shared memory, just as in a distributed system, for example.

E-LOTOS provides three specific features to manage time. Firstly, the specifier can describe *when* an action should occur. Secondly, the `wait` instruction (Section 3.6) describes a duration d in which the process must idle. Thirdly, the type `time` is loosely specified in the E-LOTOS semantics, leaving the precise implementation of time to the specifier, provided it forms a commutative, cancellative monoid, and there is a total order among time values.

The semantics of E-LOTOS is split into timed and untimed transitions; however, the two kinds of transition interact in subtle ways, affecting communication between processes and progress of the system, even for untimed operations. A process may *let time pass*, that is, be doing nothing but waiting, even if actions are available for it to perform. To ensure that processes do not idle forever, that is, that the system can *progress*, the notion of *urgency* must be introduced. An *urgent* action must be performed as soon as possible; that is, if an urgent action is enabled it must be performed unless some other action can be performed without consuming time. The special internal action `i` is urgent (`i` in LOTOS is like τ in CCS), therefore by hiding actions they become urgent and the evolution of the system is ensured. The specification of Section 3.5 illustrates how time affects a system of (untimed) communicating components, and the specification of Section 3.6 illustrates how time, and more specifically *timeout*, can be used to delay a choice.

In E-LOTOS time is *deterministic*, that is, when a process P is idle for some duration d the resulting behaviour is completely determined by P and d [16]. The avoidance of time nondeterminism has affected several operators; for example, in a choice `B1 [] B2` each branch must be guarded (and hence subsequent assignments guarded) to avoid time nondeterminism. The static semantics ensures that time is always deterministic.

A more detailed treatment of E-LOTOS may be found in [9], where the main implications of the inclusion of these elements are explained, and some of the alternatives presented during standardisation are discussed. In the remainder of the paper we present some of these topics informally, via descriptions of the Tree Identify Protocol of the IEEE 1394 standard.

3 E-LOTOS in action

3.1 Informal Overview of Tree Identify Protocol

The running example of this paper is drawn from the serial multimedia bus standard IEEE 1394 [10]. The bus carries all forms of digitised video and audio between connected systems and devices. The architecture is "hot-pluggable", so systems and peripherals can be added or removed at any time. This means that the bus manager must be reassigned every time the network is reconfigured by a leader election protocol, namely the Tree Identify Protocol of the PHY (physical) layer.

The Tree Identify Protocol takes place after a bus reset in the network, that is, 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. 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 (solid connection with arrow pointing to the parent). The exact details of the negotiations are described in the following sections, as we move from a high level abstract description of the system, to a relatively low level description close to the implementation of the standard. When all negotiations are concluded, the node which has established that it is the parent of *all* its connected nodes must

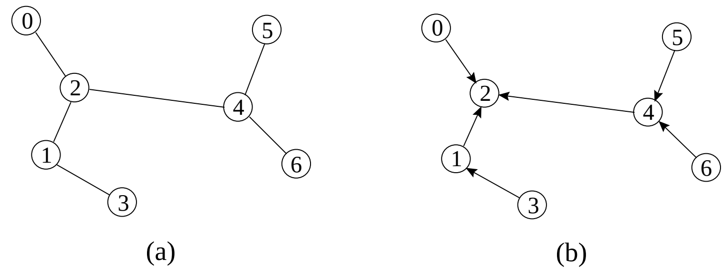

Fig. 1. Network Configurations Before and After Leader Election

be the root node of a spanning tree of the network. See Figure 1(b) in which node 2 is the root node. The protocol is only successful if the initial network has no loops. Loop detection is implemented in the standard and modelled in our most concrete description (in Section 3.7).

The IEEE standard is long, over 300 pages, and the description of the leader election protocol is distributed throughout the document. Further, it is given using a mixture of notations at varying levels of formality: state machine notation, informal text, C code, illustrative examples, and tables of values for the constants used throughout. The first task, therefore, of the modelling process is to ascertain exactly which parts are necessary for the formal description; this in turn depends on the chosen level of abstraction and on the modelling features supplied by our formal description technique. In fact, there are several ways in which to model the protocol, so a further challenge is to see how many of those different views E-LOTOS can accommodate. In other words, how expressive is E-LOTOS?.

For example, leader negotiations are carried out by various messages being passed between the nodes of the network. The network could be modelled by its state, ignoring all details of how nodes are connected. Alternatively, and more naturally for a process algebra, we view the network as a set of communicating entities; either as nodes and edges, or more concretely as a set of devices and their ports. Communication in the system can be modelled in different ways. In reality, messages between nodes are voltages on wires, but in a process algebra a natural way to model each message is as a separate *event*. For example, at the most abstract level all internal communication can be ignored, and a leader simply declared. Alternatively, internal communication can be synchronous or asynchronous. Asynchronous message passing (which is most realistic) brings further modelling choices. If messages are passed asynchronously then *contention* may arise. This is a situation in which two nodes have both requested that the other be their parent. Resolution of contention can be modelled abstractly (by nondeterminism) or more concretely (using time). Time is also used in loop detection (via the CONFIG_TIMEOUT parameter) and to affect leader election (via FORCE_ROOT parameter).

E-LOTOS has sufficient expressive power to describe all these variants of the protocol, as is demonstrated in the rest of this section.

