Root Contention in IEEE 1394

Mariëlle Stoelinga and Frits Vaandrager

Computing Science Institute, University of Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
{marielle,fvaan}@cs.kun.nl

Abstract. The model of probabilistic I/O automata of Segala and Lynch is used for the formal specification and analysis of the root contention protocol from the physical layer of the IEEE 1394 ("FireWire") standard. In our model of the protocol both randomization and real-time play an essential role. In order to make our verification easier to understand we introduce several intermediate automata in between the implementation and the specification automaton. This allows us to use very simple notions of refinement rather than the more general but also very complex simulation relations which have been proposed by Segala and Lynch.

1 Introduction

Recently, the analysis of probabilistic, distributed algorithms and protocols has gained new attention. Various methods and formalisms have been extended with probabilities, and several case studies have been carried out using these formalisms, c.f. [14, 17].

This report verifies a small sub-protocol of IEEE 1394, called root contention. 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 [9] 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. Various parts of IEEE have been specified and/or verified formally, see for instance [5, 11, 12]. However, as far as we know, root contention has not.

Root contention in IEEE 1394 is a simple but realistic protocol that involves both real-time and randomization. The verification in this report is carried out in the probabilistic automaton model of Segala and Lynch [18, 20]. Following the tradition, the correctness of the protocol is proven by establishing a probabilistic simulation between the implementation and the specification, both probabilistic automata.

The probabilistic simulation relations from [18, 20] are rather complex. In order to simplify the simulation proofs, this report introduces the notions of probabilistic step refinement and of probabilistic hyperstep refinement. These are special case of the simulations in [18, 20].

The strategy followed in the simulation proof is the following. Given the protocol automaton `Impl` and the abstract specification `Spec`, we define three intermediate automata `I1`, `I2`, and `I3`. First, `I1` abstracts from the message passing in `Impl` but keeps the same probabilistic choices and most of the timing information. Next, `I2` abstracts from all the timing information in `Impl`, and `I3` abstracts from the probabilistic choice in `I3`. The introduction of the intermediate automata allows us to separate our concerns. The simulation between `Impl` and `I1` is easy from probabilistic point of view and its proof mainly involves traditional, non-probabilistic techniques like proving invariants. The remaining simulations between automata `I2`, `I3` and `Spec` deal with probabilistic choice, but since these automata are small this is not so difficult anymore.

This paper is organized as follows. After some mathematical preliminaries in Section 2, Section 3 introduces the probabilistic automaton model. Section 4 describes the root contention protocol, both informally and formally. Then Section 5 defines the intermediate automata and established the simulation relations. Finally, Section 6 presents the conclusions and some topics for future research.

2 Probability Distributions

This section recalls a few basic notions from probability theory and introduces some notation.

Definition 1. *Let $\mathcal{I}$ be an index set and let $x_i \in [0, \infty]$ for all $i \in \mathcal{I}$. Define $\sum_{i\in\mathcal{I}} x_i$ by*

1. $\sum_{i\in\emptyset} x_i \triangleq 0$
2. $\sum_{i\in\mathcal{I}} x_i \triangleq x_{i_1} + x_{i_2} + x_{i_3} + \cdots + x_{i_n}$, *if* $\mathcal{I} = \{i_1, i_2, i_3, \ldots, i_n\}$ *is a finite set with* $n > 0$ *elements*
3. $\sum_{i\in\mathcal{I}} x_i \triangleq \sup\{\sum_{i\in\mathcal{J}} x_i \mid \mathcal{J}\subseteq\mathcal{I}$ *is finite*$\}$, *if* $\mathcal{I}$ *is infinite.*

Here $\sup X$ denotes the supremum of X. Notice that $\sum_{i\in\mathbb{N}} x_i = \sum_{i=0}^{\infty} x_i$ because the summation order is irrelevant, due to the fact that $x_i \geq 0$.

Definition 2. *A* probability distribution *over set X is a function $\mu : X \to [0, 1]$ such that $\sum_{x\in X} \mu(x) = 1$. We write $support(\mu) \triangleq \{x \in X \mid \mu(x) > 0\}$. It follows from the definitions that this is a countable set. We denote the set of all probability distributions over X by $\Pi(X)$.*

We denote a probability distribution μ on a countable domain by enumerating it as a set of pairs. So, if $\mathrm{Dom}(\mu) = \{x_1, x_2 \ldots\}$ then denote μ by $\{x_1 \mapsto f(x_1), x_2 \mapsto f(x_2) \ldots\}$. If the domain of μ is known, then we often leave out elements of probability zero. For instance, the probability distribution assigning probability one to an element $x \in X$ is denoted by $\{x \mapsto 1\}$, irrespective of X. Such distribution is called the *Dirac distribution* over x. The *uniform distribution* over a finite set with $n > 0$ elements, say $\{x_1, \dots, x_n\}$, is given by $\{x_1 \mapsto \frac{1}{n}, \dots, x_n \mapsto \frac{1}{n}\}$.

Definition 3. *Let X and Y be sets, $\mu \in \Pi(X)$ and $\nu \in \Pi(Y)$. The* product *of μ and ν, notation $\mu \times \nu$, is the probability distribution $\kappa : X \times Y \to [0,1]$ satisfying $\kappa(x,y) = \mu(x) \cdot \nu(y)$.*

Definition 4. *Let X and Y be sets, $\mu \in \Pi(X)$ and $f : X \to Y$. The* image *of μ under f, notation $f_*(\mu)$, is the probability distribution $\nu \in \Pi(Y)$ satisfying $\nu(y) = \sum_{x \in f^{-1}(y)} \mu(x)$.*

3 Probabilistic Automata

This section presents the model of probabilistic automata and two extensions, probabilistic I/O automata and timed probabilistic I/O automata. We assume that the reader is familiar with non-probabilistic (timed) automata and their simulation relations, see e.g. [13, 15] for an introduction and for the notations used in this paper.

3.1 The Basic Model

This section recalls the basic probabilistic automaton model from [18, 20], and introduces the notions of probabilistic step refinement and probabilistic hyper-step refinement.

Definition 5. *A* probabilistic automaton *A consists of four components:*

1. *A set $states_A$ of* states.
2. *A nonempty set $start_A \subseteq states_A$ of* start states.
3. *An* action signature *$sig_A = (ext_A, int_A)$, consisting of* external *and* internal actions *respectively; we define the set of* actions *as $act_A \triangleq ext_A \cup int_A$.*
4. *A* transition relation *$trans_A \subseteq states_A \times act_A \times \Pi(states_A)$.*

We write $s \xrightarrow{a}_A \mu$ for $(s, a, \mu) \in trans_A$, and $s \xrightarrow{a}_A s'$ for $s \xrightarrow{a}_A \{s' \mapsto 1\}$.

Sometimes, a more general definition of probabilistic automata is given where $trans_A \subseteq states_A \times \Pi(act_A \times states_A)$. In this context the probabilistic automata from the definition are called *simple* probabilistic automata.

