Verification of a Leader Election Protocol

Formal Methods Applied to IEEE 1394

Marco Devillers[1], David Griffioen[1,2], Judi Romijn[1,2], and Frits Vaandrager[1]

1 *Computing Science Institute, University of Nijmegen*
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
{marcod,davidg,judi,fvaan}@cs.kun.nl

2 *CWI*
P.O. Box 94079, 1090 GB Amsterdam, the Netherlands

ABSTRACT

The IEEE 1394 high performance serial multimedia bus protocol allows several components to communicate with each other at high speed. In this paper we present a formal model and verification of a leader election algorithm that forms the core of the tree identify phase of the physical layer of the 1394 protocol.
We describe the algorithm formally in the I/O automata model of Lynch and Tuttle, and verify that for an arbitrary tree topology exactly one leader is elected. A large part of our verification has been checked mechanically with PVS, a verification system for higher-order logic.

1994 Extended Computing Reviews Classification System: B.4.4 [Input/Output and Data Communications]: Performance Analysis and Design Aids - Formal models, verification, B.6.3 [Logic Design]: Design Aids - Verification, D.2.1 [Software Engineering]: Requirements - Specifications, F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying and Reasoning,

1991 Mathematics Subject Classification: 03B35 [Mathematical logic and foundations]: Mechanization of proofs and logical operations, 68Q05 [Theory of computing]: Models of computation, 68Q22 [Theory of computing]: Parallel and distributed algorithms, 68Q40 [Theory of computing]: Symbolic computation, 68Q60 [Computer science]: Specification and verification of programs, 68Q68 [Computer science]: Automata theory, 68U20 [Computing methodologies]: Simulations.

Keywords and Phrases: IEEE 1394, Protocols, Tree Identify Phase, Leader Election Algorithm, Formal Methods, Concurrency Theory, I/O automata, Theorem Provers, PVS.

Notes: The results reported in this paper have been obtained as part of the research project "Specification, Testing and Verification of Software for Technical Applications", which is being carried out for Philips Research Laboratories under Contract RWC-061-PS-950006-ps. The research of the second author is supported by the Netherlands Organization for Scientific Research (NWO) under contract SION 612-316-125.

1. Introduction

The IEEE 1394 high performance serial bus has been developed for interconnecting computer and consumer equipment such as multimedia PCs, digital cameras, VCRs, and CD players. The bus is "hot-pluggable", i.e. equipment can be added and removed at any time, and allows quick, reliable and inexpensive high-bandwidth transfer of digitized video and audio. Although originally developed by Apple (FireWire), the version documented in [IEE96] has been accepted as a standard by IEEE in 1996. More than seventy companies — including Sun, Microsoft, Lucent Technologies, Philips, IBM, and Adaptec — have joined in the development of the IEEE 1394 bus, and related consumer electronics and software. Hence there is a good chance that IEEE 1394 will become the future standard for connecting digital multimedia equipment. The introduction of the bus into consumer electronics is expected to become the driving force behind a new wave of multimedia applications, such as full-motion desktop digital video conferencing, movie quality multimedia authoring, video mail, and home video editing. Researched expansions include increasing the data transfer rate from 400 Mbps to 1 Gbps making the IEEE 1394 bus a serious competitor of, for instance, the PCI personal computer bus.

Like in the OSI model, the IEEE 1394 bus protocol has several layers, of which the physical layer is the lowest. Within the physical layer of the 1394 bus a number of phases are identified. The physical layer protocol enters the tree identify phase whenever a bus reset occurs, for instance, when a component is added to or removed from the bus. The task of the tree identify phase is to check whether the network topology is a tree, and (if this is indeed the case) to elect a leader amongst the nodes in this tree. The leader serves as bus manager in subsequent phases of the protocol. In the IEEE standard document [IEE96] the tree identify phase is specified with a state machine that describes the behavior of one component, and with C^{++} procedures that give the meaning of the actions performed in the state machine. The state machine and C^{++} code have an informal character; the mandatory part of the specification lies in the definition and duration of signals occurring on the bus. The intention of the informal specification is to provide 1394 implementors with a recipe that enables them to conform to the mandatory signal behaviour on the bus, even though in principle there is freedom for any other working solution.

The description of the tree identify phase in the IEEE standard [IEE96] is quite technical and involves details related to the timing of events and the precise signals on the bus. It is also ambiguous and incomplete, so it took us some time to extract from the IEEE standard document the simple leader election algorithm that we present in this paper, and which provides an abstract view of what goes on during the tree identify phase. Within our abstract model we ignore many important details of the 1394 tree identify phase, such as timing restrictions and problems that may occur if the network topology contains a cycle. In subsequent work we have addressed the formal modeling and verification of a number of those additional features [GV98, SV99, Rom99]. In these papers, the model described here is enriched with more detail and it is proved formally that the new models are refinements of the old one. By applying this stepwise refinement approach we arrive at a more modular description of the tree identify phase that is easier to explain and amenable to formal analysis.

