Verification of a Leader Election Protocol

M.C.A. Devillers, W.O.D. Griffioen, J.M.T. Romijn, F.W. Vaandrager

Computing Science Institute/

CSI-R9728 December 1997

Computing Science Institute Nijmegen
Faculty of Mathematics and Informatics
Catholic University of Nijmegen
Toernooiveld 1
6525 ED Nijmegen
The Netherlands

Verification of a Leader Election Protocol

Formal Methods Applied to IEEE 1394

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

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

2 *CWI*
P.O. Box 94079, 1090 GB Amsterdam, the Netherlands
judi@cwi.nl

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 the 1394 implementor with a recipe that enables him/her 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 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. We are currently working on the formal modeling and verification of a number of those additional features. Our approach is to enrich the model described in this paper with more detail and to prove formally that the new model is a refinement of the old one. By applying this stepwise refinement approach we hope to arrive at a more modular description of the tree identify phase that is easier to explain and amenable to formal analysis.

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 the I/O automata model of Lynch and Tuttle [LT89, Lyn96]. 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 a leaf node has at most one connection, it can skip the first stage. 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 request messages on all ports but has not sent a parent request message itself, then it decides that it is the root of the tree and terminates. If a node that has sent a parent request message 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 request messages 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.

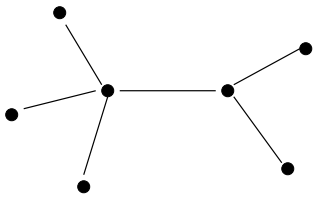

Figure 1: Initial network topology

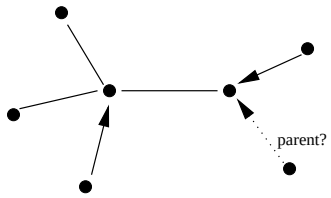

Figure 2: Intermediate configuration

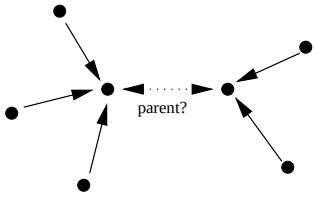

Figure 3: Two contending nodes

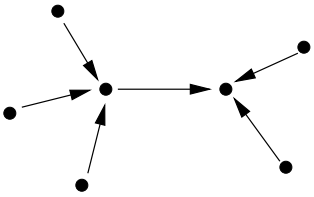

Figure 4: Final spanning tree

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 allways 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 nonempty 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 the 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.

In Figure 5, the protocol is specified as an I/O automaton *TIP* using a standard precondition/effect notation [LT89]. 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.

Each node v goes through three stages. In the first stage, in which Boolean variable $init[v]$ is 1, v receives incoming parent request messages from neighboring nodes through the action *ADD_CHILD*. Node v designates the links e along which these messages arrive as 'child' links by setting a Boolean variable $child[e]$ to 1. If all incoming links of v ór all links except one are designated as child links, v may move to the second stage in which $init[v]$ is set to 0. The second stage is condensed into a single atomic action $CHILDREN\_KNOWN(v)$, for each node v. As a result of this action an acknowledgment message (simply encoded as a 1) is sent over all reverse child links. Moreover, if there is an incoming link that is not a child link, then a parent request message (encoded as a 0) is sent to the source of that link (say w). If subsequently, in the third stage, a reply arrives in which w asks v to become its parent then we have a situation of 'root contention'. In the I/O automaton *TIP* an action *RESOLVE_CONTENTION* then nondeterministically designates one of the nodes as the parent and the other as the child. As soon as a node discovers that all its incoming links are 'child' links it decides that it has become the root of the tree and performs a *ROOT* action.

3. Verification

The verification is done with a standard I/O automaton method [LT89, Lyn96, LV95]. The required behavior of the automaton *TIP* is captured in terms of a more abstract I/O automaton *SPEC*. The goal then is to prove that the trace (observable behavior) of any fair run of I/O automaton *TIP* is also a trace of a fair run of the abstract I/O automaton *SPEC*. The abstract automaton *SPEC* is given in Figure 6. Proving that the set of fair traces of *TIP* is included in the set of fair traces of *SPEC* 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

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.

Output: $ROOT$
State Variables: $done : \mathbf{Bool}$ **Init:** $\neg done$

$ROOT(v : \mathbf{V})$
Precondition:
$\neg done$
Effect:
$done := 1$

Figure 6: I/O automaton $SPEC$.

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

3.1 Invariants

Theorem 1 *For all nodes v and for all 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.
$I_{15}(v) \triangleq (\exists_v \forall_{e \in \mathsf{to}(v)} child[e]) \rightarrow (\exists!_v \forall_{e \in \mathsf{to}(v)} child[e])$

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}.