Definition 6. *Let A be a probabilistic automaton. The automaton A^-, the* non-probabilistic variant *of A, which behaves like A but discards all probabilistic information, is defined by:*

1. *$states_{A^-} = states_A$.*

2. $start_{A^-} = start_A$.
3. $sig_{A^-} = sig_A$.
4. $trans_{A^-} = \{(s, a, s') \mid \exists \mu : s \xrightarrow{a}_A \mu \wedge \mu(s') > 0\}$.

Define $reach_A$, the set of reachable states *of A, to be the set of reachable states of A^-.*

An execution (execution fragment, trace) *of a probabilistic automaton A is an execution (execution fragment, trace) of A^-. The set of executions (execution fragments, traces) and finite executions (execution fragments, traces) of A are respectively denoted by $execs(A)(frags(A), traces(A))$ and by $execs^*(A)$ $(frags^*(A), traces^*(A))$.*

Definition 7. *If A is a probabilistic automaton and $X \subseteq ext_A$, then $hide(A, X)$ is the probabilistic automaton $(states_A, start_A, (ext_A \setminus X, int_A \cup X), trans_A)$.*

Definition 8. *We say that two probabilistic automata A_1 and A_2 are* compatible *if $int_{A_1} \cap act_{A_2} = act_{A_1} \cap int_{A_2} = \emptyset$. If A_1 and A_2 are compatible then their* parallel composition, *notation $A_1 \parallel A_2$, is the probabilistic automaton A defined by:*

- $states_A = states_{A_1} \times states_{A_2}$.
- $start_A = start_{A_1} \times start_{A_2}$.
- $sig_A = (ext_{A_1} \cup ext_{A_2}, int_{A_1} \cup int_{A_2})$.
- *$trans_A$ is the set of triples $((s_1, s_2), a, \mu_1 \times \mu_2)$ such that for $i = 1, 2$, if $a \in act_{A_i}$ then $(s_i, a, \mu_i) \in trans_{A_i}$, otherwise $\mu_i = \{s_i \mapsto 1\}$.*

Informally, within a composition two probabilistic automata synchronize on their common actions and evolve independently on others. Whenever synchronization occurs, the state reached is obtained by choosing a state independently for both automata.

Probabilistic Step Refinements The simplest form of simulations between probabilistic automata that we consider are the probabilistic step refinements. These are mappings from the states of one automaton to the states of another automaton that preserve initial states and probabilistic transitions.

Definition 9. *Let A and B be two probabilistic automata with $ext_A = ext_B$. A* probabilistic step refinement *from A to B is a function $r : states_A \to states_B$ such that:*

1. *for all $s \in start_A, r(s) \in start_B$;*
2. *for all steps $s \xrightarrow{a}_A \mu$ with $s \in reach_A$, one of the following conditions holds:*
   (a) $r(s) \xrightarrow{a}_B r_*(\mu)$, or
   (b) $a \in int_A \wedge r(s) \xrightarrow{b}_B r_*(\mu)$, for some $b \in int_B$, or
   (c) $a \in int_A \wedge r_*(\mu) = \{r(s) \mapsto 1\}$.

We write $A \sqsubseteq_{\mathrm{PSR}} B$ if there is a probabilistic step refinement from A to B. Note that condition 2(c) is equivalent to $a \in int_A \wedge \forall s'[\mu(s') > 0 \Rightarrow r(s') = r(s)]$.

Probabilistic Hyperstep Refinements Probabilistic hyperstep refinements generalize the probabilistic step refinements introduced above. They are a special case of the probabilistic forward simulations of Segala and Lynch [18, 20].

Definition 10. *Let X, Y be sets and $R \subseteq X \times \Pi(Y)$. The* lifting *of R is the relation $R_{**} \subseteq \Pi(X) \times \Pi(Y)$ given by: $(\mu, \nu) \in R_{**}$ if and only if there is a choice function $r : support(\mu) \to \Pi(Y)$ for R, i.e., a function such that $(x, r(x)) \in R$ for all $x \in support(\mu)$, satisfying*

$$\nu(y) = \sum_{x \in \mathrm{supp}(\mu)} \mu(x) \cdot r(x)(y).$$

The idea is that we obtain ν by choosing the probability distribution $r(x)$ with probability $\mu(x)$.

Example 1. Given a probabilistic automaton A and an action $a \in act_A$, we can lift the relation $\xrightarrow{a}$ over $states_A \times \Pi(states_A)$ to the relation $\xrightarrow{a}_{**}$ over $\Pi(states_A) \times \Pi(states_A)$. For instance, if $s_1 \xrightarrow{a} \mu_1$, $s_2 \xrightarrow{a} \mu_2$ and $s_1 \neq s_2$, then

$$\{s_1 \mapsto \tfrac{1}{3}, s_2 \mapsto \tfrac{2}{3}\} \xrightarrow{a}_{**} \tfrac{1}{3} \cdot \mu_1 + \tfrac{2}{3} \cdot \mu_2.$$

Intuitively, if $s_1 \xrightarrow{a} \mu_1$, $s_2 \xrightarrow{a} \mu_2$ and the probability to be in s_1 is $\frac{1}{3}$ and to be in s_2 is $\frac{2}{3}$, then we choose the next state according to μ_1 with probability $\frac{1}{3}$ and according to μ_2 with probability $\frac{2}{3}$. If there is another a–transition, say $s_2 \xrightarrow{a} \nu$, then we can also choose the next state according to μ_1 with probability $\frac{1}{3}$ and according to ν with probability $\frac{2}{3}$. Hence

$$\{s_1 \mapsto \tfrac{1}{3}, s_2 \mapsto \tfrac{2}{3}\} \xrightarrow{a}_{**} \tfrac{1}{3} \cdot \mu_1 + \tfrac{2}{3} \cdot \nu.$$

We do *not* have

$$\{s_1 \mapsto \tfrac{1}{3}, s_2 \mapsto \tfrac{2}{3}\} \xrightarrow{a}_{**} \tfrac{1}{3} \cdot \mu_1 + \tfrac{1}{3} \cdot \mu_2 + \tfrac{1}{3} \cdot \nu.$$

Definition 11. *Let A and B be probabilistic automaton with $ext_A = ext_B$. A* probabilistic hyperstep refinement *from A to B is a function $h : states_A \to \Pi(states_B)$ such that:*

1. *for all $s \in start_A$, $h(s) = \{s' \mapsto 1\}$ for some $s' \in start_B$;*
2. *for all steps $s \xrightarrow{a}_A \mu$ with $s \in reach_A$, one of the following conditions holds:*
   (a) $h(s) \xrightarrow{a}_{B**} h_{**}(\mu)$, or
   (b) $a \in int_A \wedge h(s) \xrightarrow{b}_{B**} h_{**}(\mu)$, for some $b \in int_B$, or
   (c) $a \in int_A \wedge h(s) = h_{**}(\mu)$.

Write $A \sqsubseteq_{\mathrm{PHSR}}$ if there is a probabilistic hyperstep refinement from A to B.

