# Mechanical Verification of the IEEE 1394a Root Contention Protocol using Uppaal2k

D.P.L. Simons[1] and M.I.A. Stoelinga[2]

[1] Philips Research Laboratories Eindhoven, Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands.
`david.simons@philips.com`
[2] Computing Science Institute, University of Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
`marielle@cs.kun.nl`

**Abstract.** The mechanical verification of the root contention protocol from the physical layer of the IEEE1394a standard [1] for a high performance serial bus is described. In this case study, the Uppaal2k [2, 3] model checker of real-time systems is utilized. An enhanced model of the protocol is presented, which corrects for imperfections in the communication model from [4]. Using this new model, the correctness of the root contention protocol is demonstrated. The required timing constraints are established via analysis of the protocol and via approximate parameter analysis.

## 1 Introduction

Various recent studies have evidenced the maturity of automated tools for the verification of realistic applications [5, 6]. Several case studies are reported in which automatic verification tools are used to analyze IEEE standards. The first and probably best-known example is the verification of the IEEE Futurebus+ standard by Clarke and his students using SMV [20]. This verification revealed several errors that were previously undetected. In [21], using the Caesar/Aldebaran toolset, a deadlock was revealed in the draft IEEE 1394 standard.

In this work, we investigate the applicability of the latest version of Uppaal [2, 3] to analyze the IEEE 1394 root contention protocol [1], a real-time leader election protocol for two processes. We examine to what extent we can use Uppaal to do parametric analysis and verification via stepwise abstraction.

As far as we know, our case study is the first time that a timed-automata-based tool is used to analyze a (draft) IEEE standard. Given the importance of timing constraints in many of these standards, we believe this to be a significant step forward. Although our analysis did not reveal any errors, we did discover a number of minor points were the standard is incomplete.

Timing parameters play an essential role in the root contention protocol. For certain parameter values, the protocol is correct, and for other values it fails. We are interested in deriving the precise constraints that ensure correctness. There are currently two tools available that (at least in some cases, see [7]) can do parametric analysis of timed systems: HyTech [8], and PMC [9]. Since the performance of

Hytech is limited, and we expected the protocol to be too complex for it, and since only a prototype of the parametric model checker PMC is currently available, we decided to use the Uppaal tool. Uppaal has already been applied in various verifications [10-13, refs. in 3]. It can model real-time systems with a finite control structure. A limited class of properties, *viz.* reachability properties, can be checked automatically and efficiently. Our protocol models fit naturally into its input language. By analyzing numerous instances of the protocol for different values of the parameters, Uppaal allowed us to do an approximate parameter analysis.

A manual verification of the root contention protocol has been carried out in [4], where two timing constraints where inferred that ensured protocol correctness. As an introduction to this case study, we mechanically checked the proof invariants in the model from [4], using a state mapping. Our Uppaal experiments affirm that all invariants hold under the conditions mentioned in [4]. Details are available from [17]. The latter result has been established independently in [19] using the tool PMC.

The main contribution of this work, however, is the improved protocol model that is presented, which we believe now carefully reflects the IEEE 1394 specification. The main difference between the models from [4] and from this work is the way in which communication between the processes is modeled: by a package mechanism and by continuous-time signals, respectively.

Using the new, enhanced model, we investigate the correct operation of the root contention protocol with Uppaal. The constraints that we deduce for this are stricter than those in [4]. Unlike other Uppaal case studies, we have carried out the verification using stepwise abstraction, similarly to [4]. In this method, we do not relate the implementation and the specification model immediately, but use several intermediate automata. This method is common practice in automaton-based verification, but not in the context of model checkers. Unlike [4], we do not consider probabilistic aspects of the protocol in this work.

The modeling and verification effectively took us about one month. The first author has a background in applied physics, not in computer science. Most of the time has been spent in understanding the IEEE 1394 standard and in several prototype-model improvements. Time could have been saved by automating tasks, such as certain syntactic operations on the automaton models and repeated verification of the same properties for different instances of the parameter values in a model.

Section 2 informally describes the root contention protocol, and section 3 briefly introduces model checking with Uppaal. Section 4 presents the new protocol model and contains the associated verification results. Finally, in section 5, conclusions are given.

## 2 The IEEE 1394a Root Contention Protocol

The IEEE 1394-1995 standard [16] for a high performance serial bus specifies a bus that supports isochronous and asynchronous data transfers, peer-to-peer, among up to 64 high-performance devices. Also, the bus is hot-plug-and-play, which makes it suitable for interconnecting digital multimedia and consumer electronics. At this moment, the IEEE 1394a supplement [1] to the standard is about to be released, which includes several clarifications, extensions, and performance improvements.

In the IEEE 1394 standard, four protocol layers have been defined. The Physical layer (PHY) is the lowest, and provides the electrical and mechanical interface for data transmission across the bus. Furthermore, it handles bus configuration, arbitration, and data transmission. All 1394 nodes (devices) have one or more ports. Each port may be connected to one other node's port, via a bi-directional cable. Nodes should be connected in a tree-like topology, without cycles. First, bus configuration is performed. This is done automatically upon a bus reset: after power up and after device addition or removal. After that, nodes can arbitrate for access to the bus and transfer data to any node, peer-to-peer. The bus configuration process starts upon a bus reset with a bus initialization. This is followed by the tree-identify phase (Tree ID). In this phase, it is checked whether the bus topology is a tree and, if so, a leader is elected among the nodes in the tree. Finally, in the self-identify phase (Self ID), each node selects a unique physical ID and identifies itself to the other nodes.

