CHAPTER 7

A Case Study: the IEEE 1394 Root Contention Protocol

1394's for the fun stuff...
Phil Roden, 1394 Digital Design Manager, Texas Instruments

Then he is to take the two goats and present them before the Lord at the entrance to the Tent of Meeting. He is to cast lots for the two goats — one lot for the Lord and the other for the scapegoat. Aaron shall bring the goat whose lot falls to the Lord and sacrifice it for a sin offering.
Leviticus 16:7-9

"Teach me, and I will be quiet; show me where I have been wrong. How painful are honest words! But what do your arguments prove? Do you mean to correct what I say, and treat the words of a despairing man as wind? You would even cast lots for the fatherless and barter away your friend. "
Job 6:24-27

Abstract This chapter presents a formal specification and analysis of the Root Contention Protocol from the physical layer of the IEEE 1394 ("FireWire", "iLink") standard. In our protocol models, randomization, real-time and timing constraints play an essential role. We analyze the protocol with three different methods: We first give a manual verification in the probabilistic I/O automata model of Segala and Lynch, where the emphasis is on the probabilistic and real–time behavior. Then, we provide a mechanical verification with the model checker Uppaal, where we are particularly interested in the timing constraints. And, finally, we present a fully parametric analysis using parametric model checking.

7.1 Introduction

The Root Contention Protocol (RCP) is an industrial leader election protocol, which uses randomization and timing delays as essential ingredients to determine a leader among two processes. Moreover, constraints on the timing parameters are crucial for correct protocol operation. All this makes RCP a perfect case study to investigate the applicability of the theory and analysis techniques presented in this thesis.

RCP is part of the IEEE 1394 standard. This standard defines a serial bus that allows several multimedia devices, such as TVs, PCs and VCRs, to be connected in a network and to communicate with each other at high speed. IEEE 1394 is currently one of the standard protocols for interconnecting multimedia equipment and, today, one can buy PCs, laptops and digital cameras with an IEEE 1394 serial bus port.

Various parts of IEEE have been specified and/or verified formally, see for instance [DGRV00, KHR97, Lut97]. Root contention has become a popular case study in Formal Methods. It has been analyzed in [Sha99, SV99a, BLdRT00, BST00, CS01, D'A99, KNS01, FS01, Sto99b, SV99b, SS01, HRSV01]. Section 7.3 of this chapter presents a comparative study of the verification methods applied in these works. After this section, we shift to the verification work carried out by ourselves and present three different approaches to the verification of RCP.

First, we provide a manual verification of RCP in the probabilistic automaton model of Segala & Lynch. This verification is based on [SV99b]. The emphasis is on probabilistic and real–time behavior, but we also derive constraints on the timing parameters which guarantee protocol correctness. To make the verification easier to understand, we introduce several intermediate automata in between the implementation and the specification automaton. After publication of [SV99b], we discovered that the model presented here does not fully comply to the IEEE 1394 standard. More precisely, the communication between the processes was not modeled completely appropriately. However, we present the verification as it is in [SV99b], because the very same techniques would apply to a correct protocol model and redoing the proofs is tedious and scientifically not interesting.

Moreover, the second analysis in this chapter improves the protocol model and carries out a mechanical verification. We follow the same approach as in the first verification and base the analysis on similar intermediate automata. However, we disregard the probabilistic aspects of the protocol in this verification, since they are the same as before. Instead, we concentrate on the functional and timing behavior, which is where both models differ. We use the timed model checker Uppaal to analyze the protocol's timing constraints. Since the standard version Uppaal is not able to do parameter analysis and the parametric version was not available when we carried out this research, we establish the results experimentally. By checking a large number of protocol instances, we derive two constraints that are necessary and sufficient for correct protocol operation. Although not completely formal, this is fast and easy and we believe that the results are trustworthy. Due to the differences in the communication models, the constraints derived here differ from the ones found in the first analysis.

Thirdly, we briefly report on a fully parametric verification of RCP. For this, we use the prototype parametric model checker described in Chapter 6. We formulate the protocol correctness in Uppaal's logic and analyze several automaton models from the previous verifications. Here, we profit from the fact that the parameter constraints have already been found. So, rather than having the tool generating the constraints, which is often a time and memory consuming task, we provide the constraints found before as initial constraints to the tool. The results here coincide with the ones from the previous two verifications.

Organization of the chapter

We start with an informal description of the protocol in Section 7.2. Then Section 7.3 presents an overview of several papers in the literature analyzing RCP. Sections 7.4, 7.5 and 7.6 present respectively a manual, a mechanical and a parametric verification of RCP. Each of these three sections comes with their own concluding remarks, whereas Section 7.7 presents some overall conclusions.

7.2 Root Contention within IEEE 1394

This section explains the IEEE 1394 root contention protocol informally. We start with some background information on the IEEE 1394 standard. In particular, we zoom in on the physical layer and the tree identify phase within this layer, which is where RCP is located. Finally, we describe RCP itself and its timing constraints.

7.2.1 The IEEE 1394 standard

The IEEE 1394 standard [IEE00a], which is also known under the popular names of FireWire and iLink, specifies a high performance serial bus. It has been designed for interconnecting computer and consumer equipment, such as VCRs, PCs and digital cameras. The bus supports fast and cheap, peer–to–peer data transfer among up to 64 devices, both asynchronous and isochronous. The bus is hot plug–and–play, which means that devices can be added or removed at any time.

Although originally developed by Apple (FireWire), the version documented in [IEE96] has been accepted as a standard by IEEE in 1996. More than seventy companies — including Sun, Microsoft, Lucent Technologies, Philips, IBM, and Adaptec — have joined in the development of the IEEE 1394 bus and related consumer electronics and software. Currently, IEEE 1394 is one of the standard protocols for connecting digital multimedia equipment. The IEEE 1394a supplement [IEE00a] to the standard is the latest approved standard that concerns root contention and includes several clarifications, extensions, and performance improvements over earlier standards. Ongoing standardization developments are taking place in several IEEE working groups, resulting in several standard proposals. For instance, P1394b ("gigabyte 1394") [IEE01b] develops a faster version of 1394. The proposal P1394.1 [IEE01a] concerns so–called bus bridges for interconnecting several IEEE 1394 networks and P1394.3 [IEE00b] proposes a peer–to–peer data transport protocol. This chapter concerns the IEEE 1394a standard, unless clearly stated otherwise.

The IEEE 1394 standard (and its successors) provides a layered, OSI–style description of the protocol. The presentation is rather informal. It uses text, diagrams and program code to describe the protocol operation.

The standard refers to the devices in a 1394 network as *nodes*, each having one or more *ports*. A port may be connected to one other node's port, via a bi–directional cable. Nodes should be connected in a tree–like network topology, that is, without any cycles. There are four protocol layers: the *serial bus management layer*, the *transaction layer*, the *link layer* and the *physical layer*. RCP is part of the so–called *tree identify phase*, present in the lowest, physical layer. This layer provides the electrical and mechanical interface for data transmission across the bus. Furthermore, it handles bus configuration, arbitration and data transmission.

The physical layer starts with bus configuration. This is done automatically upon a bus reset: after power up and after device addition or removal. Bus configuration proceeds in three phases. First, bus initialization is performed. This is followed by the tree identify phase. The purpose of this phase is to identify a leader (root) node and the topology of all attached nodes. The root will act as bus master in subsequent phases of the protocol. Finally, in the self identify phase, each node selects a unique physical ID and identifies itself to the other nodes. When bus configuration has been completed, nodes can arbitrate for access to the bus and transfer data to any other node.