In this paper we use as a formal model the I/O automata of Lynch and Tuttle [LT89]. For an introduction to I/O automata see [Lyn96, Chapter 8]. In order to verify leader election in IEEE 1394, we follow a standard refinement approach [LV95, Lyn96, RE98]. First we formally model the algorithm (or implementation) as an I/O automaton. Next the specification of the algorithm ("a unique leader is elected") is also formalized as an I/O automaton. The verification task is then to prove that the implementation refines the specification. This is done by first establishing some invariants for the implementation, use these to prove that there exists a refinement mapping from the implementation to the specification, next establish that all executions of the implementation are finite, and finally prove that maximal executions of the implementation are mapped to maximal executions of the specification. Roughly speaking the existence of a refinement says that at most one root is elected, whereas the preservation of maximality means that at least one root is elected. For the specific algorithm that we study in this paper it is actually somewhat circuitous to give the verification as a refinement proof: the key correctness property can trivially be phrased in terms of the implementation automaton, and in fact 95% of the reasoning only concerns the implementation. However, since in our subsequent work on IEEE 1394 we apply the stepwise refinement paradigm, we thought it was natural to use this approach also for the top level verification.

Parts of the IEEE 1394 architecture have also been formally specified and/or verified by others [Lut97, KHR97, SZ98]. Of these, [Lut97, KHR97, SM98] focus on the link layer. In [SZ98], the tree identify phase of the physical layer is verified in μCRL [GP95], a process algebra language with data. The verification approach is similar to ours and has only manual proofs. For an extensive discussion of leader election algorithms we refer to [Lyn96]. We are not aware of other formal verifications of leader election algorithms involving theorem provers. However, there is much work in which theorem provers are used to formally verify distributed algorithms, see for instance [Cho95, Hes97a, Hes97b, Vos00]. Many of these verifications also involve the formalisation of an arbitrary network topology and reason about graph-theoretic properties.

Our mechanical verification effort builds on earlier work dealing with the formalization of the theory of I/O automata [HSV94, NS95, AH96, DGM97, MN97, Mue98]. We would like stress that the emphasis in our paper is on the analysis of a specific leader election algorithm, rather than on the use of theorem provers in verification. We just used PVS as an "advanced calculator" to increase our confidence in the correctness proofs.

After we had extracted and verified our abstract version of the 1394 leader election algorithm, Nancy Lynch pointed out to us that essentially the same algorithm is described informally in her book on *Distributed Algorithms* [Lyn96, p501]. Clearly, the algorithm was conceived independently by the designers of 1394 and by Lynch. As a consequence, an alternative way to look at our paper is that we provide a formal proof of a result claimed (but not proved) in [Lyn96, Theorem 15.17]. The fact that we found this simple leader election algorithm hidden in IEEE 1394 forms a wonderful illustration of Hoare's law: inside every large program there is a small program trying to get out. It gives us hope that at some point in the future industrial protocols may and will be constructed by assembling and refining known and formally verified algorithms, perhaps described in an "Encyclopedia of Distributed Algorithms".

The outline of the paper is as follows. In Section 2, the leader election algorithm is described in terms of I/O automata. In Section 3, we sketch the mathematical verification, and also discuss the mechanization using PVS, a theorem prover based on higher-order logic [ORSH95]. We end with some conclusions in Section 4.

2. Description of the protocol

In this section we first describe the tree-identify phase of the IEEE 1394 protocol informally, and then formally specify an abstract version as an I/O automaton.

2.1 The IEEE 1394 tree identify phase

In the IEEE 1394 standard, components connected to the bus are referred to as *nodes*. Each node has a number of *ports* which are used for bidirectional *connections* to (other) nodes. Each port has at most one connection. The tree identify phase follows on completion of the bus reset phase, which is for instance started whenever a connection is added or removed. The bus reset phase clears all topology information except local information on a node, i.e., which ports have connections. During the tree identify phase a spanning tree is constructed in the network. A typical network topology is presented in Figure 1.

Informally, the basic idea of the protocol is as follows: leaf nodes send a "parent request" message to their neighbor. When a node has received a parent request from all but one of its neighbors it sends a parent request to its remaining neighbor. In this way the tree grows from the leafs to a root. If a node has received parent requests from all its neighbors, it knows that it is has been elected as the root of the tree. The root node is also referred to as the leader or the bus master.

Let us consider the operation of the protocol in some more detail. During the tree identify phase every node goes through three stages. In the first stage, a node waits until it has received a parent request on all ports or on every port except one. Since leaf nodes have at most one connection, they can skip the first stage, and are thus responsible for starting up the algorithm.