The purpose of the Tree ID phase is to identify the root node and the topology of all attached nodes. All ports will become either child or parent ports; a child port connects to a node further away from the root, whereas a parent port connects to a node closer to the root. The root node, which only has child ports, is elected as the leader. Informally, the basic operation of the leader election protocol is as follows: each node can drive a signal via the bi-directional cable to its nearest neighbor node (with a certain propagation delay). During the Tree ID phase, a node can drive a `PARENT_NOTIFY` signal or a `CHILD_NOTIFY` signal, or leave the line undriven (`IDLE`). The `PARENT_NOTIFY` signal is to ask the other node to become parent of the sending node, which is acknowledged by a `CHILD_NOTIFY` signal in return. Upon receipt of a `CHILD_NOTIFY` signal on a port, the `PARENT_NOTIFY` signal is removed to acknowledge this receipt.

A node will only send a `PARENT_NOTIFY` signal via a port, after it has received a `PARENT_NOTIFY` signal on all other ports. Thus, initially, only the leaf nodes send out a `PARENT_NOTIFY` signal. Ports on which a `PARENT_NOTIFY` signal is received are marked as *child* ports, and ports on which a `CHILD_NOTIFY` signal is received are marked as *parent* ports. If a node has received `PARENT_NOTIFY` signals on all of its ports, it only has *child* ports and it knows that is has been elected as the root of the tree. In the final stage of the Tree ID phase, however, it is possible that upon detecting that all but one of their ports have become *child* ports, two neighboring nodes each attempt to find their parent by sending a `PARENT_NOTIFY` signal to each other. In this case, these two nodes are in *root contention*. The root contention protocol is then initiated, to select one of the two nodes as the root of the tree.

## 2.1 The Root Contention Protocol

If a node receives a `PARENT_NOTIFY` signal on a port, while sending a `PARENT_NOTIFY` signal on that port, it knows it is in root contention. Note that root contention is detected by each of the two contending nodes (Node 1 and Node 2 individually. Upon detection of root contention, a node backs off by removing the

PARENT_NOTIFY signal, and leaving the line in the IDLE state. At the same time, it starts a timer and picks a random bit. If the random bit is one, the node will wait for a time ROOT_CONTEND_SLOW, whereas if the random bit is zero, it will wait for a shorter time ROOT_CONTENT_FAST. Table 2.1 lists the wait times as specified in the latest draft version of the IEEE 1394a standard [1].

When its timer expires, a node samples its contention port once again. If it sees IDLE, it starts sending PARENT_NOTIFY anew and starts waiting to see a CHILD_NOTIFY signal in return as an acknowledgment. When the timer of a node expires and this node will sample the PARENT_NOTIFY on its port, it will send the CHILD_NOTIFY signal back as an acknowledgement and it knows that it has to take on the role of the bus root. However, in the case that both nodes pick identical random bits, there is a chance of root contention again: each node may see an IDLE signal when its timer expires and the both start sending the PARENT_NOTIFY signal. In this case, both nodes will detect renewed root contention and the whole process is repeated until one of them becomes root.

**Table 2.1**: Root contend wait times from IEEE 1394a [1].

| | *minimum* | *maximum* |
|---|---|---|
| ROOT_CONTEND_FAST | *rc_fast_min* (760 ns) | *rc_fast_max* (850 ns) |
| ROOT_CONTENT_SLOW | *rc_slow_min* (1590 ns) | *rc_slow_max* (1670 ns) |

### 2.2 Protocol Timing Constraints and their Implications

The timing parameters that are used in the protocol include the wait-times as listed in Table 2.1, and the *delay* parameter, which corresponds to the total time from sending a signal by one node to receiving it by the other node. It includes the cable propagation delay, and the time to process the cable line-states by the hardware and software layers at the ports of the two nodes. For the protocol to work correctly, two constraints on these timing parameters are essential (equations 2.1 and 2.2).

$$2 * delay < rc\_fast\_min\ . \qquad (2.1)$$

$$2 * delay < rc\_slow\_min - rc\_fast\_max\ . \qquad (2.2)$$

The origin of these equations is visualized in Figure 2.1 and explained below.

*Ad equation 2.1*: In case of Node 2 selecting the short waiting period, constraint equation 2.1 ensures that the IDLE signal from Node 1 arrives at Node 2 before the waiting period of Node 2 ends (See circle 1 in Figure 2.1). Otherwise, Node 2 might still see the first PARENT_NOTIFY signal from Node 1, and erroneously send a CHILD_NOTIFY signal to acknowledge this parent request. Once the IDLE signal from Node 1 arrives (behind schedule), Node 2 removes its CHILD_NOTIFY signal again and makes itself root. When Node 1 ends its waiting period, however, it will see the IDLE signal from Node 2, as if nothing happened, and send a PARENT_NOTIFY to Node 2. Awaiting the response it will time out, which leads to a bus reset. Therefore, constraint equation 2.1 is required for correct protocol operation.