Segala [18] describes the behavior of probabilistic automata in terms of *trace distributions*, and proposes inclusion of trace distributions, notation $\sqsubseteq_{\mathrm{TD}}$, as an implementation relation between probabilistic automata that preserves safety properties. The following theorem states that probabilistic (hyper-)step refinements are a sound proof method for establishing trace distribution inclusion.

Theorem 1. *Let A and B be probabilistic automata with $ext_A = ext_B$.*

1. *If $A \sqsubseteq_{\mathrm{PSR}} B$ then $A \sqsubseteq_{\mathrm{PHSR}}$.*
2. *If $A \sqsubseteq_{\mathrm{PHSR}}$ then $A \sqsubseteq_{\mathrm{TD}} B$.*

Proof. For (1), suppose that $A \sqsubseteq_{\mathrm{PSR}} B$. Then there exists a probabilistic step refinement r from A to B. Let $R : states_A \to \Pi(states_B)$ be given by $R(s) = \{r(s) \mapsto 1\}$. It is routine to check that R is a probabilistic hyperstep refinement from A to B. Use that

$$R_{**}(\mu) = r_*(\mu),$$
$$s \xrightarrow{a}_B \nu \Leftrightarrow \{s \mapsto 1\} \xrightarrow{a}_{B^{**}} \nu.$$

Hence $A \sqsubseteq_{\mathrm{PHSR}}$.

For (2), suppose that $A \sqsubseteq_{\mathrm{PHSR}}$. Then there exists a probabilistic hyperstep refinement R from A to B. We claim that R is a probabilistic forward simulation in the sense of [18, 20]. Now $A \sqsubseteq_{\mathrm{TD}} B$ follows from the soundness result for probabilistic forward simulations, see Proposition 8.7.1 in [18]. For a simple, direct proof of (2) we refer to [22].

3.2 Probabilistic I/O Automata

This section defines the probabilistic I/O automaton model, an extension of probabilistic automata with a distinction between input and output actions, and with a notion of fair behavior.

Definition 12. *A* probabilistic I/O automaton *A is a probabilistic automaton enriched with*

1. *a partition of ext_A into* input actions *in_A and* output actions *out_A, and*
2. *a* task partition *$tasks_A$, which is an equivalence relation over $out_A \cup int_A$ with countably many equivalence classes.*

We require that A is input enabled, *which means that for all $s \in states_A$ and all $a \in in_A$, there is a μ such that $s \xrightarrow{a}_A \mu$.*

As probabilistic I/O automata are enriched probabilistic automata, we can use the notions of nonprobabilistic variant, reachable state, execution (fragment) and trace also for probabilistic I/O automata.

Definition 13. *Let A be a probabilistic I/O automaton. An execution of A is* fair *if the following conditions hold for each class C of $tasks_A$:*

1. *If α is finite then C is not enabled in the final states of α.*
2. *If α is infinite, then α contains either infinitely many actions from C or infinitely many occurrences of states in which no action in C is enabled.*

Similarly, a trace of A is fair *in A if it is the trace of a fair execution of A. The sets of fair executions and fair traces of A are denoted by $fexecs(A)$ and $ftraces(A)$ respectively.*

Definition 14. *Let A and B be probabilistic automata with $ext_A = ext_B$. Let r be a mapping from $states_A$ to $states_B$. Then r induces a relation $\tilde{r} \subseteq frags(A) \times frags(B)$ as follows: if $\alpha = s_0 a_1 s_1 \cdots \in frags(A)$, $\mathcal{I}$ is the index set of α, $\beta = t_0 b_1 t_1 \cdots \in frags(B)$ and $\mathcal{J}$ is the index set of β, then $\alpha \tilde{r} \beta$ if and only if there is a surjective, nondecreasing index mapping $m : \mathcal{I} \to \mathcal{J}$, such that for all $i \in \mathcal{I}$, $j \in \mathcal{J}$,*

1. *$m(0) = 0$*
2. *$r(s_i) = t_{m(i)}$*
3. *if $i > 0$ then either of the following conditions holds*
   (a) *$a_i = b_{m(i)} \wedge m(i) = m(i-1) + 1$ or*
   (b) *$a_i \in int_A \wedge b_{m(i)} \in int_B \wedge m(i) = m(i-1) + 1$ or*
   (c) *$a_i \in int_A \wedge m(i) = m(i-1)$.*

In [17], fair trace distribution inclusion, notation $\sqsubseteq_{\mathrm{FTD}}$, is proposed as an implementation relation between probabilistic I/O automata that preserves both safety and liveness properties.

Claim ([22]). Let A and B be probabilistic I/O automata. Let r be a probabilistic step refinement from A to B that relates each fair execution of A only to fair executions of B. Then $A \sqsubseteq_{\mathrm{FTD}} B$.

3.3 Timed Probabilistic I/O Automata

Definition 15. *A* timed probabilistic I/O automaton *A is a probabilistic automaton enriched with a partition of ext_A into* input actions *in_A,* output actions *out_A, and the set $\mathbb{R}^{>0}$ of positive real numbers or* time-passage actions. *We require[1] that, for all $s, s', s'' \in states_A$ and $d, d' \in \mathbb{R}^{>0}$ with $d' < d$,*

1. *A is input enabled,*
2. *each step labelled with a time-passage action leads to a Dirac distribution,*
3. *(Time determinism) if $s \xrightarrow{d}_A s'$ and $s \xrightarrow{d}_A s''$ then $s' = s''$.*
4. *(Wang's axiom) $s \xrightarrow{d}_A s'$ iff $\exists s'' : s \xrightarrow{d'}_A s''$ and $s'' \xrightarrow{d-d'}_A s'$.*

As timed probabilistic I/O automata are enriched probabilistic automata, we can use the notions of nonprobabilistic variant, reachable state, and execution (fragment), also for timed probabilistic I/O automata.

We say that an execution α of A is diverging *if the sum of the time-passage actions in α diverges to ∞.*

Definition 16. *Let A, B be probabilistic or timed probabilistic I/O automata. A function r is a probabilistic (hyper)step refinement from A to B if r is a probabilistic (hyper)step refinement from the underlying probabilistic automaton of A to the underlying probabilistic automaton of B.*

---

[1] For simplicity the conditions here are slightly more restrictive than those in [15].

In [22], it is argued that, under certain assumptions (met by the automata studied in this paper), $\sqsubseteq_{\mathrm{TD}}$ can be used as a safety and liveness preserving implementation relation between timed I/O automata. In addition, the relation $\sqsubseteq_{\mathrm{DFTD}}$ is proposed as a safety and liveness preserving implementation relation between timed probabilistic I/O automata and probabilistic I/O automata.

Claim ([22]). Let A be a timed probabilistic I/O automaton and let B be a probabilistic I/O automaton. Let r be a probabilistic step refinement from $\mathit{hide}(A, \mathbb{R}^{>0})$ to B that relates each divergent execution of A only to fair executions of B. Then $A \sqsubseteq_{\mathrm{DFTD}} B$.