In the second stage, a node acknowledges all the parent requests that it has received and sends a parent request on the remaining port (if any) on which it has not received a parent request. In Figure 2, two links have been assigned as "child links" and one parent request from a leaf node is still pending. In the third stage, if a node has received parent requests on all ports but has not sent a parent request itself, then it decides that it is the root of the tree and terminates. If a node that has sent a parent request receives an acknowledgement then it also terminates but decides that it has not been elected as the root. It is possible that two nodes send parent requests to each other; this situation is called root contention and is illustrated in Figure 3. Whenever root contention occurs, the nodes that are involved retransmit their parent request after random timeouts and return to the beginning of the third stage. After completion of the protocol, the resulting spanning tree in our example network may look as in Figure 4.

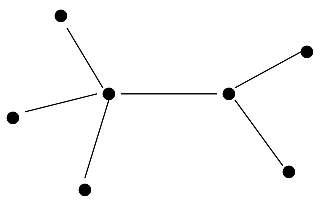

Figure 1: Initial network topology

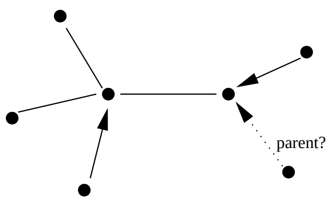

Figure 2: Intermediate configuration

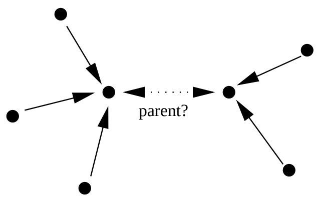

Figure 3: Two contending nodes

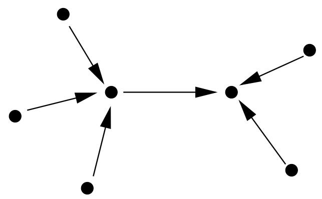

Figure 4: Final spanning tree

If the network contains a cycle, then a node that is part of this cycle can never move to the second stage of the protocol because it will always have (at least) two neighbors from whom it has not received a parent request. In 1394, this deadlock situation is recognized with the help of timers. In principle, 1394 allows every node in the network to become the root of the tree, so even leaves in the network can be elected as leader. However, a node can increase its chances of becoming root by waiting long enough before moving to the second stage.

2.2 The I/O automaton model

In the specification below we abstract from time, ignore cycle detection, and do not model the mechanism for solving root contention. More specifically, we assume that the network topology is cycle free, and that whenever root contention occurs it is resolved in a single atomic action. We describe the protocol as an I/O automaton [LT89, Lyn96].

To model the network topology, we assume the presence of digraph $\mathbf{G}$, which is a tuple of finite non-empty sets of vertices $\mathbf{V}$ and edges $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$. Vertices of $\mathbf{G}$ represent nodes in the network and edges represent connections via which nodes can send messages to each other. We assume that $\mathbf{G}$ is undirected in the sense that $(v, w) \in \mathbf{E} \Leftrightarrow (w, v) \in \mathbf{E}$. Furthermore we assume that there are no self loops in the graph: $(v, v) \notin \mathbf{E}$. Finally, we assume that $\mathbf{G}$ has a tree-like topology: for each pair of vertices v, w, there is a unique sequence of vertices $v_0, \ldots, v_n$ such that (1) $v_0 = v$, (2) $v_n = w$, (3) for all $0 \leq i < n$, $(v_i, v_{i+1}) \in \mathbf{E}$, and (4) no vertex occurs more than once in the sequence.

Internal: ADD_CHILD, $CHILDREN\_KNOWN$, $RESOLVE\_CONTENTION$, ACK
Output: $ROOT$

State Variables:
$init : \mathbf{V} \to \mathbf{Bool}$
$contention : \mathbf{V} \to \mathbf{Bool}$
$root : \mathbf{V} \to \mathbf{Bool}$
$child : \mathbf{E} \to \mathbf{Bool}$
$mq : \mathbf{E} \to \mathbf{Bool}^*$

Init: $\forall v, e : init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge \neg child[e]$
$\wedge\ mq[e] = \mathsf{empty}$

$ADD\_CHILD(e : \mathbf{E})$
Precondition:
$\wedge\ init[\mathsf{target}(e)]$
$\wedge\ mq[e] \neq \mathsf{empty}$
Effect:
$child[e] := 1$
$mq[e] := \mathsf{tl}(mq[e])$

$CHILDREN\_KNOWN(v : \mathbf{V})$
Precondition:
$\wedge\ init[v]$
$\wedge\ \forall e, f \in \mathsf{to}(v) : child[e] \vee child[f] \vee e = f$
Effect:
$init[v] := 0$
for $e \in \mathsf{from}(v)$ **do** $mq[e] := \mathsf{append}(child[e^{-1}], mq[e])$

$ACK(e : \mathbf{E})$
Precondition:
$\wedge \neg init[\mathsf{target}(e)]$
$\wedge\ mq[e] \neq \mathsf{empty}$
Effect:
$contention[\mathsf{target}(e)] := \neg \mathsf{hd}(mq[e])$
$mq[e] := \mathsf{tl}(mq[e])$