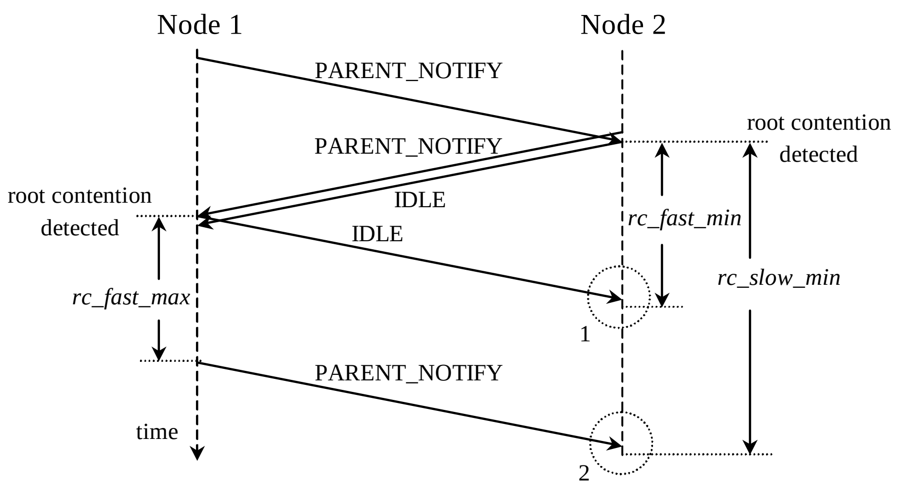

**Figure 2.1**: Visualization of the protocol timing constraints (see text for details)

*Ad equation 2.2*: In case of Node 1 selecting the short waiting period and Node 2 selecting the long waiting period, constraint equation 2.2 ensures that the new `PARENT_NOTIFY` signal from Node 1 arrives at Node 2 before the waiting period of Node 2 ends (See circle 2 in Figure 2.1). Otherwise, Node 2 might still see the `IDLE` signal from Node 1, and start sending a new `PARENT_NOTIFY` signal. Together with the `PARENT_NOTIFY` message coming from Node 1 (after schedule), this will again lead to root contention, although the two nodes picked different timers. Therefore, constraint equation 2.2 is required to ensure that root contention can only occur for a second time if both nodes pick equal random bits and wait times.

This analysis is based on informal notes [14, 15] to the IEEE-P1394a Working Group. The equations from [14] match ours, whereas [15] incorrectly cites [14] and contains errors. Note that these timing constraints do not appear in the IEEE 1394 specification [1, 16]. In [4], the above scenario cannot be mimicked, due to an imperfection in its model, and weaker constraint equations are found.

The IEEE 1394a root contention wait times, as given in Table 2.1, together with the timing constraint equations 2.1 and 2.2, imply *delay* < 370 ns. The cable propagation delay, according to the IEEE 1394a standard, should be less or equal than 5.05 ns/m. Unfortunately, any additional processing delays are not explicitly specified in the standard. Disregarding such extra delays, a maximum cable length between two nodes of 370 ns / 5.05 ns/m = 73 m is allowed. In the IEEE 1394a standard the cable length is, at the moment, limited to 4.5 m by the worst-case round-trip propagation delay during bus arbitration. The above timing constraints require the root contention times to be longer, when cable lengths increase significantly. This may be disadvantageous to future applications, and alternative root contention protocols may become necessary.

## 3 Model Checking and Uppaal

Uppaal [2, 3] is a tool box suitable for modeling, simulation and automatic verification of real-time systems, based on timed automata. In this work, the latest version of Uppaal (Uppaal2k) was used. Uppaal can simulate a model, i.e. provide a particular execution of the system, and it can automatically check whether a given (reachability) property is satisfied by the system. If the property is not satisfied, a diagnostic trace is provided showing how the property is violated. For an introduction to Uppaal, we refer to [3] which is available from the web. Section 3.1 summarizes this document. Sections 3.2 and 3.3 give a short introduction to verification based on timed automata.

### 3.1 System Description in Uppaal

In this section, an informal introduction to Uppaal is given (see [3] for details). A *system* in Uppaal is modeled as a collection of non-deterministic *processes* with a finite control structure and real-valued clocks. Communication between processes takes place via *channels* (i.e. via binary synchronization on complementary actions) or via shared integer variables. Within Uppaal it is possible to model automata via a graphical description. Furthermore, *templates* are available to facilitate the specification of multiple automata with the same control structure simultaneously.

Basically, a process is a finite-state machine (or labeled transition system) extended with *clock and integer variables*. Operations on integer variables that are available are: addition, subtraction, and multiplication with a constant. Furthermore, clock *constraints* can be set via the comparison of (differences in) clock values with an integer. The *nodes* of an automaton describe the control locations. Each location can be decorated with an invariant: a number of clock bounds expressing the clock values that are allowed that location. The *edges* of the transition system represent the changes between control locations. Each edge can be labeled by a guard, an action, and a number of clock resets and integer variable assignments. All three types of labels are optional. A *guard* is an inequality over integer and clock variables, expressing when the transition is enabled, i.e. when it can be taken. When it is taken, clock resets and integer variable assignments, if present, are executed. The action label, if present, enforces binary synchronization. This means that exactly one of the other processes should take a complementary action (where `a!` and `a?` are complementary). If no other process is able to synchronize on the action, the transition is not enabled. A process can have a state from which more than one transition is enabled, with the same action. Hence *non-deterministic* choices can be specified within Uppaal. Note that time can only elapse at the locations (conform invariants). Transitions are taken instantaneously, i.e. no time elapses during transitions. Three special types of locations are available:
*Initial locations* define the initial state of the system (exactly one per process).
*Urgent locations* are locations in which no time can be spent, hence a shorthand notation for a location that satisfies the virtual invariant `x<0`. The (fresh) clock `x` is reset on all transitions to the urgent location.

*Committed locations* are used to make the action in the incoming transition and the action on the outgoing transition atomic. Process execution cannot be interrupted and no time elapses. They can be used to encode multi-way synchronization in Uppaal.

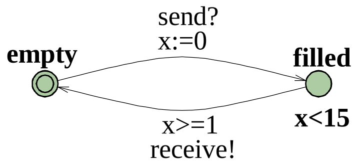

**Figure 3.1**: An example Uppaal process automaton (Buffer)

Consider the Buffer process automaton displayed in Figure 3.1. This automaton models a one-place buffer through which messages are passed with a time delay from a Sender process to a Receiver process (not shown in Figure 3.1). Data elements are delivered through the Buffer process with a time delay between 12 and 15 time units.

Uppaal is able to automatically analyze reachability properties. These properties must be of the form `E<>(p)` or `A[](p)`, where `p` is Boolean expression on clock and integer variable constraints and/or on locations of the automata. Informally, `E<>(p)` denotes that there must be some state (=location+clock and integer variable values) which is reachable from the initial state and in which the property `p` holds. Dually, `A[](p)` denotes that `p` holds for all reachable states, *i.e.* that `p` is an invariant of the automaton. This logic is sufficient to specify safety properties, invariants, and bounded liveness properties. However, general liveness and fairness properties, and the repeated occurrence of an event, infinitely often, cannot be expressed in Uppaal.