4 Description of the Protocol

The IEEE 1394 serial bus protocol has been designed for communication between multimedia equipement. 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 protocol has several layers, of which the physical layer is the lowest. Within this layer a number of phases are identified. The protocol enters the so-called tree identify phase whenever a bus reset occurs, for instance when a connection is added or removed. The task of this phase is to check whether the network topology is a tree and, if so, to elect a leader among the nodes in this tree.

This is done by constructing a spanning tree in the network and electing the root of the tree as leader. 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. It is possible that at the end of the tree identify phase two nodes send parent request messages to each other; this situation is called root contention. In this paper we will be concerned with the formal verification and analysis of the root contention protocol which is run in this case. After completion of the root contention protocol, one of the two nodes has become root of the network.

Lynch [13, p501] describes an abstract version of the tree identify protocol and suggests to elect the node with the larger unique identifier (UID) as the root in case of root contention. Since during the tree identify phase no UID's are available (these will be assigned during a later phase of the physical layer protocol), a probabilistic algorithm has been chosen that is fully symmetric and does not require the presence of UID's.

Let us, for simplicity, refer to the two contending nodes as node 1 and node 2. The timed probabilistic I/O automata describing the behavior of these nodes are given in Figure 1, using the IOA syntax of [6] extended with a simple form

```
type P = enumeration of 1, 2
type M = enumeration of ⊥, req, ack
type Status = enumeration of unknown, root, child
type Toss = enumeration of head, tail
automaton Node(i: P)
  states
    status : Status := unknown,
    coin : Toss,
    snt : M := req,
    rec : M := req,
    x : Reals := 0
  signature
    input Receive(const i, m: M) where m ≠⊥
    output Send(const i, m: M) where m ≠⊥,
           Root(const i)
    internal Flip(const i),
             Child(const i)
    delay Time(d: Reals) where d > 0
  transitions
    internal Flip(i)
      pre status = unknown ∧ snt = req ∧ rec = req
      eff coin := { head 1/2
                  { tail 1/2 ;
          x := 0;
          snt :=⊥;
          rec :=⊥
    output Send(i, m)
      pre status = unknown ∧ snt =⊥
          ∧ x ≥ mindelay(coin)
          ∧ m = if rec =⊥ then req else ack
      eff snt := m
    input Receive(i, m)
      eff rec := m
    output Root(i)
      pre status = unknown ∧ snt = ack
      eff status := root
    internal Child(i)
      pre status = unknown ∧ rec = ack
      eff status := child
    delay Time(d)
      pre status = unknown ⇒
          (snt ≠ ack ∧ rec ≠ ack ∧ ¬(snt = req ∧ rec = req)
          ∧ snt =⊥⇒ x + d ≤ maxdelay(coin))
      eff x := x + d
```

Fig. 1. Node automaton.

of probabilistic choice. Roughly, the protocol works as follows. When a node `i` has detected root contention it first flips a coin (i.e., performs the action `Flip(i)`). If head comes up then it waits a short time, somewhere in the interval $[\delta_{\text{fast}}, \Delta_{\text{fast}}]$. If tail comes up then it waits a long time, somewhere in the interval $[\delta_{\text{slow}}, \Delta_{\text{slow}}]$. So $0 \leq \delta_{\text{fast}} \leq \Delta_{\text{fast}} < \delta_{\text{slow}} \leq \Delta_{\text{slow}}$. After the waiting period has elapsed, either no message from the contender has been received, or a parent request message has arrived. In the first case the node sends a request message to its contender (i.e., performs the action `Send(i,` *req*`)`), in the second case it sends an acknowledgement message (i.e., performs the action `Send(i,` *ack*`)`). As soon as a node has sent an acknowledgement it declares itself to be the root (via the action `Root(i)`), and whenever a node has received an acknowledgement it assumes that its contender will become root and it declares itself child (via the action `Child(i)`). If a node that has sent a request subsequently receives a request, then it concludes that there is root contention again, and the protocol starts all over again. The basic idea behind the protocol is that if the outcomes of the coin flips are different, the node with outcome tail (i.e., the slow one) will become root. And since with probability one the outcomes of the two coin flips will eventually be different, the root contention protocol will terminate (with probability one).

The timed probabilistic I/O automaton for node `i` (`i` $= 1, 2$), displayed in Figure 1, has five state variables: variable `status` tells whether the node has become *root*, *child*, or whether its status is still *unknown*; variable `coin` records the outcome of the coin flip; variable `snt` records the last value (if any) that has been sent to the contender and may take values *req*, *ack* or $\perp$; similarly `rec` records the last value that has been received (if any); variable `x`, finally, models the arbitration timer that records the time that has elapsed since root contention has been detected. We use two auxiliary functions $\mathsf{mindelay}$ and $\mathsf{maxdelay}$ from Toss to Reals given by, for $\mathsf{c} \in \mathsf{Toss}$,

$$\mathsf{mindelay}(\mathsf{c}) \triangleq \textbf{if } \mathsf{c} = \textit{head} \textbf{ then } \delta_{\text{fast}} \textbf{ else } \delta_{\text{slow}}$$
$$\mathsf{maxdelay}(\mathsf{c}) \triangleq \textbf{if } \mathsf{c} = \textit{head} \textbf{ then } \Delta_{\text{fast}} \textbf{ else } \Delta_{\text{slow}}$$

Now it should not be difficult to understand the precondition/effect style definitions in Figure 1, except maybe for the definition of the `Time(d)` transitions. This part states that time will not progress if the status of the node is unknown and (1) an acknowledgement has been sent, or (2) an acknowledgement has been received, or (3) a parent request has both been sent and received. In the first case the automaton will instantaneously perform a `Root(i)` action, in the second case it will perform a `Child(i)` action, and in the third case there is contention and the automaton will flip a coin.[2] The last clause in the precondition of `Time(d)` enforces that a `Send(i, m)` action is performed within either Δ_{fast} or Δ_{slow} time after the coin flip (depending on the outcome). Once the status of the automaton has become *root* or *child* there are no more restrictions on time passage.

[2] Note that in each of these three cases we abstract in our model from the computation time required to perform these actions.

The two automata for node 1 and node 2 communicate via wires, which are modeled as the timed probabilistic automata Wire(1, 2) and Wire(2, 1) specified in Figure 2. We assume an upper bound $\Gamma \geq 0$ on the communication delay.

```
automaton Wire(i: P, j: P)
    states
        msg : M :=⊥,
        x : Reals := 0
    signature
        input Send(const i, m: M) where m ≠⊥
        output Receive(const j, m: M) where m ≠⊥
        delay Time(d: Reals) where d > 0
    transitions
        input Send(i, m)
            eff msg := m;
                x := 0
        output Receive(j, m)
            pre m = msg
            eff msg :=⊥
        delay Time(d)
            pre msg ≠⊥⇒ x + d ≤ Γ
            eff x := x + d
```

Fig. 2. Wire automaton.

The full system can now be described as the parallel composition of the two node automata and the two wire automata, with all synchronization actions hidden (see Figure 3).

```
Impl ≜ hide Send(i, m), Receive(i, m) for i : P, m : M in
           compose Node(1); Wire(1, 2); Node(2); Wire(2, 1)
```

Fig. 3. The full system.