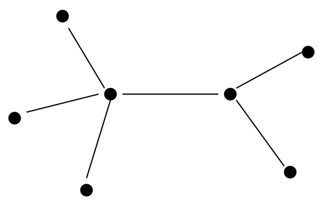

Figure 7.1: Initial network topology

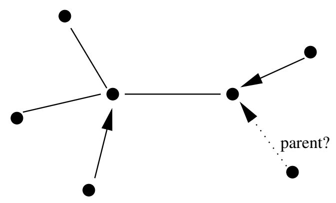

Figure 7.2: Intermediate configuration: leaf nodes send parent requests

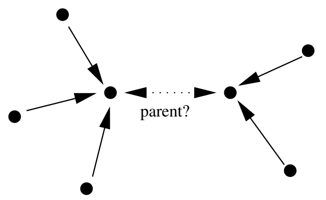

Figure 7.3: Two contending nodes

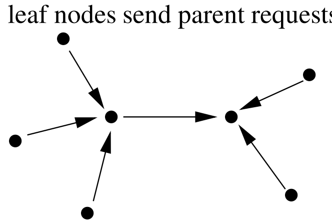

Figure 7.4: Final spanning tree: root contention has been resolved

The tree identify phase operates in the following way. First, it is checked whether the network topology is indeed a tree. If so, a spanning tree is constructed over the network and the root of this tree is elected as leader in the network.

The spanning tree is built as follows. As a basic operation, each node can drive a PARENT_NOTIFY (PN) or a CHILD_NOTIFY (CN) signal to a neighboring node. The node can also leave the line undriven (IDLE). The meaning of a PN signal is to ask the receiving node to become parent (connecting closer to the root) of the sending node. The sending node will then become its child and connects further away from the root. This signal is therefore also called a *parent request (req)*. A PN is acknowledged by a CN signal, also called *acknowledgement (ack)*. The receipt of a CN signal on a port is, in its turn, acknowledged by removing the PN signal from the connecting cable. A node only sends a PN signal via a port after it has received a PN signal on all its other ports. Thus, initially, only the leaf nodes, having only one port, send out a PN signal. If a node has received PN signals on all of its ports, then it has only child ports and it knows that is has been elected as the root of the tree. In the final stage of tree identify phase, two neighboring nodes may each try to find their parent by sending a PN signal to each other. This situation is called *root contention* and when it arises, the Root Contention Protocol is initiated to elect one of the two nodes as the root of the tree. Figures 7.1 – 7.4 (which have been borrowed from [DGRV00]) show a few snapshots of the tree identify phase. The arrows connect child ports to parent ports.

Lynch [Lyn96, 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 no UIDs are available during the tree identify phase (these will be assigned during a later phase of the physical layer protocol), the IEEE standard has chosen a probabilistic algorithm that is fully symmetric and does not require the presence of UIDs. This algorithm is described in the following section.

Timing constant		Min	Max
ROOT_CONTEND_FAST	1394	240 ns	260 ns
	1394a	760 ns	850 ns
	1394b	1600 ns	1610 ns
ROOT_CONTEND_SLOW	1394	570 ns	600 ns
	1394a	1590 ns	1670 ns
	1394b	3220 ns	3332 ns

Figure 7.5: Root contend wait times from IEEE 1394, 1394a and 1394b

7.2.2 The Root Contention Protocol

If a node receives a PN signal on a port, while sending a PN signal on that port, it knows it is in root contention, see Figure 7.3. Note that root contention is detected by each of the two contending nodes individually. Upon detection of root contention, a node backs off by removing the PN signal from the connecting cable, leaving the line in the state IDLE. At the same time, it starts a timer and picks a random bit. If the random bit is one, then the node waits for a time ROOT_CONTEND_SLOW (abbreviated $\mathrm{RC\_SLOW}$). If the random bit is zero, it waits for a shorter time ROOT_CONTEND_FAST (abbreviated $\mathrm{RC\_FAST}$). Table 7.5 lists the wait times as specified in the IEEE 1394, 1394a standards and P1394b proposal [IEE00a]. It is quite surprising that these values differ in the various standards and standard proposals, since these values influence the maximal values for several other protocol constants, see the discussion in Section 7.2.3.

When its timer expires, a node samples its contention port once again. If it sees IDLE, then it sends a PN anew and waits for a CN signal as an acknowledgment. If, on the other hand, a node samples a PN on its port, it replies with a CN signal as an acknowledgement and becomes the root.

If both nodes pick different random bits, then the slowest (picking the bit one) is elected as leader. In the case that both nodes pick identical random bits, there are two possibilities. The root contention times allow one process to wait significantly longer than the other, even if both processes pick the same random bits. If this occurs, then the slower node becomes the root. Secondly, if the nodes proceed with about the same speed, then root contention reoccurs: when its timer expires, each node sees an IDLE signal and starts sending a PN signal, which will cause renewed root contention. In that case, the whole process is repeated until one of the nodes becomes root. Eventually (with probability one), both nodes will pick different random bits, in which case root contention certainly is resolved.

7.2.3 Protocol Timing Constraints and their Implications

The timing constraints

RCP contains five timing constants, which can all be treated as parameters. The minimum and maximum values of the RCP wait times, listed in Table 7.5, yield the parameters rc_fast_min, rc_fast_max, rc_slow_min and rc_slow_max. The communication delay $delay$ in the wires is the fifth parameter. It models the maximal total time from sending a signal by one node to receiving it by the other node. That is, 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.

We assume that all parameters take values in $\mathbb{R}^{\geq 0}$ and, moreover, we assume three basic parameter constraints

$$rc\_slow\_min \leq rc\_slow\_max, \tag{B_1}$$
$$rc\_fast\_min \leq rc\_slow\_min \text{ and} \tag{B_2}$$
$$rc\_fast\_min \leq rc\_fast\_max. \tag{B_3}$$

For the protocol to work correctly, two additional constraints are essential.

$$2 * delay < rc\_fast\_min. \tag{Eq_1}$$
$$2 * delay < rc\_slow\_min - rc\_fast\_max. \tag{Eq_2}$$

Note that we do not assume $rc\_fast\_max \leq rc\_slow\_max$ beforehand, but it follows from Eq_2. The timing constraints do not appear in the IEEE 1394 specifications [IEE96, IEE00a], but — fortunately — the wait times from the various standards (Table 7.5) do all meet these constraints.

The origin of these equations is visualized in Figure 7.6 and explained below. The analysis is based on informal notes [LaF97, Nyu97] to the IEEE P1394a Working Group. The equations from [LaF97] match ours, whereas [Nyu97] incorrectly cites [LaF97] and contains some errors. Due to an imperfection in the communication model, the scenario described below to illustrate the need for Eq_2 cannot occur in the model presented in Section 7.4. Hence, that section finds weaker constraint equations.

In the explanation below, we refer to the contending notes as Node 1 and Node 2. Recall that root contention is detected by both nodes individually.

Ad Equation Eq_1: In case of Node 2 selecting the short waiting period, constraint Equation Eq_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 7.6). Otherwise, the following erroneous scenario might happen: Node 2 might still see the first PN signal from Node 1, and erroneously send a CN signal to acknowledge this parent request. Once the IDLE signal from Node 1 arrives (behind schedule), Node 2 removes its CN signal again and declares 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 PN to Node 2. Awaiting the response it will time out, which leads to a bus reset. Therefore, constraint Equation Eq_1 is required for correct protocol operation.