3.2 General Design Decisions

Four different versions of the Tree Identify Protocol are considered in Section 3.4 through to Section 3.7. These are *informal* refinements of the system; there is no formal refinement procedure for E-LOTOS at present. Our aim is to show the application of E-LOTOS in building an increasingly accurate representation of the system, culminating in the timed description of Section 3.7. Thus many simplifying assumptions are made about the system in the more abstract descriptions of Sections 3.4 and 3.5, while the more concrete descriptions of Section 3.6 and 3.7 make fewer. Specific details are discussed in the relevant sections; here we describe basic modelling decisions which can be applied to protocol description generally.

We take advantage of the module system by describing the data types as one module (shared by all specifications), the processes specifying the protocol behaviour as another, and the time constants as a third. E-LOTOS offers further parameterisation and reuse facilities via abstraction and encapsulation [9,20], but this example does not lend itself to such further modularisation.

The main entities of E-LOTOS are data types, processes and events. The first modelling decision to make is how to relate these to the entities in the protocol. The most basic action is the "leader" message: this is sent by the elected leader to indicate that a leader has been chosen, so we know the protocol is complete. This is in fact an artefact of the modelling process, since in the real system all nodes simply move on to the next phase, self identification. This phase has been referenced as a process here (`SelfID`) but not described since only the tree identification phase is of interest.

Looking at the protocol in more detail, we have nodes and communications between nodes, relating naturally to processes and events, respectively. Each node in the network has a number of ports via which it is connected to other nodes. These physical connections describe the network configuration. To model this we give each node process an identification number which is used when describing the network configuration and also to control communication. Data types can be used to model internal parameters of nodes and to distinguish different kinds and directions of communications. For example, the difference between a request and an acknowledgement, or a request from node 1 to node 2 and a request from node 2 to node 1.

Various messages are sent between nodes. 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).

Not all of these messages need be modelled. For example, the description of Section 3.4 completely ignores messages between nodes, while the description of Section 3.5 ignores the two acknowledgements since they are redundant in the setting of synchronous message passing. The descriptions of Section 3.6 and Section 3.7 both ignore the second acknowledgement, since the message passing medium is assumed to be reliable (no messages are lost or corrupted). For this exercise we chose not to model faulty media, and the Tree Identify Protocol of the IEEE 1394 standard also ignores this possibility, but such media are easily described in E-LOTOS.

Each node and port also has a number of parameters. It is possible to describe these exactly, but it is useful if the modelling technique allows us to choose a more compact representation. For example, in the standard each node has a parameter `root` which is set if the node is the current root and a parameter which indicates the current parent. The descriptions here have a set of neighbours which is reduced through communication with connected nodes until either one element is left (which is the parent) or no elements are left (therefore the node is the root).

Lastly, we may make some assumptions about the environment in which our specification operates. For example, while the timed description of Section 3.7 checks for loops in the network, the earlier descriptions of Section 3.5 and Section 3.6 assume that the network is without loops. Also, the protocol 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 modelling process that does not arise in the standard. In the description of Section 3.7 a special check for connectedness is added to counteract this artefact of the modelling process.

3.3 Data Types

The part regarding the declaration and use of data types is one of those that has been changed most in E-LOTOS with respect to its predecessor LOTOS. The LOTOS abstract data type specification language (also known as ACT ONE [5]) is rather unfriendly and suffers from limitations such as semi-decidability of equational specifications, lack of modularity, and inability to define partial operations. In E-LOTOS, ACT ONE has been replaced by a new language in which data types are declared and used in a similar way to that in functional languages, with some additional operations drawn from imperative languages (such as assignment statements). The problems mentioned above are resolved, producing a data type language which is easier to use.

A function is any process with the following characteristics: it is deterministic;

```
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 network 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. 2. Data Types for the FireWire Descriptions

it cannot communicate, that is, it has no gates, and so its only capabilities are to return values and raise exceptions; and it has no behaviour over time. Semantically, functions and processes are unified, therefore expressions are reduced by using the same operational semantic rules as those for behaviours, and equality is syntactic: two different expressions represent the same value if, as behaviours, they return the same syntactic value.

The data types for the FireWire descriptions are given in Figure 2. Comments are enclosed within `(*` and `*)`. The highlights of this specification are:

- Standard types such as integers, Booleans, lists and sets are part of the language. For example, 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 of Sections 3.5 to 3.7.
- The network configuration is modelled as a list of pairs, where the first element indicates the node identifier and the second element the set of connected nodes. For example, the network of Figure 1 is given by `[(0,{2}), (1,{2,3}), (2,{0,1,4}), (3,{1}), (4,{2,5,6}), (5,{4}), (6,{4})]`. For convenience the `network` is ordered here but this is not essential to correct operation of the specification.
- New user-defined data types can be declared by enumeration of all the constructors, as in `conntype`. Constructors may also have data arguments.
- The record type `(iden, iden, conntype)` is used to model communications in the system by giving the source and destination node identifiers and also the message type.

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

  (* Returns the number of nodes in the network, *)
  (* assuming no duplicates in nt.              *)
  function sizeof(nt:network):nat is
    case nt is
        nil -> 0
      | cons(any:pair, ?ns) -> 1 + sizeof(ns)
    endcase
  endfun

  (* Checks that all nodes are connected to at least one node. *)
  (* Networks of size one are a special case.                  *)
  function connected(nt:network):bool is
    if sizeof(nt) = 1 then true
    else
      case nt is
          nil -> true
        | cons((?n, ?c), ?ns) -> if empty(c) then false
                                 else connected(ns) endif
      endcase
    endif
  endfun

  (* Extracts the list of connections associated with index x, *)
  (* assuming no duplicates in nt.                             *)
  function neighbours(nt:network,x:iden):connections is
    case nt is
        nil -> {}
      | cons((?n, ?c), ?ns)  -> if x = n then c
                                else neighbours(ns,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. 3. Function Definitions for the FireWire Descriptions

Of course, user-defined data types are of limited use without user-defined functions; these are given in Figure 3. The function definitions supplement the built-in set functions (`singleton`), manipulate the `network`, and define a list generating function. Note the use of a `case` statement to pattern match on list constructors and the `infix` keyword to allow a more natural specification style when the function `upto` is used in the following process specifications. `any` is used as a wildcard, matching any value of the given type without producing bindings. Variables are introduced using `?` as in processes (see Section 3.5).

3.4 Specification

The most basic description of the protocol merely announces that a leader has been chosen and terminates.