$RESOLVE\_CONTENTION(e : \mathbf{E})$
Precondition:
$\wedge\ contention[\mathsf{source}(e)]$
$\wedge\ contention[\mathsf{target}(e)]$
Effect:
$child[e] := 1$
$contention[\mathsf{source}(e)] := 0$
$contention[\mathsf{target}(e)] := 0$

$ROOT(v : \mathbf{V})$
Precondition:
$\wedge \neg init[v]$
$\wedge \neg contention[v]$
$\wedge \neg root[v]$
$\wedge\ \forall e \in \mathsf{to}(v) : child[e]$
Effect:
$root[v] := 1$

Figure 5: I/O automaton *TIP*.

In Figure 5, the protocol is specified as an I/O automaton *TIP* using a standard precondition/effect notation. For each link $e{=}(v,w)$, the source v is denoted $\mathsf{source}(e)$, the target w is denoted $\mathsf{target}(e)$, and the reverse link (w,v) is denoted e^{-1}. For each node v, $\mathsf{from}(v)$ gives the set of links with source v, and $\mathsf{to}(v)$ gives the set of links with target v. So $e{\in}\mathsf{from}(v) \Leftrightarrow \mathsf{source}(e){=}v$ and $e{\in}\mathsf{to}(v) \Leftrightarrow \mathsf{target}(e){=}v$. All the other data types and operation symbols used in the specification have the obvious meaning.

The state variables in *TIP* have the following intuition: for each node v and link e, $\mathit{init}[v]$, $\mathit{contention}[v]$ and $\mathit{root}[v]$ are Boolean variables that indicate the current stage of the protocol for node v, $\mathit{child}[e]$ records whether e has become a child link, and $\mathit{mq}[e]$ records the messages in transit in e. Initially, there are no messages in transit, and all Boolean variables are 0, except $\mathit{init}[v]$ which is 1 for each v. For a node v or link e, the state variable values are interpreted as follows. If $\mathit{init}[v]$ is 1, then v is still in the first stage of the protocol, otherwise v has left the first stage and all its neighbors except one are children. If $\mathit{contention}[v]$ is 1, then v is in root contention with one of its neighbours. If $\mathit{root}[v]$ is 1, then v has found out that all its neighbors are children and has declared itself to be the root of the tree. If $\mathit{child}[e]$ is 1, then $\mathsf{source}(e)$ is the child of $\mathsf{target}(e)$. The Boolean sequence $\mathit{mq}[e]$ represents the message queue from $\mathsf{source}(e)$ to $\mathsf{target}(e)$. If the sequence $\mathit{mq}[e]$ is not empty, then the head of the sequence is the first message to be delivered. In the sequence, value 1 represents a parent request, and value 0 represents an acknowledgment.

The actions in *TIP* reflect the tree identify protocol behaviour as follows. Consider node v. Initially, $\mathit{init}[v]$ is 1, so v is in the first stage of the protocol. In this stage, v can receive incoming parent request messages from neighboring nodes through the action $\mathit{ADD\_CHILD}$. Whenever $\mathit{ADD\_CHILD}(e)$ is performed, link e along which the message arrives is designated as a 'child' link by setting the Boolean variable $\mathit{child}[e]$ to 1. If all incoming links of node v *or* all links except one are designated as child links, v may perform action $\mathit{CHILDREN\_KNOWN}(v)$, which contains all the behaviour of the second stage of the protocol. It is clear that if v is a leaf node in the network topology, then v has at most one incoming link. Thus upon the start of the protocol, v may perform $\mathit{CHILDREN\_KNOWN}(v)$ immediately. As a result of $\mathit{CHILDREN\_KNOWN}(v)$, $\mathit{init}[v]$ is set to 0, and an acknowledgment message is sent to the source of each child link. Moreover, if there is an incoming link that is not a child link, then a parent request message is sent to the source of that link. After performing $\mathit{CHILDREN\_KNOWN}(v)$, v is in the third stage of the protocol. If it has sent a parent request message to a neighbor, then it can receive a message from this neighbor with the action ACK. If the message is an acknowledgment, then the neighbor has become the parent of v in the tree, hence v is not the root and its activity is finished. If the message is a parent request then we have a situation of root contention. In the I/O automaton *TIP* an action $\mathit{RESOLVE\_CONTENTION}$ then nondeterministically designates one of the contending nodes as the parent and the other as the child. If v discovers that all its incoming links are 'child' links, it infers that it has become root and performs a $\mathit{ROOT}(v)$ action.

3. Verification

The verification is done with a standard I/O automaton method [LT89, Lyn96, LV95]. The required behavior of the I/O automaton *TIP* is captured in terms of a two state I/O automaton SPEC, given in Figure 6. The goal then is to prove that the trace (observable behavior) of any fair run of *TIP* is also a trace of a fair run of SPEC. Establishing this amounts to proving that *exactly* one node will perform a ROOT action and thereby become the root (bus master) of the network.