Ad Equation Eq_2: This constraint ensures that root contention is always resolved in case of one node (say Node 1) selecting the short waiting period and the other (Node 2) selecting the long waiting period. More precisely, Eq_2 ensures that the new PN signal from Node 1 arrives at Node 2 *before* the waiting period of Node 2 ends (see circle 2 in Figure 7.6). Otherwise, Node 2 might still see the IDLE signal from Node 1 and sends another PN signal. Together with the PN message coming from Node 1 (after schedule), this will again lead to root contention, although the two nodes picked different random bits. Thus, this equation ensures that renewed root contention can only occur for if both nodes pick equal random bits.

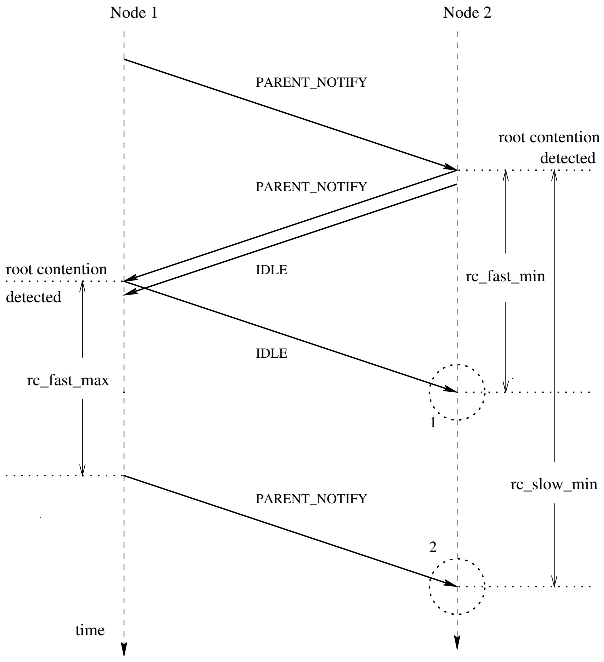

Figure 7.6: Visualization of the protocol timing constraints

Consequences of the timing constraints

The timing constraints have several consequences. Equations Eq_1 and Eq_2 can be rewritten as

$$delay < \frac{\min\left(rc\_fast\_min, rc\_slow\_min - rc\_fast\_max\right)}{2}.$$

Filling in the actual values of the wait times appearing in the various standards then yield an upper bound for the value of $delay$. We have $delay < 120$ ns for the IEEE 1394 standard, $delay < 370$ ns for IEEE 1394a and $delay < 800$ ns for P1394b. This influences the maximum cable and processing delays. Recall that the parameter $delay$ includes the propagation and the processing delay.

The cable propagation delay is specified to be less or equal than $5.05\ \mathrm{ns/m}$. Unfortunately, any additional processing delays are not explicitly specified in the standard. If, on the one hand, we disregard such extra delays, then the maximum cable length between two nodes is given by: $\frac{120\ \mathrm{ns}}{5.05\ \mathrm{ns/m}} \approx 24\ \mathrm{m}$ for 1394, approximately $73\ \mathrm{m}$ for 1394a and approximately $158\ \mathrm{m}$ is allowed for P1394b. However, for IEEE 1394 and 1394a, the cable length is already limited to $4.5\ \mathrm{m}$ by the worst case round trip propagation delay during bus arbitration.

If, on the other hand, processing delays cannot be ignored, then the current maximum cable length leaves room for a processing delay on each side of the wire of maximally $\frac{120-4.5\cdot 5.05}{2}$ ns ≈ 49 ns, $\frac{370-4.5\cdot 5.05}{2}$ ns ≈ 173 ns and $\frac{800-4.5\cdot 5.05}{2}$ ns ≈ 389 ns for IEEE 1394, 1394a and 1394b respectively.

If new applications require longer cable lengths — for instance, [IEE01b] expects $100\ \mathrm{m}$

cables to be needed for home networks — then the timing constraints above require either the root contention times to be increased, or the communication delay to be decreased. The latter is proposed in PIEEE 1394b. This proposal allows, besides copper wires, also glass optical fiber and plastic optical fiber to connect the devices. If glass optical fiber is used, then this standard proposal supports 100 meter cables, for plastic optical fibers this is 50 meter, while copper wires remain limited to 4.5 meter. Since glass optical fiber transmits signals nearly with the speed of light, the pure signal propagation delay cannot be a limiting factor anymore. Furthermore, the IEEE 1394 working groups also investigate alternative root contention protocols [LaF97] which allow for longer cable lengths.

7.3 Experiences with verifying the IEEE 1394 Root Contention protocol

This section compares several approaches to the verification of IEEE 1394 RCP and reports on the experiences and lessons to be learned when applying formal methods to industrial applications. Although RCP is small and easy to understand, the problems encountered in the verification of this protocol are in many aspects illustrative for the application of formal methods to other real–life applications.

7.3.1 Aspects of RCP

Several different features play a role in the modeling and analysis of RCP. Due to the use of random bits, the protocol is *probabilistic* in nature. *Real–time* is needed to model and analyze the root contend wait times and communication delays in the cables. Furthermore, *nondeterminism* is essential to model the fact that the root contend wait times and the communication delay are not fixed values, but lie within intervals. Within more abstract protocol descriptions, nondeterminism models the phenomenon that, if two nodes pick the same random bits, then either one of them is elected as leader or root contention reoccurs. Finally, *parametric models* treat the values of the timing delays as parameters rather than as fixed constants.

Moreover, several properties are of interest for the protocol's correctness. *Safety properties* are properties stating that "nothing bad ever happens" during execution of the protocol. A crucial safety property for RCP is that at most one leader is elected. *Liveness properties* state that "eventually, something good happens." An important liveness property is that at least one leader is elected (with probability one). Furthermore, *performance issues* concern the quantitative behavior of the protocol, for instance the probability that a leader is elected within a certain amount of time or the average number of rounds needed to elect a leader. Each of the properties mentioned can be tackled either parametrically or nonparametrically.

Obviously, it is impossible to consider all those features and properties at the same time. Therefore, each of the works described below focuses on one or more aspects, while abstracting from others.

Organization of the section This section is organized as follows. We first consider papers that study the functional and timing behavior of RCP in Section 7.3.2. We discuss several approaches to parametric verification in Section 7.3.3. Then, Section 7.3.4 presents several studies of the performance analysis of the protocol. Finally, Section 7.3.5 presents some conclusions. Within each subsection, the works are discussed in chronological order.

7.3.2 Functional and Timing behavior of RCP

Verification of RCP using stepwise refinement

The papers [Sto99b, SV99b, SS01] (where [SV99b] and [SS01] correspond to Sections 7.4 and 7.5 of this thesis.) all follow an automaton–based approach to verify RCP. The protocol and its specification are both described as automata, respectively $\mathtt{Impl}$ and $\mathtt{Spec}$, and correctness is expressed by $\mathtt{Impl} \sqsubseteq \mathtt{Spec}$. Here, $\sqsubseteq$ is a suitable notion of trace inclusion. The protocol correctness is established by stepwise abstraction: it is shown that $\mathtt{Impl} \sqsubseteq \mathtt{I}_1 \sqsubseteq \mathtt{I}_2 \sqsubseteq \mathtt{I}_3 \sqsubseteq \mathtt{Spec}$. Here the mentioned works consider different versions of $\mathtt{Impl}$ and $\mathtt{I}_1$, where $\mathtt{I}_1$ is obtained by abstracting from the communication in $\mathtt{Impl}$. Automaton $\mathtt{I}_2$ removes all timing information from $\mathtt{I}_1$ (in the discrete time case $\mathtt{I}_1 = \mathtt{I}_2$) and in $\mathtt{I}_3$ internal choices are further contracted. The main probabilistic analysis is carried out in the step $\mathtt{I}_2 \sqsubseteq \mathtt{I}_3$, which concerns very small automata.