Remark 1. As Segala [18] points out in his thesis, it would be useful to study the theory of *receptiveness* [19] in the context of randomization. As far as we know, nobody has taken up this challenge yet. Intuitively, an automaton is receptive if it does not constrain its environment, for instance by not accepting certain inputs or by preventing time to pass beyond a certain point. Behavior inclusion is used as an implementation relation in the I/O automata framework and we exclude trivial implementations by requiring that an implementation is receptive.

If we replace all probabilistic choices by nondeterministic choices in the automata of this section, then the resulting timed I/O automata are receptive in the sense of [19]. Even with a more restrictive definition of receptivity, in which we allow the environment to resolve all probabilistic choices, the automata of this section remain receptive.

5 Verification and Analysis

Of course the key correctness property of the root contention protocol which we would like to prove is that eventually exactly one node is designated as root. This correctness property is described by the two state probabilistic I/O automaton `Spec` of Figure 4. We will establish that `Impl` implements `Spec`, provided the

> **automaton** `Spec`
>   **states**
>   `done` : Bool := *false*
>   **signature**
>   **output** `Root(i: P)`
>   **transitions**
>   **output** `Root(i)`
>   **pre** `done` = *false*
>   **eff** `done` = *true*
>   **tasks**
>   *One block*

Fig. 4. Specification.

following two constraints on the parameters are met:

$$\Gamma < \delta_{\text{fast}} \tag{1}$$

$$\Delta_{\text{fast}} + 2\Gamma < \delta_{\text{slow}} \tag{2}$$

Within our proof, we introduce three intermediate automata `I1`, `I2` and `I3`, and prove that

$$\texttt{Impl} \sqsubseteq_{\text{TD}} \texttt{I1} \sqsubseteq_{\text{TD}} \texttt{I2} \sqsubseteq_{\text{TD}} \texttt{I3} \sqsubseteq_{\text{TD}} \texttt{Spec}.$$

These results (or more precisely the refinements that are established in their proofs) are then used to obtain that

$$\texttt{Impl} \sqsubseteq_{\text{TD}} \texttt{I1} \sqsubseteq_{\text{DFTD}} \texttt{I2} \sqsubseteq_{\text{FTD}} \texttt{I3} \sqsubseteq_{\text{FTD}} \texttt{Spec}.$$

`I1` is a timed probabilistic I/O automaton, which abstracts from all the message passing in `Impl`, while preserving the probabilistic choices as well as most information about the timing of the Root(i) events. I2 is a probabilistic I/O automaton which is identical to I1, except that all real-time information has been omitted. In I3 the two coin flips from each node of the protocol are combined into a single probabilistic transition.

5.1 Invariants

We will show that there exists a probabilistic step refinement from Impl to an intermediate automaton I1. In order to establish a refinement, we first need to introduce a number of invariants for automaton Impl.

We use subscripts 1 and 2 to refer to the state variables of Node(1) and Node(2), respectively, and subscripts 12 and 21 to refer to the state variables of Wire(1, 2) and Wire(2, 1), respectively. So, x_1 denotes the clock variable of Node(1), x_{12} the clock variable of Wire(1, 2), etc. Within formulas we further use the following abbreviations, for $i \in P$,

$$\begin{aligned}
\mathsf{Cont}(i) &\triangleq \mathtt{snt}_i = \mathit{req} \wedge (\mathtt{rec}_i = \mathit{req} \vee \mathtt{msg}_{ji} = \mathit{req}) \\
\mathsf{Wait}(i) &\triangleq \mathtt{snt}_i = \mathtt{rec}_i = \perp \\
\delta_i &\triangleq \mathsf{mindelay}(\mathtt{coin}_i) \\
\Delta_i &\triangleq \mathsf{maxdelay}(\mathtt{coin}_i)
\end{aligned}$$

Predicate $\mathsf{Cont}(i)$ states that node i has either detected contention (a request has both been sent and received) or will do so in the near future (the node has sent a request and will receive one soon). Predicate $\mathsf{Wait}(i)$ states that node has flipped the coin and is waiting for the delay time to expire; no message has been received yet. State function δ_i gives the minimum delay time for node i, and state function Δ_i the maximum delay time (both state functions depend on the outcome of the coin flip).

We claim that assertions (3)-(19) below are invariants of automaton Impl.

$$x_i \geq 0 \tag{3}$$
$$\mathtt{status}_i = \mathit{unknown} \wedge \mathtt{snt}_i \neq \mathit{req} \Rightarrow x_i \leq \Delta_i \tag{4}$$
$$\mathtt{snt}_i = \mathit{ack} \Rightarrow x_i \geq \delta_i \tag{5}$$
$$\mathtt{status}_i = \mathit{root} \Rightarrow \mathtt{snt}_i = \mathit{ack} \tag{6}$$
$$\mathtt{status}_i = \mathit{child} \Rightarrow \mathtt{rec}_i = \mathit{ack} \tag{7}$$
$$x_{ij} \geq 0 \tag{8}$$
$$\mathtt{msg}_{ij} \neq \perp \Rightarrow x_{ij} \leq \Gamma \tag{9}$$
$$\mathsf{Cont}(i) \Leftrightarrow \mathsf{Cont}(j) \Rightarrow | x_i - x_j | \leq \Gamma \tag{10}$$
$$\mathsf{Cont}(i) \wedge \neg\mathsf{Cont}(j) \Rightarrow \mathsf{Wait}(j) \wedge \mathtt{msg}_{ij} = \perp \wedge x_j \leq \Gamma \tag{11}$$
$$\mathtt{msg}_{ij} \neq \perp \Rightarrow \mathtt{rec}_j = \perp \tag{12}$$
$$\mathtt{msg}_{ij} = \perp \Rightarrow \mathtt{snt}_i = \perp \vee \mathtt{rec}_j \neq \perp \vee \mathsf{Cont}(i) \tag{13}$$

$$\mathtt{msg_{ij}} = \mathit{req} \wedge \neg\mathsf{Wait}(\mathtt{i}) \Rightarrow \mathtt{snt_i} = \mathit{req} \wedge \mathtt{snt_j} \neq \mathit{ack} \wedge$$
$$\delta_{\mathtt{i}} \leq \mathtt{x_i} - \mathtt{x_{ij}} \leq \Delta_{\mathtt{i}} \quad (14)$$
$$\mathtt{msg_{ij}} = \mathit{req} \wedge \mathsf{Wait}(\mathtt{i}) \Rightarrow \mathtt{snt_j} = \mathit{req} \wedge \mathtt{x_i} \leq \mathtt{x_{ij}} \quad (15)$$
$$\mathtt{snt_i} = \perp \wedge \mathtt{rec_i} = \mathit{req} \Rightarrow \mathtt{snt_j} = \mathit{req} \wedge \mathtt{rec_j} = \perp \wedge \mathtt{x_j} \geq \delta_{\mathtt{j}} \quad (16)$$
$$\mathtt{rec_i} = \mathit{ack} \Rightarrow \mathtt{snt_j} = \mathit{ack} \quad (17)$$
$$\mathtt{msg_{ij}} = \mathit{ack} \Rightarrow \mathtt{snt_i} = \mathit{ack} \quad (18)$$
$$\mathtt{snt_i} = \mathit{ack} \Rightarrow \mathtt{rec_i} = \mathtt{snt_j} = \mathit{req} \wedge \mathtt{rec_j} \neq \mathit{req} \wedge \mathtt{x_j} \geq \delta_{\mathtt{j}} \quad (19)$$