```
process Spec [leader] is
  leader
endproc
```

The network is viewed as a whole, and no communications between nodes are specified. All features of the very simple E-LOTOS description above should be familiar to the LOTOS user, but note that in E-LOTOS actions are given the same status as processes, therefore a process can consist of a single action. `Spec` is parameterised by the action `leader` (denoted by the square brackets).

A seemingly trivial, but very useful, addition to E-LOTOS is the formalisation of the convention of using `[...]` to denote the gate parameters in a process instantiation where the actual gates identifiers are equal to the formal ones.

3.5 Synchronous Communication Description

Clearly, the description of the previous section is not very useful in terms of explaining to others how the leader is elected in a fair manner; the next step is to supply more information, making the description more concrete.

We view the network as a set of connected nodes, where each node has the same programming, and carries out the negotiations mentioned in Section 3.1 with its neighbours to establish the direction of the parent-child relationship between them. More specifically, if a node has n connections then it receives "be my parent" requests from all, or all but one, of its connections. Then, 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. Leaf nodes skip the initial receive requests phase; they have only one connection therefore

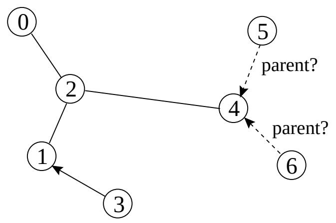

Fig. 4. Nodes make their "be my parent" requests

it must be their parent. To simplify this description, communication between nodes is assumed to be synchronous, that is, a message is sent and received simultaneously, therefore there is no need for acknowledgements. Figure 4 shows the instant when node 3 has its parent already decided (solid connection with arrow pointing to the parent), and nodes 5 and 6 are asking node 4 to be their parent (the queried relationship is shown by a dotted line).

To model the system at this level of abstraction, E-LOTOS must provide operators to combine a network of communicating nodes, allowing communication between pairs of nodes, and each node must have some internal parameters which can be used to make decisions about the flow of control (receive a "be my parent" request or send one). Therefore, in this section each node in the network is modelled by a separate process, and the whole is given by the parallel composition of these processes. Messages are modelled as events, and communication between processes in E-LOTOS is synchronous in any case.

Individual nodes in the system are modelled by the process `TreeIDSync`.

```
process TreeIDSync [leader, c:comm] (id:iden, p:connections) is
 loop
  var j : iden in
    if isempty(p) then
       leader; break
    else if singleton(p) then
           ?j := any iden [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 := {} )
         else
           (* accept any incoming "be my parent" request  *)
           (* from a connected node                        *)
           c (?j, !id, !parent)[isin(j, p)]; ?p := diff(p,{j})
         endif
    endif
  endvar
 endloop;
 SelfID [...](i,p)
endproc
```

Each node in the system is parameterised by:

- `id`, the identification number of the node, and
- `p`, the set of communications still to make. This is initialised to the set of all nodes connected to `id` and decreases with every "be my parent" request.

Some features here are standard. For example, processes are parameterised by gates (in the square brackets), and by data values (in the round brackets). Also familiar are sequencing of actions with processes (denoted by `;` in LOTOS and E-LOTOS), and choice (denoted `[]`). Note that in E-LOTOS the `;` notation is extended to include sequencing of processes. We now consider the novel features of E-LOTOS used in `TreeIDSync`.

In E-LOTOS gates can be declared together with the type of values they communicate. In this case, gate `c` communicates values of type `comm`. For compatability with LOTOS the type may be omitted, as with gate `leader`, meaning that this gate can communicate values of any type (or no value).

Only a single gate is used for communication, with data values indicating the kind of message transmitted. Each communication is parameterised by source node, destination node, and message type, for example, the communication `c (!id, !j, !parent)` is a message from `id` to node `j` saying "be my parent". The converse of this is the communication `c (?k, !j, !parent)` which is a synchronous acknowledgement from node `j` to some node `k` "you are my child". Data within communications may be restricted using a predicate, for example, `c (?j, !id, !parent)[isin(j, p)]` only allows communications with nodes `j` which belong to the set `p`. In processes `?` is used to bind a variable and `!` is used to denote a value (an expression possibly including free variables). These can be thought of as "input" and "output", but it will be seen in Section 3.6 that communication and synchronisation is more complex.

The actions of a node are determined by the number of neighbours it has to communicate with. There are three alternatives. Starting with the base case first, if the set `p` is empty then the node has communicated on all its ports to establish children and it must be the leader.

In the middle case, there is only one neighbour left with whom to communicate therefore the node may communicate either a "you are my child" or a "be my parent" message. These two offers are made available by means of the selection `[]` operator. Although the same gate `c` is used in both actions, the offered values are different, so, in principle, this is an external choice, that is, determined by the environment and not the process. If both actions are offered by the environment the choice would be made nondeterministically. Note here the use of assignable variables; we nondeterministically assign an element of `p` to `j` using the `any` wildcard. Since there is only one element in `p` the assignment is deterministic.

Finally, as long as there is more than one neighbour with whom communication is possible a node communicates with its neighbours in turn to say "you are my child" and the parameter `p` is adjusted accordingly.

The nested `if` statements are enclosed within a loop construct, broken by the `break` command. Although E-LOTOS is a specification language, the development committee felt that it was better to allow software engineers to describe systems in a familiar way, that is, using constructs from programming languages such as assignments, loops, `if` statements, `case` statements, and exceptions. The main benefit over using a programming language is, of course, that E-LOTOS has a formal basis, and is amenable to analysis.

The individual nodes as specified are connected together in parallel by the following behaviour expression. The construction, which uses the new E-LOTOS *general parallel operator*, is illustrated using the example network of Figure 1.