Invariant I_{15} plays a key role in the proof that there exists a weak refinement mapping from *TIP* to *SPEC* since it implies that at most one node can become the root of the tree.

3.2 At most one root action

We exhibit a weak 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 $r \in states(TIP) \to states(SPEC)$ be the function defined by the state predicate:*

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

Then r is a weak refinement mapping (in the sense of [LV95]) from TIP to SPEC.

In order to prove Theorem 2, it suffices to show that r satisfies the following conditions: (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 $s \xrightarrow{ROOT(v)} t$, for reachable states s and t. In this case, the precondition of action $ROOT(v)$ implies that $s \models \neg root[v] \wedge\ \forall e \in \mathsf{to}(v) : child[e]$. 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, $s \models \neg\exists_v root[v]$. Hence, by definition of r, $r(s) \models \neg done$. By the effect of action $ROOT[v]$ and by definition of r, $r(t) \models done$. This implies that *SPEC* has a transition $r(s) \xrightarrow{ROOT(v)} r(t)$, 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:

$$\begin{aligned} I(s) &= |\{v \in V \mid s \models init[v]\}| \\ U(s) &= |\{e \in E \mid s \models \neg child[e] \wedge \neg child[e^{-1}]\}| \\ M(s) &= |\{e \in E \mid s \models mq[e] \neq \mathsf{empty}\}| \\ R(s) &= |\{v \in V \mid s \models \neg root[v]\}| \end{aligned}$$

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_3, 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 weak 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 a node is in its initial stage then its neighbors on incoming non-child and empty links are also in their initial stage.

$$I_{16}(e) \triangleq \mathit{init}[\mathsf{target}(e)] \wedge \neg \mathit{child}[e] \wedge \mathit{mq}[e] = \mathsf{empty} \;\rightarrow\; \mathit{init}[\mathsf{source}(e)]$$

If a node is not in its initial stage then its neighbors on undirected empty links are in root contention.

$$I_{17}(e) \triangleq \neg \mathit{init}[\mathsf{source}(e)] \wedge \mathit{mq}[e] = \mathsf{empty} \wedge \neg \mathit{child}[e] \wedge \neg \mathit{child}[e^{-1}] \;\rightarrow\; \mathit{contention}[\mathsf{target}(e)]$$

Now to prove Theorem 4, assume that s is a state of *TIP* without any outgoing transitions. We will show that $s \models \exists_v : \mathit{root}[v]$.

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

We prove by contradiction that, for each node v, $s \models \neg \mathit{init}[v]$. Suppose there exists a node v_0 with $s \models \mathit{init}[v_0]$. Since s does not enable the action $\mathit{CHILDREN\_KNOWN}(v_0)$, there exist links $e, f \in \mathsf{to}(v_0)$ with $e \neq f$ and $s \models \neg \mathit{child}[e] \wedge \neg \mathit{child}[f]$. Let $v_1 = \mathsf{source}[e]$. Invariant I_{16} implies $s \models \mathit{init}[v_1]$. Repeating this argument, we can 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 $s \models \mathit{init}[v_i]$. This contradicts our assumption that $\mathbf{G}$ is finite and tree-like.

We claim that, for each link e, $s \models \mathit{child}[e] \vee \mathit{child}[e^{-1}]$. Again, the proof is by contradiction: suppose there exists a link e with $s \models \neg \mathit{child}[e] \wedge \neg \mathit{child}[e^{-1}]$. Then we may infer, by invariant I_{17}, that $s \models \mathit{contention}[\mathsf{source}(e)] \wedge \mathit{contention}[\mathsf{target}(e)]$. But this means that s enables the action $\mathit{RESOLVE\_CONTENTION}(e)$. 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 $s \models \forall e \in \mathsf{to}(v) : \mathit{child}[e]$. This means that $s \models \mathit{root}[v]$, otherwise s would enable a $\mathit{ROOT}(v)$ action.

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 are obtainable at the URL `http://www.cs.kun.nl/~marcod/1394.html`.

We used PVS to check the proofs of the invariants and the weak-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. 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 acyclic finite strongly-connected digraphs. It took us a few days to explain all the concepts involved here to PVS. After that, checking the proof of the weak 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. Currently, we are investigating various refinements of our abstract model, which, amongst others, involve timing aspects. Our aim is to formally prove the correctness of (key parts of) these refinements using PVS.

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.M. Archer and C.L. Heitmeyer. Mechanical verification of timed automata: A case study. In *Real-Time Technology and Applications Symposium*, Lecture Notes in Computer Science. Springer-Verlag, 1996.

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

[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.

[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.

[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.