Assertions (3)-(9) are local invariants, which can be proven straightforwardly for automata `Node(i)` and `Wire(i, j)` in isolation. Most of the time nodes 1 and 2 are either both in contention or both not in contention. Assertion (10) states that in these cases the values of the clocks of the two nodes differ by at most Γ. Assertion (11) expresses that the only case where node `i` is in contention but the other node `j` is not occurs when `j` has just flipped a coin but the request message that `j` sent to `i` has not yet arrived or been processed. If a channel contains a message then nothing has been received at the end of this channel (12). If the channel from `i` to `j` is empty then either no message has been sent into the channel at `i`, or a message has been received at `j`, or we have a situation where `i` is in contention and `j` has just flipped a coin and moved to a new phase (13). If the channel from `i` to `j` contains a request message then there are two possible cases. Either `i` has sent the message and is waiting for a reply (14), or there is contention and `i` has just flipped a coin (15). If `i` has received a request message without having sent anything, then `j` has sent this message but has not received anything (16). The last three invariants deal with situations where there is an acknowledgement somewhere in the system (17)-(19). In these cases the global state is almost completely determined: if an acknowledgement is in a channel or has been received then it has been sent, and if a node has sent an acknowledgement then it has received a request, which in turn has been sent by the other node.

The proofs of the following two lemmas are tedious but completely standard since they only refer to the non-probabilistic automaton $\mathtt{Impl}^-$. Detailled proofs can be obtained via URL `http://www.cs.kun.nl/~fvaan/PAPERS/SVproofs`.

Lemma 1. *Suppose state s satisfies assertions (3)-(19) and $s \stackrel{\mathtt{Send(i,\,m)}}{\rightarrow} s'$. Then $s \models \mathtt{msg_{ij}} = \mathtt{rec_j} = \perp$ and $s' \models \mathsf{Cont}(\mathtt{i}) \Leftrightarrow \mathsf{Cont}(\mathtt{j})$.*

Lemma 2. *Assertions (3)-(19) hold for all reachable states of* `Impl`.

Remark 2. The first constraint on the timing parameters ($\Gamma < \delta_{\mathrm{fast}}$) is used in the proof of Lemma 1 and ensures that there can never be two messages travelling in a wire at the same time. This property allows for a very simple model of the wires, in which a new message overwrites an old message. The constraint is not needed to prove the correctness of the algorithm. Nevertheless, since the constraint is implied by the standard, we decided to include it as an assumption in our analysis.

5.2 The First Intermediate Automaton

Intermediate automaton `I1` is displayed in Figure 5. This probabilistic timed I/O

```
automaton I1
  type Phase = enumeration of init, head, tail, done
  states
    phase : Array[P, Phase] := constant(init),
    x : Reals := 0
  signature
    output Root(i: P)
    internal Flip(i: P),
             Retry(c: Toss)
    delay Time(d: Reals)  where d > 0
  transitions
    internal Flip(i)
      pre phase[i] = init
      eff phase[i] := { head 1/2
                      { tail 1/2  ;
          if phase[next(i)] ≠ init then x := 0
    output Root(i)
      pre {phase[1], phase[2]} ⊆ {head, tail}
          ∧ ¬(phase[i] = head ∧ phase[next(i)] = tail)
          ∧ x ≥ mindelay(phase[i]) − Γ
      eff phase := constant(done)
    internal Retry(c)
      pre phase = constant(c)
          ∧ x ≥ mindelay(c)
      eff phase := constant(init);
          x := 0
    delay Time(d)
      pre  init ∈ {phase[1], phase[2]}  ⇒  x + d ≤ Γ
          ∧ {phase[1], phase[2]} ⊆ {head, tail}  ⇒
              x + d ≤ max(maxdelay(phase[1]), maxdelay(phase[2]))
      eff x := x + d
```

Fig. 5. Intermediate automaton `I1`.

automaton records the status for each of the two nodes to be either *init*, *head*, *tail*, or *done*. In addition `I1` maintains a clock `x` to impose timing constraints between events. Apart from the delay action there are three actions: `Flip(i)`, which corresponds to node `i` flipping a coin, `Root(i)`, which corresponds to node `i` declaring itself to be the root, and `Retry(c)`, which models the restart of the protocol in the case where the outcome of both coin flips is `c`. Node `i` performs a (probabilistic) `Flip(i)` action in its initial state. A `Root(i)` transition may occur if both nodes have flipped a coin and it is *not* the case that the outcome for $\mathtt{i}$ is *head* and for $\mathtt{j}$ *tail*. A $\mathtt{Retry(c)}$ transition may occur if both nodes have flipped $\mathtt{c}$. Clock $\mathtt{x}$ is used to express that both nodes flip their coin within time Γ after the (re-)start of the protocol. In addition it ensures that subsequently (depending on the outcome of the coin flips) at least $\delta_{\text{fast}} - \Gamma$ or $\delta_{\text{slow}} - \Gamma$ time and at most Δ_{fast} or Δ_{slow} time will elapse before either a $\mathtt{Root(i)}$ or a $\mathtt{Retry(c)}$ action occurs.

Proposition 1. $\mathtt{Impl} \sqsubseteq_{\text{TD}} \mathtt{I1}$. *More specifically the conjunction, for* $\mathtt{i} \in \mathsf{P}$, *of*

$$\begin{aligned} \mathtt{phase[i]} &= \textbf{if } \mathtt{status}_1 = \textit{root} \vee \mathtt{status}_2 = \textit{root} \textbf{ then } \textit{done} \textbf{ else} \\ &\qquad \textbf{if } \mathsf{Cont}(\mathtt{i}) \textbf{ then } \textit{init} \textbf{ else } \mathtt{coin}_\mathtt{i} \textbf{ fi fi} \\ \mathtt{x} &= \textbf{if } \mathsf{Cont}(1) \vee \mathsf{Cont}(2) \textbf{ then } \min(\mathtt{x}_{12}, \mathtt{x}_{21}) \textbf{ else } \min(\mathtt{x}_1, \mathtt{x}_2) \end{aligned}$$

determines a probabilistic step refinement from $\mathtt{Impl}$ *to* $\mathtt{I1}$.

Proof. Routine. See `http://www.cs.kun.nl/~fvaan/PAPERS/SVproofs`.

Remark 3. The second constraint on the timing parameters ($\Delta_{\text{fast}} + 2\Gamma < \delta_{\text{slow}}$) is used in the proof of Proposition 1 and ensures that contention may only occur if the outcomes of both coin flips are the same. This property is needed to prove termination of the algorithm (with probability 1).