## 3.2 Theoretical Background

Given the control location and the values of the clock and data variables of an automaton, we can determine whether a transition can be taken or whether time can elapse in this location. This is expressed by the associated *timed labeled transition system* (TLTS) to an automaton. The states of the TLTS consists of a *control location*, a *clock valuation* and a *data valuation*. A valuation is function that assigns a value to each (in this case clock and data) variable. If the automaton consists of several components, then the control location is in fact a tuple containing the control locations of each component. The TLTS is equipped with time passage actions $s \rightarrow^{d} s'$ that augment the clocks in $s$ with $d$ time units and leave the location unchanged, resulting in the state $s'$. Such a transition is enabled in the TLTS if augmenting the clocks with $d$ time units is allowed by the state invariant in $s$. The discrete actions (where internal actions of the automaton are labeled with $\tau$) $s \rightarrow^{a} s'$ change the control location and update the data variables, leaving the clocks untouched. It is enabled if there exists a $a$-transition in the Uppaal automaton, such that the guard of this transition is set for the clock values in $s$. Hence, the TLTS has an infinite set of states, actions and – for non-trivial automata – transitions. The initial state of the TLTS is given by the initial location and the clock and data valuations that assign zero to each variable. A *timed execution* of a TLTS is a possibly infinite sequence $s_0$, $a_1$, $s_1$, $a_1$, … such that for all $i$, $s_i$ is a state and $a_i$ is a (discrete or delay) action, and $s_i \rightarrow^{a_{i+1}} s_{i+1}$ is a transition. A timed execution of an Uppaal model is a timed execution of its underlying TLTS – a sequence ($s_0$, $c_0$, $d_0$), $a_0$, ($s_1$, $c_1$, $d_1$), $a_1$, ($s_2$, $c_2$, $d_2$), … where $s_i$, $c_i$, $d_i$ are a location, clock valuation and a data valuation respectively.

**Notation 3.1**: *The sets of timed traces of a TLTS or an automaton A are both denoted by* ttrace($A$). *We write* $A \sqsubseteq_{\mathrm{TTR}} B$ *if* ttrace($A$) $\subseteq$ ttrace($B$).

A timed trace (of either an automaton or a TLTS) arises from an execution by omitting the states and internal actions. A state is *reachable* if there exists an execution passing through it. Theorem 3.2 expresses the main result by Alur and Dill [18], upon which Uppaal and other model checkers, based on timed automata, are built:

**Theorem 3.2**: *The set of reachable states of a timed automaton is decidable*.

An important class of automata and TLTS are the deterministic systems. Intuitively, determinism means that given the current state (for an automaton this includes the location and clock values.) and the action to be taken, the next state (if any) is uniquely determined.

**Definition 3.5**: *1. A TLTS is called* deterministic *if for every state s and every action label a there is at most one transition* $s \rightarrow^{a} t$*, and there are no* $\tau$*-transisions.*
*2. An automaton is called* deterministic *if for every state s and every action label a,* $s \rightarrow^{(a,g,Y)} t$ $s \rightarrow^{(a,g',Y')} t'$ *with* $t \neq t'$ *implies that g and g' are disjoint, (i.e.* $g \wedge g'$ *is unsatisfiable).*

It is not difficult to prove that if an automaton is deterministic then its underlying TLTS is so.

## 3.3 Verification of Timed Trace Inclusion

Within automaton-based verification, it is common practice to give both the implementation and the specification of the protocol as automata. One automaton *A* is said to implement another one *B* if $A \sqsubseteq_{\mathrm{TTR}} B$. The rest of this section describes how this relation can be checked by Uppaal.

Although in general it is undecidable whether $A \sqsubseteq_{\mathrm{TTR}} B$, Alur and Dill [18] have shown that deciding whether $A \sqsubseteq_{\mathrm{TTR}} B$ can be reduced to reachability checking, provided that *B* is deterministic. This is done via the automaton $B^{err}$, which adds a state *error* to *B* and transitions $s \rightarrow^{(a,g)}$ *error* for all states *s* and *a* such that this transition is enabled if no other *a*-transition is enabled from *s*, or if the invariant in *s* does not hold. All state invariants are removed.

**Proposition 3.6**: *Assume that B is deterministic. Then* $A \sqsubseteq_{\mathrm{TTR}} B \Leftrightarrow B^{err}$ *is reachable in the composition of A and* $B^{err}$.

If *B* is non-deterministic, then we can try to make it so by renaming the labels and then use the above method. The following result states that, after a consistent re-labeling of *A* and *B* into *C* and *D*, respectively, $C \sqsubseteq_{\mathrm{TTR}} D \Rightarrow A \sqsubseteq_{\mathrm{TTR}} B$.

**Lemma 3.7**: *Suppose that A, B, C, and D are TLTSs, such that C and D have the same set of action labels. h is a (re-labeling) function mapping the discrete actions of*

*C to those of A, with h*(τ) = τ. *Suppose that:*

1. *A is obtained from C by replacing every discrete action a in C by h(a),*
2. *Similarly, B is obtained from C by applying replacing every discrete action a in D by h(a).*

*Then* $C \sqsubseteq_{\mathrm{TTR}} D \Rightarrow A \sqsubseteq_{\mathrm{TTR}} B$.