The verification proceeds in a number of steps. First, a number of invariant properties of automaton *TIP* are established. Next, these properties are used to prove that a refinement exists from automaton *TIP* to automaton SPEC. This implies that each run of the *TIP* automaton can be simulated by the SPEC automaton. As a third step in the verification it is shown that all executions of *TIP* are finite. Finally, we establish that the trace of any fair run of *TIP* is also the trace of a fair run of SPEC. Since SPEC only has fair executions consisting of a single (observable) action ROOT, it suffices to

Output: $ROOT$	
State Variables: $done$: **Bool**	**Init:** $\neg done$
$ROOT(v : \mathbf{V})$	
Precondition:	
$\neg done$	
Effect:	
$done := 1$	

Figure 6: I/O automaton *SPEC*.

prove that each fair run of *TIP* contains exactly one $ROOT$ action.

3.1 Invariants

Theorem 1 *For all nodes v and links e, f, the following properties are invariants of TIP, i.e., are valid for all reachable states:*

If a node is in the initial stage then it is not involved in root contention.
$I_1(v) \triangleq init[v] \;\rightarrow\; \neg contention[v]$

If a node is in the initial stage then its outgoing links are empty.
$I_2(e) \triangleq init[\mathsf{source}(e)] \;\rightarrow\; mq[e] = \mathsf{empty}$

If a node is in the initial stage then it is not a child of any of its neighbors.
$I_3(e) \triangleq init[\mathsf{source}(e)] \;\rightarrow\; \neg child[e]$

If a node has left the initial stage then all links, or all links but one, are child links.
$I_4(e, f, v) \triangleq \mathsf{target}(e) = \mathsf{target}(f) = v \wedge e \neq f \;\rightarrow\; init[v] \vee child[e] \vee child[f]$

Each link contains at most one message at a time.
$I_5(e) \triangleq \mathsf{length}(mq[e]) \leq 1$

If a node is in the initial stage then none of its neighbors is involved in root contention.
$I_6(e) \triangleq init[source(e)] \;\rightarrow\; \neg contention[\mathsf{target}(e)]$

Child links are empty.
$I_7(e) \triangleq child[e] \;\rightarrow\; mq[e] = \mathsf{empty}$

If a node is involved in root contention then all its incoming links are empty.
$I_8(e) \triangleq contention[\mathsf{target}(e)] \;\rightarrow\; mq[e] = \mathsf{empty}$

A node only sends acknowledgements to its children.
$I_9(e) \triangleq mq[e] \neq \mathsf{empty} \wedge \mathsf{hd}(mq[e]) \;\rightarrow\; child[e^{-1}]$

A node never sends a parent request to its children.
$I_{10}(e) \triangleq mq[e] \neq \mathsf{empty} \wedge \neg\mathsf{hd}(mq[e]) \;\rightarrow\; \neg child[e^{-1}]$

Two nodes can never be children of each other.
$I_{11}(e) \triangleq child[e] \;\rightarrow\; \neg child[e^{-1}]$

If a node is involved in root contention then it is not a child of any of its neighbors.
$I_{12}(e) \triangleq contention[\mathsf{source}(e)] \;\rightarrow\; \neg child[e]$

All the incoming links of a root node are child links.
$I_{13}(e) \triangleq root[\mathsf{target}(e)] \;\rightarrow\; child[e]$

All incoming links of the source of a child link, except for its reverse, are child links as well.
$I_{14}(e, f) \triangleq child[e] \wedge \mathsf{source}(e) = \mathsf{target}(f) \wedge e \neq f^{-1} \;\rightarrow\; child[f]$

There is at most one node for which all incoming links are child links.

$$I_{15}(v) \triangleq (\exists_v \forall_{e \in \mathsf{to}(v)} \mathit{child}[e]) \rightarrow (\exists!_v \forall_{e \in \mathsf{to}(v)} \mathit{child}[e])$$

The proofs of invariants I_1 to I_{14} are given by induction on the length of the executions leading to a state. Invariant I_{15} follows by contradiction. Suppose that we have two nodes v, w with the property that all incoming links are child links. There exists an unique cycle free path between v and w. Then by invariant I_{14} all child edges on that path must point to v. But then w must also have an outgoing child edge. This contradicts invariant I_{11}.

Invariants I_{13} and I_{15} are used in the proof that there exists a step refinement mapping from TIP to SPEC: together they imply that at most one node can become the root of the tree. All the other invariants are introduced only to prove I_{13} and I_{15}.

3.2 At most one root action

We exhibit a step refinement mapping from the states of TIP to the states of SPEC. The existence of such a mapping implies that every trace of observable actions of the TIP automaton is a trace of observable actions of the SPEC automaton. Since at most one ROOT-action may occur in each trace of SPEC, this implies that TIP may also perform at most one ROOT-action.