Discrete time model ([Sto99b]) As a starting point for further verification, [Sto99b] describes a probabilistic, discrete time model of the protocol. It uses the probabilistic I/O automata framework developed by Segala [Seg95b]. The abstraction to discrete time is justified by the observation that RC_SLOW is about 2 times RC_FAST and that the communication delay is negligible compared to the root contention wait times.

In this model, the probabilistic behavior (in combination with fairness) has been studied. Most of the verification has been done manually, but several invariants and fairness properties have been checked with the model checker SMV [McM92]. It turned out that it is not so difficult to model the protocol in SMV and check the desired properties. However, the formal relationship between the I/O automaton model and the derived SMV model, needed to infer the correctness of the I/O automaton model model from the correctness of the SMV model, involves many technical details.

Real–time model ([SV99b] and Section 7.4 of this thesis.) In order to study the real–time behavior, timing has been modeled more precisely in [SV99b], using the probabilistic timed I/O automaton [Seg95b]. As in the discrete time model, the communication between the nodes is modeled as the transfer of packages (PN or CN). That is, single messages which are sent only once and, upon receipt, removed from the wire. The analysis of this model has been done manually, where the constants rc_fast_min, rc_fast_max, rc_slow_min, rc_slow_max and $delay$ are treated as timing parameters. Two constraints on these parameters are derived that ensure protocol correctness:

$$delay < rc\_fast\_min, \qquad (Eq_0)$$

$$2 * delay < rc\_slow\_min - rc\_fast\_max. \qquad (Eq_2)$$

These constraints differ from the constraints Eq_1 and Eq_2 in Section 7.2.3, which shows that the model in [SV99b] does not conform to the IEEE standard.

Detailed model ([SS01] and Section 7.5 of this thesis.) A close inspection of the IEEE documentation yielded that it is inappropriate to model the communication between the nodes by a packet mechanism as in [Sto99b, SV99b]. This is for two reasons. First, it is necessary to model the absence of a message (IDLE) explicitly. Secondly, signals, unlike packages, may remain unseen by the receiving node. This is the case if a second signal (possibly IDLE) arrives at the receiving node's port while the node has not sampled its port since the first signal has arrived.

This analysis yielded a more detailed model $\mathtt{Impl}^{\mathtt{B}}$, where the communication has been modeled by signals. More precisely, the model uses events to represent changes in the signals — the signals themselves are driven continuously across the wire. There is one (minor) point where the model in [SS01] does not conform to the IEEE standard: initially, both nodes in this model detect root contention simultaneously, whereas one can infer from the standard that any delay less than $delay$ is allowed between the detection of root contention by both nodes. When root contention reoccurs, the model in [SS01] does, however, allow for exactly this maximal delay between both detections.

Since the probabilistic analysis of this model is very similar to the real–time model, [SS01] only considers the timing aspects. By checking several safety and liveness properties for many different parameters values with the model checker Uppaal, [SS01] verified experimentally that the equations Eq_1 and Eq_2 from [LaF97] are necessary and sufficient for correct protocol operation. As is the case with SMV, it is not difficult to model the protocol in Uppaal, but the formal relationship between the I/O automaton model and the Uppaal model involves many technical details.

Modeling RCP with E–LOTOS

E–LOTOS Shankland et al. [Sha99, SV99a] present a formal description of RCP in E–LOTOS – an extension of LOTOS with time – of the entire tree identify phase in 1394, including RCP. An advantage of E–LOTOS is its similarity with programming languages, making it easy to read for engineers, see [MS00]. Since tools for this language have not been developed yet, no rigorous verification is carried out for the E–LOTOS models.

Although created independently, the models [Sha99, SV99a] (the RCP part) and [SV99b] are quite similar and both do not completely comply to the standard. Each of these works models the communication by a packet mechanism. Secondly, in [Sha99, SV99a], a CN is sent immediately after a PN has been detected, whereas the standard requires to wait at least the minimal root contention time. It is said in [SV99a, MS00] that this has been done because checking for a message after the waiting time has been expired is not expressible in E–LOTOS. If this is indeed the case, then this would plead for an extension of E-LOTOS with more expressive means.

Being integrated in the tree identify phase, the nodes in [Sha99, SV99a] automatically detect root contention asynchronously, in less than $delay$ time one after another, also in the initial state, which was not the case in [SS01].

7.3.3 Parametric Verification of RCP

Currently, several model checkers exist that can verify parametric timed systems, namely HyTech [HHW97], LPMC [gDUoT], TReX [ABS01] and a parametric extension to Uppaal [HRSV01]. All have been applied to RCP.

Given a system model $\mathcal{A}$, a property ϕ and (optionally) an initial parameter constraint C_0, the aim of parametric verification is to synthesize a parameter constraint C_1 which describes the exact conditions on the parameter values needed and sufficient for $\mathcal{A}$ to satisfy ϕ, where we may assume that the parameters meet C_0. Formally, we require that $C_0 \implies (\mathcal{A} \models \phi \iff C_1)$. Although this problem is undecidable ([AHV93]) for the input models used by the tools (timed automata), the tools mentioned about did manage to do parameter analysis for several practical examples, including RCP.

LPMC Toetenel and his team ([BLdRT00, BST00]) have used their parametric model checker LPMC to investigate the timing constraints of RCP. Here the values of $delay$ and (in some cases) $rc\_slow\_min - rc\_fast\_max$ are taken as parameters. The other values are taken as constants. Since the time and memory consumption in the verification was very modest, one might wonder why [BLdRT00, BST00] did not analyze a fully parametric model.

The entire verification is done with LPMC, which is unlike [SS01, HRSV01, CS01], where additional machinery is needed to deal with liveness properties and with probabilistic choice. The probabilistic choice has been replaced with a fairness property. Since only functional behavior is considered, this is appropriate, as the fairness property is implied by the probabilistic behavior of the protocol. The model in [BLdRT00] is similar to the model in [SV99b]; [BST00] is similar to [SS01]. The same timing constraints are found for the corresponding models.

However, by designating a different initial state, the model in [BST00] allows the nodes to detect root contention asynchronously also when the protocol starts. Thus, this model in [BST00] does completely conform to the standard.

Parametric Uppaal ([HRSV01] and Section 7.5 of this thesis.) The work [HRSV01] verified the models in [SV99b] and [SS01] with a parametric extension of the model checker Uppaal (see also Chapter 6), where all the five constants of RCP are treated as parameters. The sufficiency of the constraints in [SV99b] and [SS01] was checked by providing these as initial constraints to the tool and checking that no extra constraints are needed for the property under investigation to hold. For several cases, necessity is established similarly, that is, by checking that the property fails for all parameter values satisfying the negated constraints. This approach is different from [BLdRT00, BST00, CS01], which all synthesize the constraints.

The special format of the automata modeling RCP allowed certain parameters to be eliminated and still to obtain general results, which gained serious speed ups.

TReX Another parametric verification of RCP has been carried out by Collomb–Annichini & Sighireanu [CS01]. Using their TReX tool [ABS01] as well as HyTech, [CS01] analyzed liveness and safety properties of variations of `Impl`, including a model that allows for asynchronous detection of root contention in the initial state. All 5 timing constants are taken as parameters and the same constraints as in [SS01, HRSV01] are established.