The problem is now to find for given automata *A*, and *B*, an automaton *C* and a deterministic automaton *D* such that their underlying TLTSs fulfill the conditions from lemma 3.7. In our verification, we constructed *C* from *A* and a step refinement *r*, yielding an automaton $A^r$, and *D* from *B* and *r*, yielding $B^r$. Informally, a step refinement is a function *r* (on the TLTSs) from the state space of *A* to the state space of *B* such that every transition $s \rightarrow^a t$ of *A* can be mimicked in the second by the transition $r(s) \rightarrow^a r(t)$ of *B*, where we also allow internal transitions in *A* to be mimicked by remaining in the same state in *B*. We conjecture that the method we used can be generalized and that it is possible to construct in that way the desired automata *C* and *D* from a step refinement from *A* to *B* in finite representation. Although the existence of a step refinement already implies timed trace inclusion, this construction would yield a method to check this with Uppaal.

## 4 The Enhanced Protocol Model

A manual verification of the root contention protocol is described in [4]. However, a major difference between the model in [4] and the enhanced model presented in this work lies in the way in which communication between the nodes across the wires is handled. In [4], this is modeled as the transfer of single messages (`PARENT_NOTIFY` or `CHILD_NOTIFY`) that are sent only once, and upon reception removed from the wire. Also, messages can be overwritten and lost. This abstraction is inappropriate, since in IEEE 1394, communication is done via signals continuously being driven across the wire. These signals persist at the input-port of the receiving node, until the sending node changes its output-port signal. Besides driving `PARENT_NOTIFY` and `CHILD_NOTIFY` signals, the wire can be left undriven (`IDLE`). We believe that the enhanced model presented in this work now adequately reflects the root contention protocol as specified in the draft IEEE 1394a standard [1]. With this model, we will show that constraints 2.1 and 2.2 are both necessary and sufficient for the correctness of the protocol.

### 4.1 The Protocol Model Automata

Figures 4.1 and 4.2 display the Uppaal templates of the `Node` and `Wire` automata of the enhanced model. These template automata are instantiated to a total system (`Impl`) of two nodes `Node1` and `Node2`, connected by a bi-directional communication channel (`Wire12` for messages from `Node1` to `Node2`, and `Wire21` for the opposite direction). The Uppaal model files can be found on the web [17].