Remark 4. Figure 6 gives the values for some of the relevant parameters of the protocol as listed in the standard IEEE 1394 [9] and in the more recent draft standard IEEE 1394a [10]. Interestingly, the values in two documents are different. Given our timing constraints (1) and (2), this leads to a maximum value for

Timing constant	Min (1394)	Max (1394)	Min (1394a)	Max (1394a)
ROOT_CONTENT_FAST	$0.24\mu s$	$0.26\mu s$	$0.76\mu s$	$0.80\mu s$
ROOT_CONTENT_SLOW	$0.57\mu s$	$0.60\mu s$	$1.60\mu s$	$1.64\mu s$

Fig. 6. Timing parameters.

Γ of $\frac{0.57-0.26}{2}\mu s = 0.155\mu s$ for IEEE 1394, and $\frac{1.60-0.8}{2}\mu s = 0.4\mu s$ for the draft IEEE 1394a. With the maximal signal velocity of $5.05 ns/meter$ that is specified in both documents, this gives a maximum cable length of appr. 31 meter for IEEE 1394 and 79 meter for IEEE 1394a. However, these values should be viewed as upper bounds since within our model we have not taken into account the processing times of signals. IEEE 1394 specifies a maximum cable length of 4.5 meter.

Remark 5. In [16] it is claimed that if both nodes happen to select slow timing or if both nodes select fast timing, contention results again. This is incorrect. In automaton $\mathtt{I1}$ each of the two nodes may become root if both nodes happen to select the same timing delay. This may also occur within a real-world implementation of the protocol: if in the implementation the timing parameters of one node are close to their minimum values, in the other node close to their maximum values, and if the communication delay is small, then it may occur that a message of node `i` arrives at node `j` before the timing delay of node `j` has expired. In fact, by instantiating the timing parameters differently in different devices (for instance via some random mechanism!) one may reduce the expected time to resolve contention. Unfortunately, a more detailed analysis of this phenomenon falls outside the scope of this paper.

Remark 6. Another way in which the performance of the protocol could be improved is by repeatedly polling the input during the timing delay, rather than checking it only at the end. We suggest that, if the process receives a request when the timing delay has not yet expired, then it immediately sends an acknowledgement (and declares itself root). If the process has not received a request during the timing delay, then it sends a request and proceeds as the current implementation. In a situation where node i flips head and selects a timing delay of δ_{fast} and the other node j flips tail and selects a timing delay of Δ_{slow}, our version elects a leader within at most $\delta_{\text{fast}} + 3\Gamma$, whereas in the current version this upperbound is $\Delta_{\text{slow}} + 3\Gamma$.

automaton `I2`
    states
        $\mathtt{phase} : \mathsf{Array}[\mathsf{P}, \mathsf{Phase}] := \mathsf{constant}(\mathit{init})$
    signature
        output `Root(i: P)`
        internal `Flip(i: P)`,
                `Retry(c: Toss)`
    transitions
        internal `Flip(i)`
            pre $\mathtt{phase}[\mathtt{i}] = \mathit{init}$
            eff $\mathtt{phase}[\mathtt{i}] := \begin{cases} \mathit{head} & \frac{1}{2} \\ \mathit{tail} & \frac{1}{2} \end{cases}$
        output `Root(i)`
            pre $\{\mathtt{phase}[1], \mathtt{phase}[2]\} \subseteq \{\mathit{head}, \mathit{tail}\}$
                $\wedge \neg(\mathtt{phase}[\mathtt{i}] = \mathit{head} \wedge \mathtt{phase}[\mathtt{next(i)}] = \mathit{tail})$
            eff $\mathtt{phase} := \mathsf{constant}(\mathit{done})$
        internal `Retry(c)`
            pre $\mathtt{phase} = \mathsf{constant}(\mathtt{c})$
            eff $\mathtt{phase} := \mathsf{constant}(\mathit{init})$
    tasks
        One block

Fig. 7. Intermediate automaton `I2`.

```
automaton I3
    type Loc = enumeration of init, win1, win2, same, done
    states
        loc : Loc := init
    signature
        output Root(i: P)
        internal Flips,
                 Retry
    transitions
        internal Flips
            pre loc = init
            eff loc := { win1  1/4
                       { win2  1/4
                       { same  1/2
        output Root(i)
            pre loc ∈ {win_i, same}
            eff loc := done
        internal Retry
            pre loc = same
            eff loc := init
    tasks
        One block
```

Fig. 8. Intermediate automaton I3.

5.3 The Second Intermediate Automaton

In Figure 7 the second intermediate automaton I2 is described. I2 is a probabilistic I/O automaton that is identical to I1 except that all real-time information has been abstracted away; instead a (trivial) task partition is included. The proof of the following Proposition 2 is easy: the projection function π from I1 to I2 trivially is a probabilistic step refinement (after hiding of the time delays).

Proposition 2. I1 $\sqsubseteq_{\mathrm{TD}}$ I2.

Proposition 3. *If $\alpha \in execs(\mathtt{I1})$ is diverging π relates α and β, then β is fair.*

The result formulated in the Proposition 3 above follows by the fact that a diverging execution of I1 either contains infinitely many Retry actions, or contains an infinite suffix with a Root(i) transition followed by an infinite number of delay transitions. Now the claim at the end of Section 3.3 implies I1 $\sqsubseteq_{\mathrm{DFTF}}$ I2.

5.4 The Third Intermediate Automaton

Figure 8 gives the IOA code for the probabilistic I/O automaton I3. This automaton abstracts from I2 since it only has a single probabilistic transition. Within automaton I3, *init* is the initial state and *done* is the final state in which a root has been elected. The remaining states win_1, win_2, *same* correspond to situations in which both processes have flipped but no leader has been elected yet. The value $win_{\mathtt{i}}$ indicates that the results are different and the outcome of `i` equals tail. In state *same* both coin flips have yielded the same result.

Proposition 4. `I2` $\sqsubseteq_{\mathrm{TD}}$ `I3`. *More specifically, the following function r from (reachable) states of* `I2` *to discrete probability spaces over states of* `I3` *is a probabilistic hyper step refinement from* `I2` *to* `I3` *(we represent a state with a list containing the values of its variables):*

$$\begin{aligned}
r(init, init) &= \{init \mapsto 1\}\\
r(head, init) &= \{win_2 \mapsto \tfrac{1}{2}, same \mapsto \tfrac{1}{2}\}\\
r(init, head) &= \{win_1 \mapsto \tfrac{1}{2}, same \mapsto \tfrac{1}{2}\}\\
r(tail, init) &= \{win_1 \mapsto \tfrac{1}{2}, same \mapsto \tfrac{1}{2}\}\\
r(init, tail) &= \{win_2 \mapsto \tfrac{1}{2}, same \mapsto \tfrac{1}{2}\}\\
r(head, head) &= \{same \mapsto 1\}\\
r(tail, tail) &= \{same \mapsto 1\}\\
r(head, tail) &= \{win_2 \mapsto 1\}\\
r(tail, head) &= \{win_1 \mapsto 1\}\\
r(done, done) &= \{done \mapsto 1\}
\end{aligned}$$

The proofs of the following Propositions 5 and 6 can be found in [21]. These proofs are the only places in our verification where nontrivial probabilistic reasoning takes place: establishing $\sqsubseteq_{\mathrm{FTD}}$ basically amounts to proving that the probabilistic mechanism in the protocol ensures termination with probability 1. Note that the automata involved are all very simple: `I2` has 10 states, `I3` has 5 states, and `Spec` has 2 states.