TReX synthesizes the constraints for RCP automatically. Since TReX overapproximates the constraints for which the property does not hold, several runs of the tool with different initial constraints are needed to derive the constraints which are sufficient for the property under investigation to hold. In this way, [CS01] derives the constraints for several properties.

Moreover, [CS01] establishes that $\mathtt{Impl} \sqsubseteq \mathtt{I}_1$ if the parameters meet the constraints Eq_1 and Eq_2. Here, Eq_1 and Eq_2 are given as initial constraints, not synthesized. The notion of observation in TReX allows to smoothly check whether $\mathtt{Impl} \sqsubseteq \mathtt{I}_1$, whereas [HRSV01] needs rather complicated constructions on timed automata.

The performance of TReX is worse, both in time and in space, than the parametric extension to Uppaal, also in the cases where constraints were checked and not synthesized. This is explained by the fact that TReX is more general than Parametric Uppaal. In particular, it is able to deal with nonlinear constraints, which require more complex comparison algorithms. Compared with HyTech, TReX is slower, but HyTech was not able to synthesize the constraints for RCP and run out of memory for several crucial properties.

Comparison All tools are able to analyze the constraints for RCP and report the same results. Each of the approaches has its own weak and strong points – basically determined by the power of underlying tools. LPMC is the only tool that can handle fairness, but analyzes only 2 parameters; parametric Uppaal is fast, but did not synthesize the constraints; TReX can synthesize the constraints, but since it overapproximates the constraints, multiple runs of the tool are needed. Thus, it seems reasonable to expect that a combination of the techniques implemented in the various tools will yield further improvements.

7.3.4 Performance Analysis of RCP

Performance analysis aims at a quantitative analysis of a system. Interesting performance measures for RCP are for instance the minimal probability to elect a leader within a certain time or the maximal average number of rounds needed before a leader is elected. Traditionally, performance analysis techniques have been developed for purely probabilistic systems and cannot be applied directly to systems combining probability with nondeterminism, such as RCP. Moreover, most performance measures are no longer expressible as a single number, but, depending on how the nondeterminism is resolved, yield a number within an interval. Several performance analysis methods have been extended for systems with nondeterminism in [LSS94, BA95, Alf97, McI99, KNSS01]. Since the timing delays of RCP lie in intervals, rather than having fixed values, only [LSS94, KNSS01] are directly applicable here.

Thus, the approach taken by [D'A99] and [FS01] is to remove the nondeterminism and replace it respectively by a probabilistic and a deterministic choice. The work [KNS01] is one of the few case studies taking the challenge of doing performance analysis in the presence of nondeterminism.

Spades D'Argenio [D'A99] investigates the performance of RCP using a discrete event simulator for ♤ (Spades). ♤ is a stochastic process algebra which allows to specify delays by arbitrary probability distributions. Discrete event simulation is similar to testing in the sense that many runs from a system are taken, for which the performance measures are then computed. However, since all choices are probabilistic, one can exactly quantify the accuracy of the simulation — which is high if very many runs are taken.

The protocol model is based on [SV99b]. Although the standard specifies timing delays to be taken nondeterministically within their respective intervals, [D'A99] assumes a uniform distribution for the root contention times and β–distribution for the communication delay. Since techniques and tools for doing performance analysis in the presence of nondeterminism and real–time hardly exist, resolving the nondeterministic choices by probabilistic ones is currently the best one can do. The analysis shows that, in most of the cases, root contention is resolved in one round of the protocol. It also revealed that both the average time until root contention is resolved and its variance grow approximately linearly with the cable length.

Deadline properties Kwiatkowska, Norman and Sproston [KNS01] focus on a quantitative analysis of RCP and study *deadline properties* of the protocol. Given a deadline of d ns, they compute the minimal probability that RCP elects a leader within that deadline, for different values of d. Such deadline properties can be verified automatically by the tool Prism [dAKN$^+$00, KNP02] for systems modeled as finite Markov decision processes (MDPs).

The models analyzed are $\mathtt{Impl}$ and $\mathtt{I}_1$ from [SS01], augmented with probabilities $\frac{1}{2}$ for the outcomes of the coin flips. Since $\mathtt{Impl}$ and $\mathtt{I}_1$ include real–time, these do not fall into the class of MDPs. Three techniques are used to translate $\mathtt{I}_1$ into a finite MDP: the first uses the forward reachability method from [KNSS01] implemented in Hytech; the second partitions the state space of $\mathtt{I}_1$ according to the region equivalence as in [AD94]; and the last one interprets $\mathtt{I}_1$ in integer semantics, which yields in this case a model that is equivalent to the standard real semantics, but which is finite. The resulting MDPs were then given as input to Prism and all yield (upon termination) the same minimal probability for electing a leader within the choose deadlines. Since $\mathtt{Impl} \sqsubseteq \mathtt{I}_1$, we know that the minimal probability to elect a leader within deadline d for $\mathtt{I}_1$ is a *lower bound* for the minimal probability for $\mathtt{Impl}$ to do so. This needs not be the exact probability, since it is not known whether $\mathtt{I}_1 \sqsubseteq \mathtt{Impl}$ – probably so, but a firm result would be useful here.

The model $\mathtt{Impl}$ was analyzed using integer semantics only, which was the most efficient technique for the analysis of $\mathtt{I}_1$. Since the state space of the generated MDP grows with the deadline d, smaller deadlines can be analyzed for $\mathtt{Impl}$ than for $\mathtt{I}_1$. The minimal probabilities for $\mathtt{Impl}$ are the same as for $\mathtt{I}_1$, thus suggesting that $\mathtt{I}_1 \sqsubseteq \mathtt{Impl}$ indeed.

pGCL A third analysis of the performance of RCP has been carried out by Fidge & Shankland [FS01] using pGCL (probabilistic guarded command language). The language pGCL [MM99] is a probabilistic extension of Dijkstra's GCL where pre– and postcondition predicates no longer yield booleans, but values in $[0, 1]$ representing probabilities.

The model analyzed in [FS01] is a high–level description of the protocol in which no nondeterminism is present. Instead, root contention always reoccurs when two nodes pick the same random bits. When computing the worst case performance, this abstraction is appropriate. However, a formal proof to justify this would be valuable.

Using pGLC proof rules, [FS01] establishes that the probability to terminate in M rounds of the protocol equals $1 - (p^2 + (1-p)^2)^M = 1 - (1 - 2p + 2 \cdot p^2)^M$, where p is the probability to select fast timing. Since a round is bounded by $\mathit{rc\_slow\_max}$ (communication delays are neglected), the probability to terminate within a deadline $M \cdot \mathit{rc\_fast\_max}$ is at least $1 - (1 - 2p + 2 \cdot p^2)^M$. These lower bounds are strictly smaller than the exact ones derived by [KNS01]. This can be explained by the fact that, if both nodes select fast timing, then a round is bounded by the smaller figure $\mathit{rc\_fast\_min}$. However, [FS01] present an easy to compute symbolic expression, whereas the tool [KNS01] computes exact bounds for a fixed number of deadlines.

Besides RCP, this work also provides a pGCL analysis of the tree identify phase, of which RCP is a part.

Future work Several interesting performance aspects of RCP remain to be investigated, such as the minimum and maximum average number of rounds and the minimal and maximum average time before a leader is elected. The works [D'A99, KNS01, FS01] give a good starting point here.