Theorem 2 *Let $r \in \mathit{states}(\mathit{TIP}) \rightarrow \mathit{states}(\mathit{SPEC})$ be the function defined by the state predicate:*

$$\mathit{SPEC.done} \quad \Leftrightarrow \quad \exists_v \, \mathit{TIP.root}[v].$$

Then r is a step refinement (in the sense of [GV98]):

1. *The start state of TIP is mapped onto the start state of SPEC.*
2. *for every reachable state s and transition $s \xrightarrow{a} t$ in TIP,*
   - *If a is an external action then $r(s) \xrightarrow{a} r(t)$ is a transition of SPEC,*
   - *otherwise $r(s) = r(t)$.*

The only nontrivial case in the proof is the one in which TIP has a transition $s \xrightarrow{\mathit{ROOT}(v)} t$, for reachable states s and t. In this case, the precondition of action $\mathit{ROOT}(v)$ implies that $s \models \neg \mathit{root}[v] \wedge \; \forall e \in \mathsf{to}(v) : \mathit{child}[e]$. By invariant I_{15}, v is the only node for which all incoming links are child links. Since by invariant I_{13} all incoming links of a root node are child links but v is not a root node, $s \models \neg \exists_v \mathit{root}[v]$. Hence, by definition of r, $r(s) \models \neg \mathit{done}$. By the effect of action $\mathit{ROOT}[v]$ and by definition of r, $r(t) \models \mathit{done}$. This implies that SPEC has a transition $r(s) \xrightarrow{\mathit{ROOT}(v)} r(t)$, as required.

3.3 Termination

Theorem 3 *All executions of TIP are finite.*

We prove Theorem 3 by defining a norm function on the states of automaton TIP. For this, four auxiliary state functions are required. Let s be a state of TIP. Then:

$$\begin{aligned}
I(s) &= |\{v \in V \mid s \models \mathit{init}[v]\}| \\
U(s) &= |\{e \in E \mid s \models \neg \mathit{child}[e]\}| \\
M(s) &= |\{e \in E \mid s \models \mathit{mq}[e] \neq \mathsf{empty}\}| \\
R(s) &= |\{v \in V \mid s \models \neg \mathit{root}[v]\}|
\end{aligned}$$

Define $norm(s) = (I(s), U(s) + M(s) + R(s))$. If $\prec$ denotes the usual lexicographical ordering on tuples then it is straightforward to prove, using invariants I_5, I_7 and I_{12}, that for each transition $s \xrightarrow{a} t$ of *TIP* with s reachable, $norm(t) \prec norm(s)$.

3.4 At least one root action

As final step in our verification we prove that each fair trace of *TIP* is also a fair trace of *SPEC*. Since, by Theorem 2, r is a step refinement from *TIP* to *SPEC*, it suffices to prove that r maps each fair execution of *TIP* to a fair execution of *SPEC*. However since, by Theorem 3, *TIP* has only finite executions, it in fact suffices to prove that r maps each quiescent (final) state of *TIP* to the unique quiescent (final) state of *SPEC*. This follows from the following theorem.

Theorem 4 *In each quiescent state of TIP at least one node has been elected as root.*

In order to prove Theorem 4, we need two additional invariants.

Lemma 5 *For each link e, the following properties are invariants of TIP:*

If the target of an empty, non-child link is in stage init then the source is in stage init as well.
$$I_{16}(e) \triangleq init[\mathsf{target}(e)] \wedge \neg child[e] \wedge mq[e] = \mathsf{empty} \ \rightarrow \ init[\mathsf{source}(e)]$$

If a node has left the initial stage then its neighbors on undirected empty links are in root contention.
$$I_{17}(e) \triangleq \neg init[\mathsf{source}(e)] \wedge mq[e] = \mathsf{empty} \wedge \neg child[e] \wedge \neg child[e^{-1}] \ \rightarrow \ contention[\mathsf{target}(e)]$$

Now to prove Theorem 4, assume that s is a quiescent state of *TIP*. We show that $s \models \exists_v : root[v]$.

For each link e, $s \models mq[e] = \mathsf{empty}$, otherwise *ADD_CHILD* or *ACK* would be enabled (depending on whether or not $s \models init[\mathsf{target}(e)]$).

We prove by contradiction that, for each node v, $s \models \neg init[v]$. Suppose there exists a node v_0 with $s \models init[v_0]$. Since s does not enable *CHILDREN_KNOWN*(v_0), there exist links $e, f \in \mathsf{to}(v_0)$ with $e \neq f$ and $s \models \neg child[e] \wedge \neg child[f]$. Let $v_1 = \mathsf{source}[e]$. Invariant I_{16} implies $s \models init[v_1]$. Repeating this argument, we find an infinite sequence of adjacent nodes $v_0, v_1, v_2, \ldots$ such that, for all i, $v_{i+2} \neq v_i$ and $s \models init[v_i]$. This contradicts our assumption that $\mathbf{G}$ is finite and tree-like.