```
process SyncImp [leader] (NW:network) is
  hide c:comm in
    par c#2 in
       [c] -> TreeIDSync[leader,c](0, neighbours(NW,0))
    || [c] -> TreeIDSync[leader,c](1, neighbours(NW,1))
    || [c] -> TreeIDSync[leader,c](2, neighbours(NW,2))
    || [c] -> TreeIDSync[leader,c](3, neighbours(NW,3))
    || [c] -> TreeIDSync[leader,c](4, neighbours(NW,4))
    || [c] -> TreeIDSync[leader,c](5, neighbours(NW,5))
    || [c] -> TreeIDSync[leader,c](6, neighbours(NW,6))
    endpar
  endhide
endproc
```

As required, this specifies communication of some nodes amongst a network of many nodes. In this case communication on the gate `c` is specified as having degree 2, `c#2`. This means that exactly two processes should communicate on this gate at any one time, although all the processes in the expression (seven here) have the ability to transmit on this gate. The parallel construct does not specify *which* two processes are to communicate, but clearly only those nodes which are connected in the network configuration should be allowed to communicate. The data of the event is used to restrict synchronisation (events may only synchronise when both gate name and data coincide).

In standard LOTOS this "n from m" style of synchronisation (where $n < m$) is not allowed. Standard LOTOS would insist that either all seven processes simultaneously communicate on the gate, or that no processes communicate on the gate, that is, interleaving the actions. This has long been recognised as a shortcoming of LOTOS [1,6], and is a style of communication which is available in other process algebras, for example, CCS [15] and μCRL [7]. Now E-LOTOS

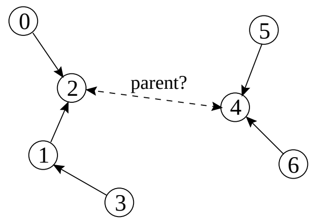

Fig. 5. Nodes make contentious "be my parent" requests

can describe two communication as easily as multiway communication.

Hiding is inherited from LOTOS, although in E-LOTOS the hidden gate can be typed. In `SyncImp` all the events on gate `c` will be hidden from the environment, therefore the only events observable in the environment are those on gate `leader`. Recall that hiding is used to force the evolution of the system, since events on hidden gates are *urgent*. In this case, each instantiation of the `TreeIdSync` process may idle forever, because an action, seen alone, can idle any period of time. The general parallel operator can idle if all its components can idle, even though `TreeIdSync` processes have the potential to communicate. However, when we add the `hide` operator it is no longer possible for the whole `SyncImp` process to idle forever. A `hide` behaviour can idle a time d only if its body *cannot* communicate on a hidden gate before d.

3.6 Asynchronous Communication Description

The previous description was designed to be easy to understand, so that the basic ideas of the leader negotiations could be described. We now add more concrete detail by making communication *asynchronous*, as is usual in distributed systems. In this example, messages are sent along wires and are therefore subject to delay. Nodes can no longer assume that their message has been received, so after sending a "be my parent" request a node must wait for an acknowledgement. As a consequence, it is possible for two nodes to simultaneously request that the other is their parent, leading to *root contention* (each wants the other to be the root node). Figure 5 shows how nodes 2 and 4 have each asked the other one to be their parent. Root contention must be resolved, since there can be only one leader. So, by changing one aspect of the model (communication method) we are required to reconsider other aspects (synchronisation) and may have to add more detail (contention resolution).

As before, each node is modelled by a process. Asynchronous communication between nodes is modelled explicitly by making `TreeIDAsync` processes communicate via one place buffers. Resolution of contention requires time and nondeterminism.

Here we give a top-down description of the E-LOTOS model of the system.