A useful fact for efficiency reasons, which has been successfully exploited in [KNS01], is that the implementation relation $\sqsubseteq$ mentioned above preserves many relevant performance measures (namely those that can be expressed by traces). In other words, if $\mathcal{A} \sqsubseteq \mathcal{B}$ then $\mathcal{A}$

does not perform worse than $\mathcal{B}$ for those measures. If we also have $\mathcal{B} \sqsubseteq \mathcal{A}$, then $\mathcal{A}$ and $\mathcal{B}$ satisfy *exactly* the same performance measures.

Moreover, we remark that the minimal and maximal average *number of rounds* can be computed easily with the methods by [BA95, Alf97] on the automaton $\mathtt{I}_3$, because we can abstract from the exact timing delays. However, no techniques exist yet for calculation of the average *time* before a leader is elected; an extension of the results by [BA95, Alf97] to probabilistic timed automata would be useful here.

A funny property of RCP is that the performance can (theoretically) be improved by using a biased coin. More precisely, both the minimal and the maximal time before a leader is elected become smaller, if the probability to choose fast timing (random bit zero) slightly increases. This is the case because, although the minimal and maximal number of rounds are smallest if an unbiased coin is used, the time per round is lowest when both processes select fast timing. Therefore, favoring fast timing a little bit, yields faster termination of the protocol.

7.3.5 Concluding Remarks

From the papers [Sha99, SV99a, BLdRT00, BST00, CS01, D'A99, KNS01, FS01] and our own experiences with the formal verification of the IEEE 1394 Root Contention protocol, we conclude the following.

In order for the results of a formal verification to be reliable, the protocol models must – of course – be realistic. Constructing a realistic protocol model is, however, not easy. It is unavoidable to abstract from certain details in the standard, but it is hard to judge whether these abstractions are appropriate. This is hampered by the fact that industrial standards are often informal, incomplete and difficult to read for nonexperts.

Since it turned out to be inappropriate to model the communication delay between the nodes by a packet mechanism in RCP, it is worthwhile considering to what extent this is appropriate in the other parts of the Tree Identify Phase.

For a maximal profit from tool support, it is desirable to have more established translations between different formalisms and input languages of tools. Encoding a model specified in a certain formalism into the input language of a tool often involves a lot of technical details due to differences in synchronization principles, fairness assumptions, priorities, etc. Since such encodings can often be automated quite easily, a lot of time verification effort could be saved.

7.4 First Approach: A Manual Verification of the Root Contention Protocol

The rest of this chapter is devoted to the verification of RCP, using three different techniques. We start with a manual verification.

The verification in this section is carried out in the PA model [Seg95b, SL95], described in Chapter 2. More precisely, we use two extensions of this model, viz. probabilistic I/O automata and timed probabilistic I/O automata. These are also due to Segala [Seg95b] and present below. Following a well–tried method, we prove the correctness of the protocol by establishing a probabilistic simulation between the implementation and the specification, both modeled as (timed) probabilistic I/O automata.

The probabilistic simulation relations from [Seg95b, SL95] are rather complex. To simplify the simulation proofs, we use the simpler kind of simulation relations, namely probabilistic step refinements and probabilistic hyperstep refinements, which were introduced in Section 2.5.

The strategy followed in the simulation proof is the following. Given the protocol automaton $\mathtt{Impl}^{\mathtt{A}}$ and the abstract specification $\mathtt{Spec}$, we define three intermediate automata $\mathtt{I}_1^{\mathtt{A}}$, $\mathtt{I}_2^{\mathtt{A}}$, and $\mathtt{I}_3^{\mathtt{A}}$. First, $\mathtt{I}_1^{\mathtt{A}}$ abstracts from the message passing in $\mathtt{Impl}^{\mathtt{A}}$, but keeps the same probabilistic choices and most of the timing information. Next, $\mathtt{I}_2^{\mathtt{A}}$ abstracts from all the timing information in $\mathtt{I}_1^{\mathtt{A}}$, and $\mathtt{I}_3^{\mathtt{A}}$ abstracts from the probabilistic choice in $\mathtt{I}_2^{\mathtt{A}}$. The introduction of the intermediate automata allows us to separate our concerns. The simulation between $\mathtt{Impl}^{\mathtt{A}}$ and $\mathtt{I}_1^{\mathtt{A}}$ is easy from a probabilistic point of view and its proof mainly involves traditional, non–probabilistic techniques like proving invariants. The remaining simulations between automata $\mathtt{I}_2^{\mathtt{A}}$, $\mathtt{I}_3^{\mathtt{A}}$ and $\mathtt{Spec}$ deal with probabilistic choice, but since these automata are small this is not so difficult anymore.

Randomized algorithms and protocols have been the topic of various case studies. For instance, [LSS94] analyzes time bounds in the randomized dining philosopher's problem, in [Agg94] a number of algorithms for constructing a spanning tree in a network are designed and verified, [PSL97] verifies the randomized consensus protocol by Aspnes & Herlihy. All these case studies are carried out in the probabilistic I/O automata framework, but only the verification in the latter work is (partially) based on simulation relations; the others use different techniques.

This rest of this section is organized as follows. Section 7.4.1 introduces the probabilistic I/O automaton and the timed probabilistic I/O automaton model. Section 7.4.2 presents the automaton models describing the protocol implementation and specification. Then, Section 7.4.3 defines the intermediate automata and establishes the protocol correctness via the simulation relations. Finally, Section 7.4.4 presents some concluding remarks.

7.4.1 Probabilistic I/O Automata

The protocol analysis in this section is carried out in the probabilistic I/O automaton framework. This is a variation of the PA framework, where each PA is equipped with a distinction between input and output actions, and with a notion of fair behavior. The probabilistic I/O automaton model enables, even more than the PA model, unambiguous, concise and understandable system models, see [Lyn96] for supporting arguments. This section introduces the probabilistic I/O automaton model together with several results and operations we need in our verification.

Definition 7.4.1 A *probabilistic I/O automaton* $\mathcal{A}$ is a probabilistic automaton enriched with

1. a partition of $V_{\mathcal{A}}$ into *input actions* $in_{\mathcal{A}}$ and *output actions* $out_{\mathcal{A}}$, and
2. a *task partition* $tasks_{\mathcal{A}}$, which is an equivalence relation over $out_{\mathcal{A}} \cup int_{\mathcal{A}}$ with countably many equivalence classes.

We require that $\mathcal{A}$ is *input enabled*, which means that for all $s \in S_{\mathcal{A}}$ and all $a \in in_{\mathcal{A}}$, there is a μ such that $s \xrightarrow{a}_{\mathcal{A}} \mu$.

As probabilistic I/O automata are enriched PAs, we can use the notions of nonprobabilistic variant, reachable state, execution and trace also for probabilistic I/O automata. For their definitions, see Section 2.3.1 on page 40. The set of executions (execution fragments, traces) and finite executions (execution fragments, traces) of $\mathcal{A}$ are respectively denoted by $execs(\mathcal{A})$ ($frags(\mathcal{A})$, $traces(\mathcal{A})$) and by $execs^*(\mathcal{A})$ ($frags^*(\mathcal{A})$, $traces^*(\mathcal{A})$).

In the protocol analysis, there is one more operation we need, namely, hiding of actions. This operation turns external actions into internal ones, so that no synchronization takes place on those actions. We define this operation on the level of PAs, rather than on probabilistic I/O automatons.

Definition 7.4.2 If $\mathcal{A}$ is a PA and $X \subseteq ext_{\mathcal{A}}$, then $hide(\mathcal{A}, X)$ is the probabilistic automaton $(S_{\mathcal{A}}, S^0_{\mathcal{A}}, (ext_{\mathcal{A}} \setminus X, int_{\mathcal{A}} \cup X), \Delta_{\mathcal{A}})$.