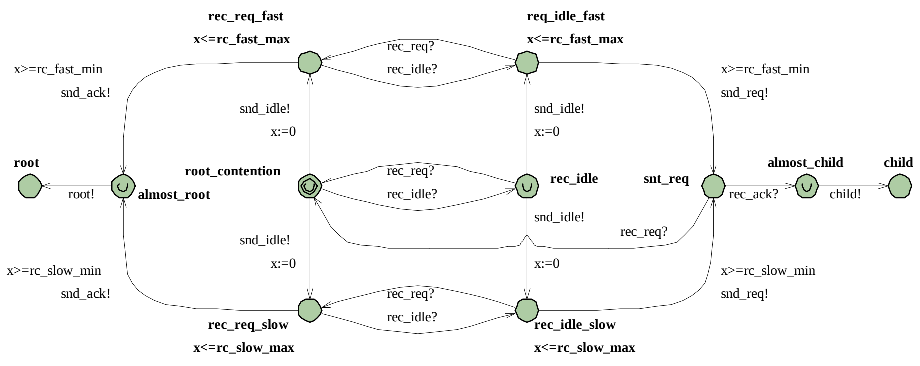

**Figure 4.1**: The Uppaal Node automaton template

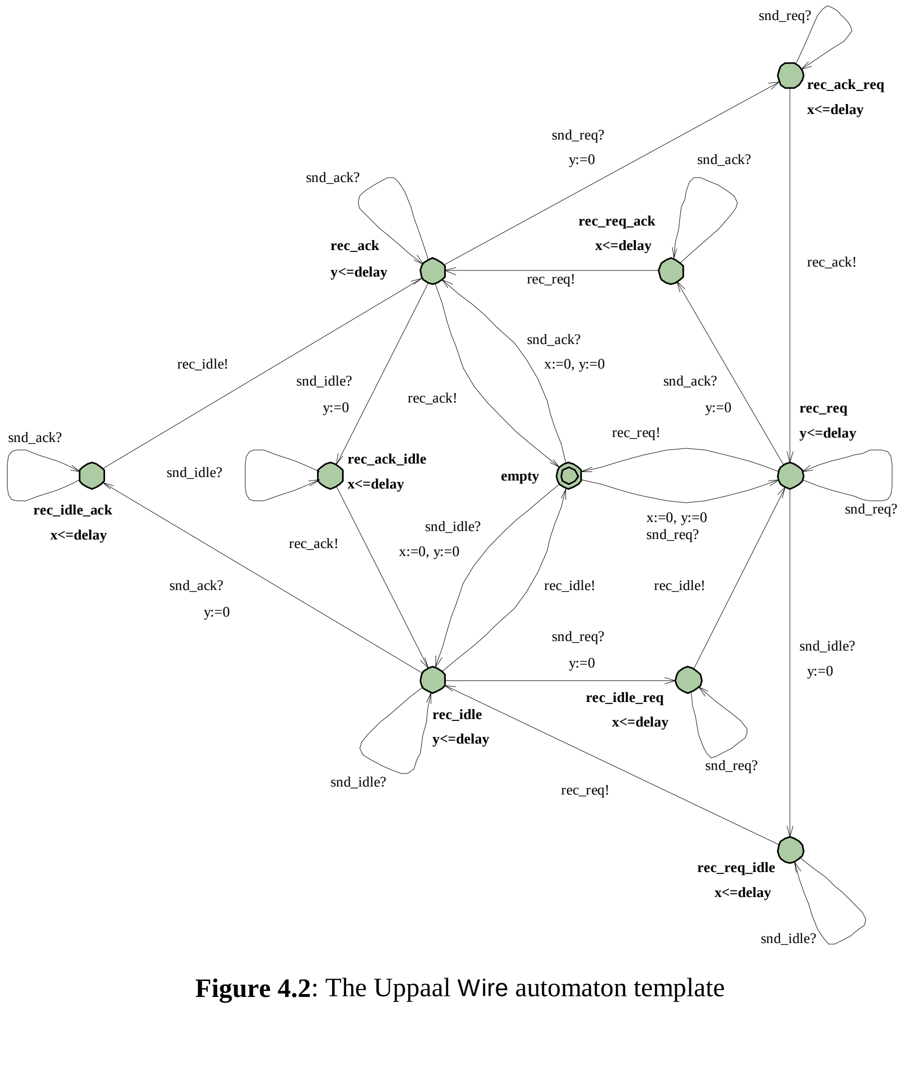

**Figure 4.2**: The Uppaal Wire automaton template

First of all, the PARENT_NOTIFY message has been abbreviated to req, and the CHILD_NOTIFY message to ack. A number of timing parameter constants is defined to include the root contention wait times and the cable propagation delay into the model. The root contention wait times, like rc_fast_min, have been set to the values as specified in Table 2.1. The channels like snd_ack and rec_req are used to send and receive ack and req messages by the nodes through the wires. The slow/fast differentiation causes the Node automaton to be rather symmetric. The root and child channels are used to audit the root/child status of each node, via a dummy AuditStatus automaton.

A key property of I/O automata is that they are always input-enabled, i.e. no input is blocked. The Uppaal Node automata were also made input-enabled, by adding both an *error* state and synchronization transitions to this *error* state from all states in which input (via rec_req and rec_ack) would otherwise be blocked. The verification that the error state in this Node$^{err}$ automata cannot be reached, implies equivalence between this input-enabled Node$^{err}$ automaton and the Node automaton without the error state, as shown in Figure 4.1.

Starting in the root contention state, a Node picks a random bit (slow or fast). Simultaneously timer x is reset, and an idle message is sent to the Wire, which models the removal of the PARENT_NOTIFY signal. Independently, but at about the same time, the contending Node also sends an idle, possibly followed by a renewed req. Therefore, the reception of this idle and req message is interleaved with the choice of the random bit and with the sending of the idle message. In this way, the two contending Node automata can operate autonomously.

The Wire templates have been extended, compared to the model in [4], such that they can now transmit PARENT_NOTIFY (req), CHILD_NOTIFY (ack), and IDLE (idle) messages. These messages mark the leading edge of a new signal being driven across the wire. Until a new message arrives, signals continue to be driven across the wire. Furthermore, the wires now comprise a two-place buffer, such that two messages at the same time can travel across a wire. Using an additional *error*-state, which has been left out in Figure 4.2 for clarity, it is shown (error state unreachable) in section 4.2 that a two-place buffer suffices for each Wire.

## 4.2 Verification of the Enhanced Model

A key correctness property of the root contention protocol is that eventually, exactly one of the processes is elected as root. This property is described by the automaton Spec in Figure 4.3. We demonstrate that Impl (the parallel composition of the two Node and Wire automata) is a correct implementation of Spec provided that Impl meets the timing constraint equations 2.1 and 2.2.

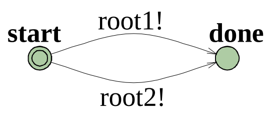

**Figure 4.3**: The Uppaal Spec automaton of the root contention protocol

The stepwise abstraction from [4], which is a widely used method in automaton-based verification, since it allows for separation of concerns, is mimicked, in order to explore the applicability of Uppaal to the method of stepwise abstraction. Rather than proving $\mathsf{Impl} \sqsubseteq_{\mathrm{TTR}} \mathsf{Spec}$ at once, we consider three intermediate automata $\mathsf{I}_1$, $\mathsf{I}_2$, and $\mathsf{I}_3$ (from [4]) and use Uppaal to verify that $\mathsf{Impl} \sqsubseteq_{\mathrm{TTR}} \mathsf{I}_1 \sqsubseteq_{\mathrm{TTR}} \mathsf{I}_2 \sqsubseteq_{\mathrm{TTR}} \mathsf{I}_3 \sqsubseteq_{\mathrm{TTR}} \mathsf{Spec}$. Here, $\mathsf{I}_1$ is a timed automaton, which abstracts from all message passing in $\mathsf{Impl}$ while preserving the timing information of $\mathsf{Impl}$. The automaton $\mathsf{I}_2$ is obtained from $\mathsf{I}_1$ by removing all timing information. In $\mathsf{I}_3$ internal choices are further contracted. Since timing aspects are only present in $\mathsf{Impl}$ and $\mathsf{I}_1$, the timing parameters only play a role in the first inclusion ($\mathsf{Impl} \sqsubseteq_{\mathrm{TTR}} \mathsf{I}_1$). The necessity of the timing constraints will be established below.

## The First Intermediate Automaton

The intermediate automaton $\mathsf{I}_1$ is displayed in Figure 4.4. It is a timed automaton variant of the timed I/O automaton model from [4], restricted to the reachable states. It abstracts from the communication between the nodes and records the status (*start*, *fast*, *slow*, or *done*) for each of the two nodes. Also, $\mathsf{I}_1$ has a clock $\mathsf{x}$ to impose timing constraints between events. The outgoing internal transitions from *start_start*, *fast_start*, *start_fast*, *start_slow*, and *slow_start* model the consecutive random bit selection of the two nodes. For example, *fast_start* corresponds to `Node1` having picked the *fast* random bit, and `Node2` still in root contention. The internal transitions from *fast_fast* and from *slow_slow* back to *start_start* represent the protocol restart, which is an option if the two random bits are equal.