```
process AsyncImp [leader] (NW:network) is
 var id:iden, n:nat, l:list of nat in
   ?n := sizeof(NW);
   ?l := (0 upto (n-1));
   hide s:comm, r:comm in
        par ?id in l
          ||| TreeIDAsync[...](id, neighbours(NW,id))
        endpar
     |[s,r]|
        Buffers[s,r](n)
   endhide
 endvar
endproc
```

The novel feature used here is the description of a network of nodes using *parallel over values*, that is,

```
   par ?id in l
     ||| TreeIDAsync [...](id, neighbours(NW,id))
   endpar
```

The variable `id` is instantiated from a list of n values, producing n interleaving instantiations of the `TreeIDAsync` process, each with parameters dependent on the current value of `id`. `TreeIDAsync` describes the behaviour of an individual node in the system. Parallel over values therefore allows an abstract description of a group of processes (which interleave rather than communicating with each other) without specifying the number of processes. In LOTOS such systems must have their parameters and the number of processes hardwired into the specification, which is impractical for systems of a realistic size.

The standard LOTOS form of parallelism `|[`*gates*`]|` connects the `Buffers` and the network of nodes using the gates `s` and `r`. Each `s` (or `r`) in the nodes must communicate with exactly one `s` (or `r`) in `Buffers`. Again, communication on these gates is hidden in order to ensure progress.

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

Each `TreeIDAsync` process uses two buffers to communicate with another

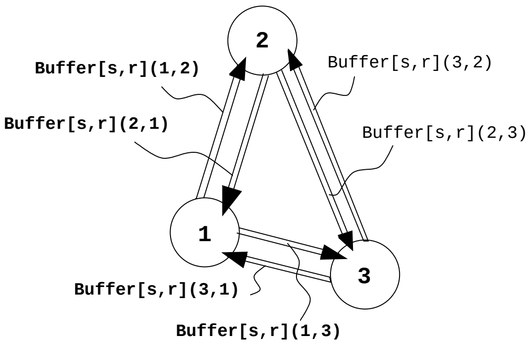

Fig. 6. Buffers

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 again.

Buffers are interleaved; they do not communicate with each other. This description produces more buffers than are necessary (since the network should not be fully connected), but the specification is simple, and as in the last description we can use the data communicated and the synchronisation mechanism to ensure only the buffers appropriate for the network interact with the nodes. The same mechanism is used to force each node to communicate with the *right* buffer and hence the right destination node, although potentially each node can communicate with any of the buffers (because they all use the same gates to synchronise). Figure 6 shows the buffers between nodes 1, 2 and 3 of Figure 1. The wire between nodes `j` and `k` is represented by `Buffer[s,r](j, k)`. Those buffers between nodes 2 and 3 are shown in a lighter typeface to indicate that they are not used (they do not model a connection in the real network).

Notice that communication now requires two gates. In many process algebras the data offers `?` and `!` correspond to input and output and synchronise in pairs with each other. In LOTOS and E-LOTOS all combinations may synchronise; this is because of multiway synchronisation. Multi-way synchronisation can be very useful in constraint-oriented specification, where the basic description can be restricted or enhanced by additional parallel processes, but in this particular example it leads to undesirable behaviour.

In `AsyncImp`, if a single gate were used for communication we generate unexpected synchronisations, for example, a `Buffer` can force a node to generate a request (whereas we think of the `Buffer` as waiting quiescently for a communication to arrive). With two gates this does not happen because the "inputs" of a node (`r` actions) are distinct from the "outputs" of a node (`s` actions).

Time was not mentioned in our original requirements for this level of specification, but the `Buffer` is a convenient place to introduce time. The `Buffer` process is straightforward. Messages in the protocol are sent along wires of variable length (up to 4.5m), yielding variable delay. Buffers receive a message, nondeterministically choose a delay time `t`, wait for time `t`, pass the message on, then loop. The `any` wildcard is used to make a nondeterministic assignment to `t`, subject to the constraints of `lower_buffer` and `upper_buffer`.

```
process Buffer[ip:comm,op:comm](j:iden, k:iden) is
var m:conntype, 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
```

The `wait` statement is used to force time to pass, that is, `wait(t)` is idle while the time `t` passes. Communication on gate `op` cannot happen before this time passes, but is forced to occur just after that time, because all `s` and `r` actions are hidden and therefore urgent. (Note too that the formal and actual event parameters of `Buffer` are different, allowing modular specification.)

Having described the overall network of nodes and buffer communications, we now describe the precise behaviour of a single node `TreeIDAsync`. Since the node behaviour is more complex the description is modularised by division into sequential phases: `ReceiveReqs`, `SendAcks`, `WaitParent`, and `Contention`.