Definition 7.4.3 Let $\mathcal{A}$ be a probabilistic I/O automaton. An execution α of $\mathcal{A}$ is *fair* if the following conditions hold for each class C of $tasks_{\mathcal{A}}$:

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

Definition 7.4.4 Let $\mathcal{A}$ and $\mathcal{B}$ be probabilistic automata with $ext_{\mathcal{A}} = ext_{\mathcal{B}}$. Let r be a mapping from $S_{\mathcal{A}}$ to $S_{\mathcal{B}}$. Then r induces a relation $\tilde{r} \subseteq frags(\mathcal{A}) \times frags(\mathcal{B})$ as follows: if $\alpha = s_0 a_1 s_1 \cdots \in frags(\mathcal{A})$, $\mathcal{I}$ is the index set of α, $\beta = t_0 b_1 t_1 \cdots \in frags(\mathcal{B})$ 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) $a_i \in I_{\mathcal{A}} \wedge b_{m(i)} \in I_{\mathcal{B}} \wedge m(i) = m(i-1) + 1$ or

   (c) $a_i \in I_{\mathcal{A}} \wedge m(i) = m(i-1)$.

In [PSL97], 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 7.4.5 ([SV02a]) *Let $\mathcal{A}$ and $\mathcal{B}$ be probabilistic I/O automata. Let r be a probabilistic step refinement from $\mathcal{A}$ to $\mathcal{B}$ that relates each fair execution of $\mathcal{A}$ only to fair executions of $\mathcal{B}$. Then $\mathcal{A} \sqsubseteq_{\mathrm{FTD}} \mathcal{B}$.*

Timed probabilistic I/O automata

Timed probabilistic I/O automata extends probabilistic I/O automata with time passage actions, c.f. Definition 2.2.10.

Definition 7.4.6 A *timed probabilistic I/O automaton* $\mathcal{A}$ is a probabilistic automaton enriched with a partition of $ext_{\mathcal{A}}$ into *input actions* $in_{\mathcal{A}}$, *output actions* $out_{\mathcal{A}}$, and the set $\mathbb{R}^{>0}$ of positive real numbers or *time-passage actions*. We require[1] that, for all $s, s', s'' \in S_{\mathcal{A}}$ and $d, d' \in \mathbb{R}^{>0}$ with $d' < d$,

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

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 $\mathcal{A}$ is *diverging* if the sum of the time-passage actions in α is ∞.

Definition 7.4.7 Let $\mathcal{A}, \mathcal{B}$ be timed or untimed probabilistic I/O automata. A function r is a *probabilistic (hyper)step refinement* from $\mathcal{A}$ to $\mathcal{B}$ if r is a probabilistic (hyper)step refinement from the underlying probabilistic automaton of $\mathcal{A}$ to the underlying probabilistic automaton of $\mathcal{B}$.

In [SV02a], it is argued that, under certain assumptions (met by the automata studied in this section), $\sqsubseteq_{\mathrm{TD}}$ 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 7.4.8 ([SV02a]) *Let $\mathcal{A}$ be a timed probabilistic I/O automaton and let $\mathcal{B}$ be a probabilistic I/O automaton. Let r be a probabilistic step refinement from $hide(\mathcal{A}, \mathbb{R}^{>0})$ to $\mathcal{B}$ that relates each divergent execution of $\mathcal{A}$ only to fair executions of $\mathcal{B}$. Then $\mathcal{A} \sqsubseteq_{\mathrm{DFTD}} \mathcal{B}$.*

7.4.2 The Protocol model

An informal description of the protocol has been given in Section 7.2. The timed probabilistic I/O automata describing the behavior of these nodes are given in Figure 7.7, using the IOA syntax of [GLV97] extended with a simple form of probabilistic choice. For simplicity, we refer to the two contending nodes as node 1 and node 2. Roughly, the idea behind the protocol model is as follows. When node `p` has detected root contention, it first flips a coin (i.e., performs the action `Flip(p)`). If head comes up then it waits a short time, somewhere in the interval [*rc_fast_min*, *rc_fast_max*]. If tail comes up then it waits a long time, somewhere in the interval [*rc_slow_min*, *rc_slow_max*]. After the waiting period has elapsed, either no message

[1] For simplicity the conditions here are slightly more restrictive than those in [LV96b].

```
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(p: 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(p)
      pre status = unknown ∧ snt = req ∧ rec = req
      eff coin := { head 1/2
                  { tail 1/2 ;
        x := 0;
        snt := ⊥;
        rec := ⊥
    output Send(p, m)
      pre status = unknown ∧ snt = ⊥
        ∧ x ≥ rc_min(coin)
        ∧ m = if rec = ⊥ then req else ack
      eff snt := m
    input Receive(p, m)
      eff rec := m
    output Root(p)
      pre status = unknown ∧ snt = ack
      eff status := root
    internal Child(p)
      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 ≤ rc_max(coin))
      eff x := x + d
```

Figure 7.7: Node automaton.

from the contender has been received, or a parent request has arrived[2]. In the first case the node sends a request message to its contender (i.e., performs the action $\mathtt{Send}(\mathtt{p},\ req)$), in the second case it sends an acknowledgement message (i.e., performs the action $\mathtt{Send}(\mathtt{p},\ ack)$). As soon as a node has sent an acknowledgement, it declares itself to be the root (via the action $\mathtt{Root}(\mathtt{p})$), and whenever a node has received an acknowledgement it assumes that its contender will become root and it declares itself child (via the action $\mathtt{Child}(\mathtt{p})$). If a node that has sent a request subsequently receives a request, then it concludes that there is root contention again. In that case, 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 $\mathtt{p}$ ($\mathtt{p} = 1, 2$), displayed in Figure 7.7, has five state variables: variable $\mathtt{status}$ tells whether the node has become $root$, $child$, or whether its status is still $unknown$; variable $\mathtt{coin}$ records the outcome of the coin flip; variable $\mathtt{snt}$ records the last value (if any) that has been sent to the contender and may take values req, ack or $\perp$; similarly $\mathtt{rec}$ records the last value that has been received (if any); variable $\mathtt{x}$, finally, models the arbitration timer that records the time that has elapsed since root contention has been detected. We use two auxiliary functions $\mathsf{rc\_min}(\mathsf{c})$ and $\mathsf{rc\_max}(\mathsf{c})$ from Toss to Reals given by, for $\mathsf{c} \in \mathsf{Toss}$,

$$\begin{aligned}
\mathsf{rc_min}(\mathsf{c}) &\triangleq \ \textbf{if } \mathsf{c} = head \textbf{ then } \mathit{rc_fast_min} \textbf{ else } \mathit{rc_slow_min} \\
\mathsf{rc_max}(\mathsf{c}) &\triangleq \ \textbf{if } \mathsf{c} = head \textbf{ then } \mathit{rc_fast_max} \textbf{ else } \mathit{rc_slow_max}
\end{aligned}$$

Now it should not be difficult to understand the precondition/effect style definitions in Figure 7.7, except maybe for the definition of the $\mathtt{Time}(\mathtt{d})$ 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 $\mathtt{Root}(\mathtt{p})$ action, in the second case it will perform a $\mathtt{Child}(\mathtt{p})$ action, and in the third case there is contention and the automaton will flip a coin.[3] The last clause in the precondition of $\mathtt{Time}(\mathtt{d})$ enforces that a $\mathtt{Send}(\mathtt{p},\ \mathtt{m})$ action is performed within either *rc_fast_max* or *rc_slow_max* 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.