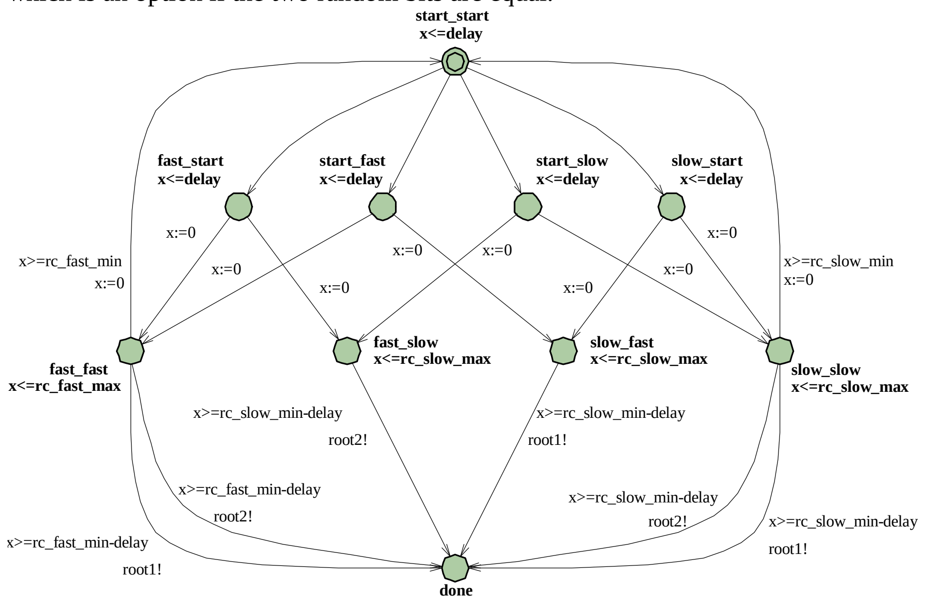

**Figure 4.4**: The Uppaal $\mathsf{I}_1$ automaton of the root contention protocol

The invariants on clock `x` cause both nodes to pick a random bit within time interval *delay* after the protocol (re-)start. Also, within an interval [*rc_fast_min* - *delay*, *rc_fast_max*] or [*rc_slow_min* - *delay*, *rc_slow_max*], which depends on the random bit, either a root is selected (`root1!` or `root2!`) or a restart of the protocol occurs.

In line with section 3.2, unlabeled transitions in $\mathsf{I}_1$ are labeled, such that they synchronize with Impl (see Figure 4.1 and 4.2) according to the step refinement from Impl to $\mathsf{I}_1$ (not discussed here). The transitions in `Node1` and `Node2` of the random bit selection (`snd_idle!, x:=0`) are coupled to the corresponding transitions of random bit selection in $\mathsf{I}_1$, according to the fast/slow differentiation. Furthermore, an auxiliary automaton EchoRetry synchronizes with the protocol restart transitions in $\mathsf{I}_1$, as soon as both `Node1` and `Node2` have taken their `snd_req!` transition to the *snt_req* state. This results in the automata $\mathsf{Impl}^r$ and $\mathsf{I}_1^r$. It is thus required that the `Node` and `Wire` automaton from Impl, and the $\mathsf{I}_1$ automaton are able to concurrently synchronize on the same label. Although Uppaal does not support multi-way synchronization, this was solved via auxiliary committed states and/or automata.
Manual parameter analysis shows that the *error*-state is only reachable in $\mathsf{Impl}^r \parallel \mathsf{I}_1^r$, if the constraint equations 2.1 and 2.2 do not hold. Therefore, as a direct consequence of Lemma 3.7, Proposition 4.1 holds:

**Result 4.1**: *If the timing parameters in* Impl *satisfy equations 2.1 and 2.2, then* $\mathsf{Impl} \sqsubseteq_{TTR} \mathsf{I}_1$.

Although from these experiments, the converse cannot be concluded, it is not difficult to come up with scenarios that show the necessity of the timing constraints for protocol correctness. In Uppaal, we have checked that messages can get lost in the wire if equation 2.1 does not hold, and that if equation 2.2 does not hold, renewed root contention may also occur, in case the nodes pick different random bits. We conclude Proposition 4.2:

**Result 4.2**: *If the parameters in* Impl *do not satisfy equations 2.1 and 2.2,* Impl *is not a correct implementation of* Spec.

### The Second Intermediate Automaton

The intermediate automaton $\mathsf{I}_2$ is identical to $\mathsf{I}_1$, except that all timing information has been removed. Although timed trace inclusion is obvious, labeling internal transitions with identical source and target locations in $\mathsf{I}_1$ and $\mathsf{I}_2$ with the same, unique labels yields Proposition 4.3, as expected:

**Proposition 4.3**: $\mathsf{I}_1 \sqsubseteq_{TTR} \mathsf{I}_2$.

### The Third Intermediate Automaton

Figure 4.5 shows intermediate automaton $\mathsf{I}_3$, in which internal choices have been further contracted. Selection of the two random bits is no longer represented via separate, subsequent transitions, but done at once via a single transition. Again, we added labels to certain unlabeled transitions in $\mathsf{I}_2$ and $\mathsf{I}_3$.

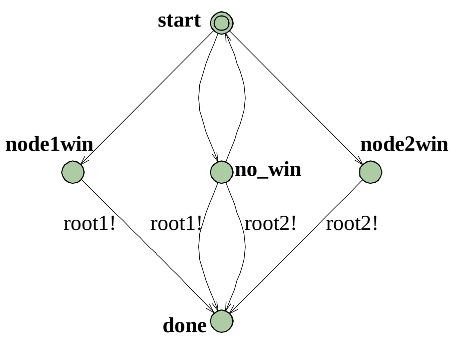

**Figure 4.5**: The Uppaal $I_3$ automaton of the root contention protocol

For example, the transition in $I_2$ from *fast_start* to *fast_fast* is coupled to the transition in $I_3$ from *start* to *no_win*. Notice that the transitions in $I_2$ leaving from *start_start* remain unlabeled. It has been established by Uppaal that $I_2^r \sqsubseteq_{TTR} I_3^r$. As an immediate corollary of Lemma 3.7, Proposition 4.5 holds:

**Proposition 4.5**: $I_2 \sqsubseteq_{TTR} I_3$.

Since the specification automaton Spec is deterministic, we need only to check for reachability in the automaton $\mathsf{Spec}^{err}$ to obtain Proposition 4.6:

**Proposition 4.6**: $I_3 \sqsubseteq_{TTR}$ Spec.

This analysis results in the final conclusion as stated in Proposition 4.7:

**Result 4.7**: *The root contention protocol is correct if and only if the parameters in* Impl *satisfy equations 2.1 and 2.2.*

## 6 Conclusions

In this case study, the mechanical verification of the IEEE 1394 root contention protocol using the Uppaal2k tool is presented. The manual verification of this protocol using I/O automata has been published in [4].