We proof by contradiction that, for each link e, $s \models child[e] \vee child[e^{-1}]$. Suppose there is an e such that $s \models \neg child[e] \wedge \neg child[e^{-1}]$. Then, by invariant I_{17}, $s \models contention[\mathsf{source}(e)] \wedge contention[\mathsf{target}(e)]$. Thus s enables *RESOLVE_CONTENTION*(e). Contradiction.

Because in s each link or its reverse is a child link, and because $\mathbf{G}$ is finite and tree-like, there exists a node v such that $s \models \forall e \in \mathsf{to}(v) : child[e]$. Hence $s \models root[v]$, otherwise s would enable *ROOT*(v).

3.5 Verification by hand and in PVS

The mathematical proof has been worked out in great detail by hand. Since invariant proofs often result in a lot of bookkeeping work, small faults are easily introduced. For that reason, we used a series of LaTeXmacros that support the structured proof style as advocated by Lamport[Lam93]. The handwritten proofs can be obtained at the URL `http://www.cs.kun.nl/~marcod/1394.html`.

We used PVS to check the proofs of the invariants and the step refinement mapping, i.e. the results of Theorem 1, Theorem 2 and Lemma 5. The PVS specification and proof files can also be obtained at the above URL.

The specification consists of ten modules. Module `auxdefs` contains some simple definitions on lists and relations. Module `digraph` defines a strongly connected digraph and the notions neighbours, reachable nodes, and path. Module `topology` defines a topology as a strongly connected digraph with a finite number of vertices. The state space and actions of automaton *TIP* are defined in `stateact`, and the transaction relation of *TIP* is in `trans`. Module `reachstate` gives a recursive definition of reachability, in `invariant` invariants I_1 to I_{14} and I_{16} and I_{17} are proved, and in `one_ root` invariant I_{15} is proved. In `automaton` one can find definitions of the notions of an I/O automaton and refinement. In the last module, `weak_refinement`, the automata *TIP* and *SPEC* are defined and the refinement is proved.

In our experience, see also [AH96], it is much faster to check invariants with the PVS system then to prove them by hand. The PVS system takes care of the bookkeeping, and trivial steps in the proof are often done automatically. During the construction of a proof, the backward style of reasoning of the PVS system leads to a clear sub-goal which often suggests how to proceed with the proof. Most invariants could therefore be proven without looking into the hand proofs.

However, the use of a proof checker also produces overhead in the sense that the model used has to be formalized in the specification language of the proof checker. In our case, the I/O automata and the the proof principle for invariants were trivially specified, and the proofs of invariants I_1-I_{14} were done within a few days. However, for the proof of invariant I_{15} one needs to reason about paths between nodes in an acyclic finite strongly connected digraph. It took us a few days to explain all the concepts involved here to PVS. After that, checking the proof of the step refinement was trivial.

In general, we feel that using a proof checker is to be preferred above constructing handwritten proofs. First, a proof checker can be applied for something which we would like to name 'rapid proof development'. As mentioned above, a proof checker often helps the user by supplying a goal which clarifies which problem has to be solved. Also, a proof checker can be used to rapidly try different approaches to prove a specific invariant. Second, bookkeeping is done automatically by the prover, which mechanizes a substantial part of the proof development. Third, and most important, use of a theorem prover dramatically increases the confidence in the correctness of a proof.

4. Conclusions

In our opinion the formal specification and verification of the tree identify phase of IEEE 1394 constitutes a nice challenge for the formal methods community. As a first step, we have presented in this paper the verification of a highly abstract version of this protocol. In subsequent work, we investigate various refinements of our model, which, amongst others, involve timing and probabilistic aspects [GV98, SV99, Rom99]. In [GV98], PVS is also used to formally check the correctness of (key parts of) the refinements.

For our verification, we first worked out all the proofs by hand and only afterwards checked them using PVS. Probably we could have gained some time by using some modern tools that help the verifier to derive invariants automatically. For instance, we expect that the tool of [BLS96] will be able to derive most of the invariants I_1–I_{14} automatically when asked to prove I_{15}–I_{17}. An alternative would be to use a tool such as STeP [MBSU98], which has been designed precisely for proofs of this sort. However, thus far STeP does not support stepwise refinement.

In our view, the current IEEE 1394 standard is ambiguous and incomplete. It is only possible to understand the intended meaning of this document through personal contact with the experts who were directly involved in creating it. Clearly, the standard can be improved by providing a more formal and better modularized specification of different parts of the protocol. This would serve a manifold purpose. First, it will help to remove ambiguities in the current specification, which will be useful for engineers who have to build applications on top of 1394. Second, as the IEEE 1394 bus is of great industrial importance, it is bound to be extended with functionalities. A more formal and better modularized specification will facilitate the development of extensions of the protocol. Finally, an improved specification may serve as a basis for formal verification, and thereby help to increase confidence in the correctness of the protocol.