```
process TreeIDAsync[leader, s:comm, r:comm](id:iden,p:connections) is
    ReceiveReqs[...](id,p,{})
endproc
```

The function of process `ReceiveReqs` is similar to that of the main `TreeIDSync` process of Section 3.5. As each "be my parent" request is made the node making the request is removed from the set `p` and added to the set `ch`, the set of children to be acknowledged. The `ReceiveReqs` phase terminates and `SendAcks` is called when `p` is empty, or is a singleton set.

In `SendAcks` children are acknowledged and removed from the set `ch`. When `SendAcks` is finished sending acknowledgements to child nodes it either sends a "be my parent request" to its parent (if there is an element in `p`) and moves to the `WaitParent` phase, or declares itself leader (if the set `p` is empty).

When in `WaitParent` a parent request has been sent, and the node is waiting for an acknowledgement. If a parent request arrives instead then the node and the originating node of the parent request are in contention for leader. The processes are as follows:

```
process ReceiveReqs [leader, s:comm, r:comm]
                (id:iden, p:connections, ch:connections) is
 var j : iden in
   if singleton(p) then
       (  (* branch node, so move to send acknowledgements. *)
          SendAcks[...](id, p, ch)
       [] (* 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, {}, union(ch,{j})))
   else
       (* accept any incoming "be my parent" request *)
       r(?j, !id, !parent)[isin(j, p)];
       ReceiveReqs[...](id, diff(p,{j}), union(ch,{j}))
   endif
 endvar
endproc

process SendAcks [leader, s:comm, r:comm]
                 (id:iden, p:connections, ch:connections) is
 var j:iden, k:iden in
  if isempty(ch) then (* No more acks to make *)
     if isempty(p) then
        leader; SelfID[...](id, p)
     else
        (* send "be my parent" request *)
        s(!id, ?k, !parent)[isin(k, p)];
        WaitParent [...] (id, p)
     endif
  else (* while more children, send an ack *)
     s(!id, ?j, !ack)[isin(j,ch)];
     SendAcks[...](id, p, diff(ch,{j}))
  endif
 endvar
endproc

process WaitParent [leader, s:comm, r:comm](id:iden, p:connections) is
 var j:iden in
   (* There will be only one member of p - the parent.           *)
   ?j := any iden [isin(j,p)];
   (  r(!j, !id, !ack); SelfID[...](id, p)
   [] r(!j, !id, !parent); Contention[...](id, p))
 endvar
endproc
```

In the IEEE 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 present at the port 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. This may lead to contention again, but fairness guarantees that eventually one node will become the root.

```
process Contention [leader, s:comm, r:comm](id:iden, p:connections) is
 (* There will be only one j in p. *)
 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
```

The use of `wait` in this choice context allows us to model the way root contention is resolved in the IEEE 1394 standard quite accurately. The choice between `wait(t)` and `r(!j, !id, !parent)` can idle if both branches can. In this case, although gate `r` is hidden (and urgent), it is in the scope of a parallel operator, so it must idle until communication is possible. The `wait` behaviour idles for duration `t`. Thus, the selection behaviour as a whole idles until either communication on gate `r` is possible or time `t` passes, *but while it idles no choice takes place*, that is, `wait` does not make a pre-emptive choice, committing to actions before time `t` expires. Both branches progress together with respect to time. If communication at gate `r` becomes available before time `t` is reached, then that communication happens. If time `t` is reached, the action on gate `s` becomes available and there is an external choice between the action on gate `s` and the action on gate `r`. Since communication on gate `s` is with a `Buffer` it is offered immediately, and since it is hidden it will occur as soon as possible.

To complete the specification we declare some constants of type time.

```
module Constants is
(* The unit of time is the nanosecond *)
  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
```

The time values are taken from the standard [10]:

- `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),
- `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.

3.7 Complete Timed Description

The previous description did not make use of the time information added to the `Buffers`. In this section we add two timing considerations affecting the flow of control in the process, thereby further demonstrating the use of the timed operators of E-LOTOS, and making the description more like the implementation of the standard [10].

The first timing consideration concerns loop detection. The Tree Identify Protocol establishes that the network contains a cycle when the time limit `CONFIG_TIMEOUT` is reached before all but one of the "be my parent" requests have been made, and an error should be reported. The second timing consideration concerns the force root parameter `fr`. Setting `fr` forces the node to wait a bit longer before moving to the `SendAcks` phase, in the hope that all neighbours will make parent requests (and the node becomes the leader).

We take advantage of the structuring facilities of E-LOTOS to separate timing concerns from the procedure of negotiating the parent relationship. This allows much of the description to remain as it was in Section 3.6.

The new top level description is as follows.

```
process TreeIDTimed[leader, s:comm, r:comm]
        (id:iden, p:connections, fr:bool) raises [ErrorExc] is
 hide alarmct, alarmfr, stopAlarmct, stopAlarmfr in
    ReceiveReqsTimed[...](id, p, {}, card(p), fr)[ErrorExc]
  |[alarmct, alarmfr, stopAlarmct, stopAlarmfr]|
    ( AlarmClock[alarmct, stopAlarmct](CONFIG_TIMEOUT)
    |||
      if fr then AlarmClock[alarmfr, stopAlarmfr](FRTIME) endif)
  endhide
endproc
```

Two separate alarm clocks (one for `CONFIG_TIMEOUT` and the other for `FRTIME`) are used to monitor time passing. These are run in parallel with a modified version of `ReceiveReqs` (called `ReceiveReqsTimed`) in `TreeIDTimed`. Timing considerations affect only the first phase of the protocol. Note that the alarm events are hidden to ensure progress, and that the alarm clock corresponding to `FRTIME` is only needed if the `fr` parameter is set. The alarm clock process is

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

The alarm simply waits the specified time and sets off an alert, or allows itself to be deactivated (before `TO` time passes). As in the previous section the choice is not made until either `stopAlarm` is available or time `TO` passes.

Two new parameters have been added to the `ReceiveReqsTimed` process:

- `n`, the cardinality of `p` when `ReceiveReqsTimed` is first invoked, and
- `fr`, the `FORCE_ROOT` parameter.