First, an enhanced Uppaal2k model of the root contention protocol is presented, that resolves imperfections in the model from [4]. With a new concept of signal communication, it now fully expresses the IEEE 1394a standard.

Second, the correctness of the root contention protocol was verified, using stepwise abstraction via intermediate automata, in analogy with [4]. Via manual parameter analysis within Uppaal2k, constraints on the timing parameters for the correctness of the protocol, like the maximum communication delay, were deduced. These constraints exactly match the theoretical constraints, as discussed. Fortunately, the current IEEE 1394a standard also meets these constraints.

Although Uppaal has not been designed for the applying method of stepwise abstraction, in our case study, this was quite easy. We recommend that, to fully exploit the method in Uppaal, the associated theory needs to be elaborated, and tool support should be provided, such as the automatic construction of the automata $A^{err}$ and, if possible, $A^r$.

Third, it is concluded that model checkers like Uppaal are useful to study protocols and distributed algorithms, also in the presence of parameters. Especially the iterative modeling via trial and error, the exploration of protocol parameters, and the checking of invariant equations, are valuable. Once an appropriate model has been obtained, more complex aspects, such as rigorous parameter analysis and probabilistic facets can be analyzed.

Finally, the fact that the modeling and verification took us a relatively short time, illustrates that model checkers can be used effectively in the design and evaluation of industrial protocols.

## References

1. IEEE Computer Society, *P1394a Draft Standard for a High Performance Serial Bus (Supplement)*, Draft 4.0, 15 September 1999.
2. Uppaal, version Uppaal2k, web pages, http://www.docs.uu.se/docs/rtmv/uppaal/ .
3. K.G. Larsen, P.Pettersson, and W. Yi, *Uppaal in a Nutshell*, Springer International Journal of Software Tools for Technology Transfer 1, pp. 1-2, 1997. (http://www.docs.uu.se/docs/rtmv/papers/lpw-sttt97.pdf)
4. M.I.A. Stoelinga and F.W. Vaandrager, *Root Contention in IEEE 1394*, Proc. of the 5th AMAST Workshop on Real-Time and Probabilistic Systems (ARTS '99), Bamberg, Germany, LNCS 1601, 1999.
5. J.M.T. Romijn, *Analysing Industrial Protocols with Formal Methods*, Ph.D. Thesis, University of Twente, 1999 (ISBN 90-9013086-1), http://www.cs.kun.nl/~judi .
6. E.M. Clarke & J. Wing, *Formal Methods: State of the Art and Future Directions*, ACM Computing Surveys 28(4), 1996.
7. R. Alur, T.A. Henzinger, and M.Y. Vardi, *Parametric Real-time Reasoning*, 25th Annual ACM Symposium on Theory of Computing (STOC 1993), pp. 592-601.
8. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: *A Model Checker for Hybrid Systems*, Software Tools for Technology Transfer 1, pp. 110-122, 1997 (see also http://www-cad.eecs.berkeley.edu/~tah/HyTech/)
9. *PMC: Prototype Model Checker*, http://tvs.twi.tudelft.nl/toolset.html.
10. K. Havelund, A. Skou, K.G. Larsen and K. Lund, *Formal Modelling and Analysis of an Audio/Video Protocol: An Industrial Case Study Using UPPAAL*, Proc. of the 18th IEEE Real-Time Systems Symposium, San Francisco CA, USA, 1997, pp. 2-13.
11. F.W. Vaandrager, *Analysis of a Biphase Mark Protocol with Uppaal*, Technical Report CSI-R99XX, Computing Science Institute, University of Nijmegen, 1999. To appear.
12. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans, *The bounded retransmission protocol must be on time!*, Proc. of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Enschede, The Netherlands, LNCS 1217, pp. 416-431, 1997.
13. K.G. Larsen, P. Petterson, and Wang Yi, *Diagnostic Model-Checking for Real-Time Systems*, Proc. of the 16th IEEE Real-Time Systems Symposium, 1995, pp. 575-586.
14. Dave LaFollette, *SubPHY Root Contention*, Sheets, 1 August 1997, (see for example ftp://ftp.symbios.com/pub/standards/io/1394/P1394a/Documents/97-043r0.pdf).
15. Takayuki Nyu, *Modified Tree-ID Process for Long-haul Transmission and Long PHY_DELAY*, Sheets, (see for example ftp://ftp.symbios.com/pub/standards/io/1394/P1394a/Documents/97-051r1.pdf).

16. IEEE Computer Society, *IEEE Standard for a High Performance Serial Bus* (Std 1394-1995), 30 August 1996.
17. www.cs.kun.nl/~marielle/ .
18. R. Alur and D.L. Dill, *A theory of timed automata*, Theoretical Computer Science 126, pp. 183-235, 1994.
19. *Case study parametric model checking: Root contention in IEEE 1394 (Firewire)*, http://tvs.twi.tudelft.nl/cases/firewire.html
20. K.L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, 1993.
21. 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 2(1), pp. 68-88, 1998.