References

[AH96] M. Archer and C. Heitmeyer. Mechanical verification of timed automata: A case study. In *Proceedings IEEE Real-Time Technology and Applications Symposium (RTAS'96).* IEEE Computer Society Press, 1996.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saidi. Powerful techniques for the automatic generation of invariants. In R. Alur and T.A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, NJ, USA, volume 1102 of *Lecture Notes in Computer Science*, pages 323–335. Springer-Verlag, July/August 1996.

[Cho95] C.-T. Chou. Mechanical verification of distributed algorithms in higher-order logic. *The Computer Journal*, 38(2):152–161, 1995.

[DGM97] M.C.A. Devillers, W.O.D. Griffioen, and O. Müller. Possibly infinite sequences: A comparative case study. In E.L. Gunter and A. Felty, editors, *10th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'97)*, volume 1275 of *Lecture Notes in Computer Science*, pages 89–104. Springer-Verlag, 1997.

[GP95] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer-Verlag, 1995.

[GV98] W.O.D. Griffioen and F.W. Vaandrager. Normed simulations. In A.J. Hu and M.Y. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification,* Vancouver, BC, Canada, volume 1427 of *Lecture Notes in Computer Science*, pages 332–344. Springer-Verlag, June/July 1998.

[Hes97a] W.H. Hesselink. A mechanical proof of Segall's PIF algorithm. *Formal Aspects of Computing*, 9:208–226, 1997.

[Hes97b] W.H. Hesselink. The verified incremental design of a distributed spanning tree algorithm - extended abstract, November 1997. To appear in *Formal Aspects of Computing.*

[HSV94] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings International Workshop TYPES'93,* Nijmegen, The Netherlands, May 1993, volume 806 of *Lecture Notes in Computer Science*, pages 127–165. Springer-Verlag, 1994.

[IEE96] IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.

[KHR97] L. Kühne, J. Hooman, and W.P. de Roever. Towards mechanical verification of parts of the IEEE P1394 serial bus. In I. Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design,* Zagreb, pages 73–85, 1997.

[Lam93] L. Lamport. How to write a proof. Research Report 94, Digital Equipment Corporation, Systems Research Center, February 1993.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, September 1989.

[Lut97] S.P. Luttik. Description and formal specification of the Link layer of P1394. In I. Lovrek, editor, *Proceedings of the 2nd International Workshop on Applied Formal Methods in System Design,* Zagreb, pages 43–56, 1997. Also available as Report SEN-R9706, CWI, Amsterdam. See URL `http://www.cwi.nl/~luttik/`.

[LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.

[MBSU98] Z. Manna, A. Browne, H.B. Sipma, and T.E. Uribe. Visual abstraction for temporal verification. In *Proceedings AMAST'98*, Lecture Notes in Computer Science, pages 28–41. Springer-Verlag, 1998.

[MN97] O. Mueller and T. Nipkow. Traces of I/O-automata in Isabelle/HOLCF. In M. Bidoit and M. Dauchet, editors, *Proceedings TAPSOFT'97: Theory and Practice of Software Development,* Lille, France, volume 1214 of *Lecture Notes in Computer Science*, pages 580–594. Springer-Verlag, April 1997.

[Mue98] O. Mueller. *A Verification Environment for I/O Automata Based on Formalized Meta-Theory.* PhD thesis, Technical University of Munich, 1998.

[NS95] T. Nipkow and K. Slind. I/O automata in Isabelle/HOL. In P. Dybjer, B. Nordström, and J. Smith, editors, *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, pages 101–119. Springer-Verlag, 1995.

[ORSH95] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[RE98] W.P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge Tracts in Theoretical Computer Science 47. Cambridge University Press, 1998.

[Rom99] J.M.T. Romijn. A timed verification of the IEEE 1394 leader election protocol. In S. Gnesi and D. Latella, editors, *Proceedings of the Fourth International ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'99)*, pages pages 3–29, 1999. Full version available from URL `http:www.cs.kun.nl/~judi`

[SM98] 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.

[SV99] M.I.A. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings 5th International AMAST Workshop on Formal Methods for Real-Time and Probabilistic Systems,* Bamberg, Germany, volume 1601 of *Lecture Notes in Computer Science*, pages 53–74. Springer-Verlag, 1999.

[SZ98] C. Shankland and M.B. van der Zwaag. The tree identify protocol of IEEE 1394 in μCRL. *Formal Aspects of Computing*, 10:509–531, 1998.

[Vos00] T.E.J. Vos. *UNITY in Diversity: A Stratified Approach to the Verification of Distributed Algorithms.* PhD thesis, Utrecht University, 2000. In preparation.