```
process ReceiveReqsTimed
        [leader,s:comm,r: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 from any connected node *)
     r(?j, !id, !parent)[isin(j, p)];
     ReceiveReqsTimed[...](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];
     if fr then stopAlarmfr endif;
     SendAcks[...](id, p, ch)
  []
     (* Timeout on FRTIME, so set fr to false.       *)
     alarmfr;
     ReceiveReqsTimed[...](id, p, ch, n, false)[LoopExc]
  []
     (* Timeout on CONFIG_TIMEOUT, therefore there is a loop. *)
     (* alarmfr, if fr is set, must have expired previously.  *)
     alarmct;
     raise LoopExc
endvar
endproc
```

The first branch describes the familiar behaviour from before, that of receiving "be my parent" requests. The second branch deals with the slightly more complex conditions under which a node can move on to the next phase of the protocol. Note that the use of the predicate on the action means that `stopAlarmct` is only enabled when the predicate is satisfied. The force root alarm (if set) must be disabled before `SendAcks` is invoked.

The last two branches deal with the timeout alarm events. The alarm is run in parallel with `ReceiveReqsTimed`, so when `CONFIG_TIMEOUT` time passes, the action `alarmct` is enabled (similarly for `FRTIME` and `alarmfr`). Since the alarm events are hidden they are also urgent. The exception mechanism is used to indicate that the network is incorrectly configured. Setting `fr` to false when `FRTIME` has been exceeded affects the predicate enabling `stopAlarmct` (next time round the recursion).

The new time constants (already in the `Constants` module) are:

- `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, and
- `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.

This description is the closest we can get to the behaviour described in the standard (which is more concrete, since it is described using C). To make the model more realistic we could consider inserting a delay in `ReceiveReqsTimed` to model the length of time a node takes to sample a port. However, this will not affect any of the timed decisions of the protocol. The only time critical part concerns the `CONFIG_TIMEOUT`, and in the case of a loop in the network this will always be reached. In any case, it seems difficult to choose appropriate values to model the time taken to sample ports. 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 the bus, and the translation of that implementation to machine code. 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, however, it is not clear that this is true when a number of nodes have the force root parameter set. A revision to the standard [11] lowers the time bounds for the `FORCE_ROOT` parameter. To illustrate the use of modules, we assume a module called `FireWireProcsTimed` containing the above process definitions, and a top level description `TimedImp` which is just `AsynchImp` of the last section with the call to `TreeIDAsync` replaced by a call to `TreeIDTimed`. Then, the full specification for this most concrete level of abstraction is