Proposition 5. `I2` $\sqsubseteq_{\mathrm{FTD}}$ `I3`.

Proposition 6.

1. `I3` $\sqsubseteq_{\mathrm{TD}}$ `Spec`. *More specifically, the function determined by the predicate* $done \Leftrightarrow \mathtt{loc} = 4$ *is a probabilistic step refinement from* `I3` *to* `Spec`.
2. `I3` $\sqsubseteq_{\mathrm{FTD}}$ `Spec`.

6 Concluding Remarks

In order to make our verification easier to understand, we introduced three auxiliary automata in between the implementation and the specification automaton. We also used the simpler notion of probabilistic (hyper)step refinement rather than the more general but also complex simulation relations (especially in the timed case!) which have been proposed by Segala and Lynch [18, 20]. The complexity of the definitions in [18, 20] is mainly due to the fact that a single step in one machine can in general be simulated by a sequence of steps in the other machine with the same external behavior. In the probabilistic case this means that a probabilistic transition in one machine can be simulated by a tree like structure in the other machine. In the simulations that we use in this paper, a single transition in one machine is simulated by at most one transition in the other machine. In our case study we were able to carry out the correctness proof by using only probabilistic (hyper)step refinements. However, it is easy to come up with counterexamples which show that this is not possible in general. Griffioen and Vaandrager [7] introduce various notions of *normed simulations* and prove that these notions together constitute a complete proof method for establishing trace inclusion between (nonprobabilistic, untimed automata). In normed simulations a single step in one machine is always simulated by at most one step in the other machine. We think that it is possible to come up with a complete method for proving trace distribution inclusion between probabilistic automata by defining probabilistic versions of the normed simulations of [7].

For timed automata, trace inclusion is in general not an appropriate implementation relation. In [15] the coarser notion of *timed* trace inclusion is advocated instead. Similarly, [18] suggests the notion of timed trace distribution inclusion as an implementation relation between probabilistic timed automata. Since trace distribution inclusion implies timed trace distribution inclusion, and the two preorders coincide for most practical cases, we prefer to use the much simpler proof techniques for trace distribution inclusion.

The idea to introduce auxiliary automata in a simulation proof has been studied in many papers, see for instance [1]. The verification reported in this paper indicates that the introduction of auxiliary automata can be very useful in the probabilistic case: it allowed us to first deal with the nonprobabilistic and real-time behavior of the protocol, basically without being bothered by the complications of randomization; nontrivial probabilistic analysis was only required for automata with 10 states or less.

As a final remark we would like to point out that the root contention protocol which we discussed in this paper is essentially finite state. It is therefore an interesting challenge for tool builders to analyze this protocol fully automatically. Most of the verification effort in our case study was not concerned with randomization at all, but just consisted of standard invariant proofs. In fact, one could use existing tools for the analysis of timed automata such as UPPAAL [3], KRONOS [4] and HyTech [8] to check these invariants. It would be especially interesting to derive the constraints on the timing parameters fully automatically (at the moment only HyTech [8] can do parametric analysis). Tool support will be essential for the analysis of more detailed models of the protocol in which also computation delays have been taken into account.

Acknowledgement

We thank Judi Romijn for her explanation of some subtle points in IEEE 1394, and for her constructive criticism on early versions of our I/O automata model.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.
2. R. Alur, T.A. Henzinger, and E.D. Sontag, editors. *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
3. J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and Wang Yi. UPPAAL: a tool suite for the automatic verification of real-time systems. In Alur et al. [2], pages 232–243.
4. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Alur et al. [2], pages 208–219.
5. M.C.A. Devillers, W.O.D. Griffioen, J.M.T Romijn, and F.W. Vaandrager. Verification of a leader election protocol — formal methods applied to IEEE 1394. Technical Report CSI-R9728, Computing Science Institute, University of Nijmegen, December 1997. Submitted.
6. S.J. Garland, N.A. Lynch, and M. Vaziri. IOA: A language for specifiying, programming, and validating distributed systems, September 1997. Available through URL `http://larch.lcs.mit.edu:8001/~garland/ioaLanguage.html`.
7. W.O.D. Griffioen and F.W. Vaandrager. Normed simulations. In A.J. Hu and M.Y. Vardi, editors, *Proceedings of the 8th 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.
8. T.A. Henzinger and P.-H. Ho. HyTech: The Cornell HYbrid TECHnology Tool. In U.H. Engberg, K.G. Larsen, and A. Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Aarhus, Denmark, volume NS-95-2 of *BRICS Notes Series*, pages 29–43. Department of Computer Science, University of Aarhus, May 1995.
9. IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.
10. IEEE Computer Society. P1394a Draft Standard for a High Performance Serial Bus (Supplement). Draft 2.0, March 1998.
11. 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.
12. 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/`.
13. N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, Inc., San Fransisco, California, 1996.
14. N.A. Lynch, I. Saias, and R. Segala. Proving time bounds for randomized distributed algorithms. In *Proceedings of the 13th Annual ACM Symposium on the Principles of Distributed Computing*, pages 314–323, Los Angeles, CA, August 1994.
15. N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, II: Timing-based systems. *Information and Computation*, 128(1):1–25, July 1996.
16. MindShare, Inc, and D. Anderson. *FireWire System Architecture: IEEE 1394*. Addison Wesley, 1998.

17. A. Pogosyants, R. Segala, and N.A. Lynch. Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. In M. Mavronicolas and Ph. Tsigas, editors, *Proceedings of 11th International Workshop on Distributed Algorithms (WDAG'97),* Saarbrucken, Germany, September 1997, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997. Also, Technical Memo MIT/LCS/TM-555, Laboratory for Computer Science, Massachusetts Institute of Technology.
18. R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1995. Available as Technical Report MIT/LCS/TR-676.
19. R. Segala, R. Gawlick, J.F. Søgaard-Andersen, and N.A. Lynch. Liveness in timed and untimed systems. *Information and Computation*, 141(2):119–171, March 1998.
20. R. Segala and N.A. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.
21. M.I.A. Stoelinga. Gambling for leadership: Root contention in IEEE 1394. Technical Report CSI-R9904, Computing Science Institute, University of Nijmegen, 1999.
22. M.I.A. Stoelinga and F.W. Vaandrager. Gambling together in Monte Carlo: Step refinements for probabilistic automata. Technical Report CSI-R99xx, Computing Science Institute, University of Nijmegen, 1999. To appear.