```
specification CompleteTimedFireWire
  import Types, TimedFireWireProcs, Constants is
  gates read, leader, error
  behaviour
  var
    NW:network
  in
    trap
      exception LoopException is error endexn
      exception ConnectionError is error endexn
    in
      (* input the network configuration from an external source *)
      read (?NW);
      if connected(NW) then
        TimedImp[leader](NW)[LoopException]
      else raise ConnectionError
      endif
    endtrap
  endvar
endspec
```

Rather than make the assumption that the network is connected (as in the previous sections), we check for connectedness here before invoking the Tree Identify Protocol and raise an exception if there is an unconnected component. The check for loops is part of the protocol in this description, and also triggers an exception.

4 Analysis of E-LOTOS Specifications

The E-LOTOS standard concentrates on the semantics of the language and does not mention analysis techniques or a formal framework for analysis; however, we can consider "standard" analysis for process algebras.

The first analysis exercise to carry out is to check that that specification conforms to the E-LOTOS syntax; as a newly developed language undergoing standardisation there are as yet few tools available, although there are syntax and static semantics checkers being developed in Stirling and in Madrid. There is also a similar tool for a variant of E-LOTOS. For now, we have to rely on detailed manual cross checking with the E-LOTOS standard.

Secondly, we can consider validation in the form of simulation. Here we have carried out simulation by hand, but clearly it would be more desirable to have tool support for this, allowing the user to walk through various execution paths. Such a tool would have to take account of the progression of time.

The formal underpinnings for such a tool (the transition rules of the dynamic semantics of E-LOTOS) are available in the E-LOTOS standard, and a tool for executing E-LOTOS specifications is under development [21].

More rigorous verification can be carried out by taking two descriptions and showing they are consistent in some sense. For example, if both descriptions are in E-LOTOS a suitable equivalence or preorder should be used to establish the relation between the specifications. In this case, we desire that all of the specifications of Section 3 are equivalent. This captures the requirements of the leader election protocol, namely:

- a single leader is chosen (safety),
- a leader is eventually chosen (liveness).

No research has been carried out to establish such relations for E-LOTOS, although since the semantics of the language is based on a variant of labelled transition systems there is a large body of related work which may be drawn upon (at least for the untimed portion).

The other approach is to define those abstract system properties in terms of a modal or temporal logic. Again, since the basis of the semantics is similar to that of established process algebras, establishing such a logic for E-LOTOS should be straightforward. For example, we can consider adapting the logic FULL [2] for E-LOTOS. The real time operators pose the main challenge, since model checking for real time systems is a difficult question [3].

5 Comparison with Other Approaches to the FireWire

Having used E-LOTOS for a fairly intricate leader election protocol, how does it compare as a description language to its predecessor LOTOS? Also, how does it compare to other formal descriptions of this particular example?

It would not be possible to conveniently describe the leader election protocol in such detail using LOTOS. Certainly the data types could be described (although an ACT ONE description would be much longer, and rather cumbersome). The node processes, for example, `Spec`, `TreeIDSync`, `TreeIDAsync` can also be described using LOTOS, since features such as local variables and loops are merely convenient. However, describing a network of nodes, where the number of node is unknown, communicating in pairs is not possible [6] in LOTOS, nor is it possible to talk about quantitative time. So, LOTOS only allows us to describe the protocol at a more abstract level.

Two other formal methods have been used to describe this protocol: I/O automata [4,17] and μCRL [18].

In the (untimed) I/O automata paper [4] the behaviour of the network as a whole is described in terms of predicates over a graph. The level of abstraction is similar to that of `AsynchImp` here. The description is shown to be correct with respect to the basic requirements mentioned in Section 4. A timed I/O automata description is presented in [17]. This is closest to the timed description of Section 3.7, although the effect of the `FORCE_ROOT` parameter is ignored. The description is shown to be correct with respect to the basic requirements.

In the μCRL paper [18] there are three descriptions, corresponding to `Spec`, `SynchImp` and `AsynchImp` in terms of level of abstraction. 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 μCRL descriptions are shown to be equivalent.

Aside from the issue of lack of support for analysis through either a formal framework or tools as discussed in the last section, 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 `SynchImp`). 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 3.6 takes just under a page, as do the I/O automata versions.

We may also consider how E-LOTOS compares to the IEEE standard itself. The standard uses informal text, C implementations to describe the actions taken in particular states (roughly corresponding to the phases of `TreeIDAsynch` or `TreeIDTimed`) and state transition diagrams to 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. So far, no errors have been detected; the absence of errors is much more obvious when constructing a formal model.

In addition, much information is included in the IEEE standard 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 more practical details which are not essential to the operation of the protocol, but also allows more detail to be added when necessary. This is more helpful, in the opinion of the authors, in understanding the protocol.

6 Conclusions

We have described the Tree Identify Protocol of the IEEE 1394 standard for a serial multimedia bus at several different levels of abstraction and exploited some of the new features of E-LOTOS (in particular, the new parallel and timing operators).

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 communications protocols (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 also 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. Moreover, it is possible to express the system at a number of levels of abstraction. The high levels of abstraction are useful for understanding, and it may be possible in the future to generate code directly from the lower levels of abstraction.

We may also ask something more of a specification, particularly if it is to be incorporated in an international standard, namely that it be easily *understandable* to the majority of users. This is a much 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 much more like a programming language while other formalisms such as μCRL are much 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 if it is expected to be accessible to a wide audience of computer professionals. It is still possible to construct systems without using the programming language like constructs, yielding a more abstract specification. This is more desirable when we do not want to be distracted by low level details.

While we did validate 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 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 [19] 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

We gratefully acknowledge the financial support of the British Council and the Ministerio de Educación y Ciencia. Thanks also to Ken Turner of the University of Stirling, whose idea the EASEL project was, 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 the IEEE 1394 standard. Lastly, we would like to thank the referees for their helpful comments.

References

[1] 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.

[2] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Early Symbolic Transition Systems. *The Computer Journal*, 2001. To appear.

[3] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, 1999.

[4] 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.

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

[6] H. Garavel and M. Sighireanu. A graphical parallel composition operator for process algebras. In Jianping Wu, Qiang Gao, and Samuel T. Chanson, editors, *Proceedings of the Joint International Conference on Formal Description*

Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China), pages 185–202. IFIP, Kluwer Academic Publishers, October 1999.

[7] 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.

[8] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[9] G. Huecas, L.F. Llana-Díaz, T. Robles, and A. Verdejo. E-LOTOS: an overview. In *Formal Methods and Telecommunications*, pages 94–102, September 1999.

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

[11] Institute of Electrical and Electronics Engineers. *IEEE Standard for a High Performance Serial Bus — Amendment 1. Std 1394a-2000.*, June 2000.

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

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

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

[15] R. Milner. *Communication and Concurrency.* Prentice-Hall International, 1989.

[16] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *Computer Aided Design*, LNCS 575, pages 376–398, 1991.

[17] 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.

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

[19] M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the IEEE-1394 serial bus (FireWire): an experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer (STTT)*, 2(1):68–88, December 1998.

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

[21] A. Verdejo and N. Martí-Oliet. Executing E-LOTOS processes in Maude. In H. Ehrig, M. Grosse-Rhode, and F. Orejas, editors, *INT 2000, Integration of Specification Techniques with Applications in Engineering, Extended Abstracts Technical report 2000/04, Technische Universitat Berlin, March 2000*, pages 49–53, 2000.