The two automata for node 1 and node 2 communicate via wires, which are modeled as the timed probabilistic automata $\mathtt{Wire}(1,\ 2)$ and $\mathtt{Wire}(2,\ 1)$ specified in Figure 7.8. We assume an upper bound $delay \geq 0$ on the communication delay.

The full system $\mathtt{Impl}^{\mathtt{A}}$ 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 7.9).

Remark 7.4.9 As Segala [Seg95b] points out in his thesis, it would be useful to study the theory of *receptiveness* [SGSL98] 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

[2]The IDLE signal is not modeled explicitly here and this model is therefore not completely conform standard, but it does provide useful insight in the probabilistic and real–time behavior.

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

```
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 ≤ delay
      eff x := x + d
```

Figure 7.8: Wire automaton.

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

Figure 7.9: The full system.

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 [SGSL98]. 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.

7.4.3 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 7.10. We will establish that $\mathtt{Impl}^{\mathtt{A}}$ implements Spec, provided the following two constraints on the parameters are

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

Figure 7.10: Specification.

met:[4]

$$delay < rc\_fast\_min \qquad (Eq_0)$$

$$2 \cdot delay < rc\_slow\_min - rc\_fast\_max. \qquad (Eq_2)$$

Moreover, we assume the basic constraint

$$rc\_fast\_min \le rc\_fast\_max \le rc\_slow\_min \le rc\_slow\_max.$$

Note that this is slightly stronger than the basic constraints B_1 – B_3 from Section 7.2. Moreover, recall that the parameters range over $\mathbb{R}^{\geq 0}$.

Within our proof, we introduce three intermediate automata $\mathtt{I}_1^\mathtt{A}$, $\mathtt{I}_2^\mathtt{A}$ and $\mathtt{I}_3^\mathtt{A}$, and prove that

$$\mathtt{Impl}^\mathtt{A} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_1^\mathtt{A} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_2^\mathtt{A} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_3^\mathtt{A} \sqsubseteq_{\mathrm{TD}} \mathsf{Spec}.$$

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

$$\mathtt{Impl}^\mathtt{A} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_1^\mathtt{A} \sqsubseteq_{\mathrm{DFTD}} \mathtt{I}_2^\mathtt{A} \sqsubseteq_{\mathrm{FTD}} \mathtt{I}_3^\mathtt{A} \sqsubseteq_{\mathrm{FTD}} \mathsf{Spec}.$$

$\mathtt{I}_1^\mathtt{A}$ is a timed probabilistic I/O automaton, which abstracts from all the message passing in $\mathtt{Impl}^\mathtt{A}$, while preserving the probabilistic choices as well as most information about the timing of the `Root(i)` events. $\mathtt{I}_2^\mathtt{A}$ is a probabilistic I/O automaton which is identical to $\mathtt{I}_1^\mathtt{A}$, except that all real-time information has been omitted. In $\mathtt{I}_3^\mathtt{A}$ the two coin flips from each node of the protocol are combined into a single probabilistic transition.

Invariants

We will show that there exists a probabilistic step refinement from $\mathtt{Impl}^\mathtt{A}$ to an intermediate automaton $\mathtt{I}_1^\mathtt{A}$. To establish a refinement, we first need to introduce a number of invariants for automaton $\mathtt{Impl}^\mathtt{A}$.

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)`,

[4] As mentioned before, the timing constraints here are different from those in Section 7.2, due to an imperfection in the communication model.

respectively. So, $\mathtt{x_1}$ denotes the clock variable of $\mathtt{Node(1)}$, $\mathtt{x_{12}}$ denotes the clock variable of $\mathtt{Wire(1,\ 2)}$, etc. Within formulas we further use the following abbreviations, for $\mathtt{i} \in \mathtt{P}$,

$$
\begin{array}{rcl}
\mathsf{Cont}(\mathtt{i}) & \triangleq & \mathtt{snt_i} = \mathit{req} \wedge (\mathtt{rec_i} = \mathit{req} \vee \mathtt{msg_{ji}} = \mathit{req}) \\
\mathsf{Wait}(\mathtt{i}) & \triangleq & \mathtt{snt_i} = \mathtt{rec_i} = \perp \\
\mathsf{rc\_min_i} & \triangleq & \mathsf{rc\_min}(\mathtt{coin_i}) \\
\mathsf{rc\_max_i} & \triangleq & \mathsf{rc\_max}(\mathtt{coin_i})
\end{array}
$$

Predicate $\mathsf{Cont}(\mathtt{i})$ states that node $\mathtt{i}$ 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 $\mathsf{Wait}(\mathtt{i})$ states that node has flipped the coin and is waiting for the delay time to expire; no message has been received yet. State function $\mathsf{rc\_min_i}$ gives the minimum delay time for node $\mathtt{i}$, and state function $\mathsf{rc\_max_i}$ the maximum delay time (both state functions depend on the outcome of the coin flip).

We claim that assertions (7.1)-(7.17) below are invariants of automaton $\mathtt{Impl^A}$.

$$\mathtt{x_i} \geq 0 \tag{7.1}$$
$$\mathtt{status_i} = \mathit{unknown} \wedge \mathtt{snt_i} \neq \mathit{req} \Rightarrow \mathtt{x_i} \leq \mathsf{rc\_max_i} \tag{7.2}$$
$$\mathtt{snt_i} = \mathit{ack} \Rightarrow \mathtt{x_i} \geq \mathsf{rc\_min_i} \tag{7.3}$$
$$\mathtt{status_i} = \mathit{root} \Rightarrow \mathtt{snt_i} = \mathit{ack} \tag{7.4}$$
$$\mathtt{status_i} = \mathit{child} \Rightarrow \mathtt{rec_i} = \mathit{ack} \tag{7.5}$$
$$\mathtt{x_{ij}} \geq 0 \tag{7.6}$$
$$\mathtt{msg_{ij}} \neq \perp \Rightarrow \mathtt{x_{ij}} \leq \mathit{delay} \tag{7.7}$$
$$\mathsf{Cont}(\mathtt{i}) \Leftrightarrow \mathsf{Cont}(\mathtt{j}) \Rightarrow \mid \mathtt{x_i} - \mathtt{x_j} \mid \ \leq \mathit{delay} \tag{7.8}$$
$$\mathsf{Cont}(\mathtt{i}) \wedge \neg\mathsf{Cont}(\mathtt{j}) \Rightarrow \mathsf{Wait}(\mathtt{j}) \wedge \mathtt{msg_{ij}} = \perp \wedge \mathtt{x_j} \leq \mathit{delay} \tag{7.9}$$
$$\mathtt{msg_{ij}} \neq \perp \Rightarrow \mathtt{rec_j} = \perp \tag{7.10}$$
$$\mathtt{msg_{ij}} = \perp \Rightarrow \mathtt{snt_i} = \perp \vee \mathtt{rec_j} \neq \perp \vee \mathsf{Cont}(\mathtt{i}) \tag{7.11}$$

$$\mathtt{msg_{ij}} = \mathit{req} \wedge \neg\mathsf{Wait}(\mathtt{i}) \Rightarrow \mathtt{snt_i} = \mathit{req} \wedge \mathtt{snt_j} \neq \mathit{ack} \wedge \mathsf{rc\_min_i} \leq \mathtt{x_i} - \mathtt{x_{ij}} \leq \mathsf{rc\_max_i} \tag{7.12}$$
$$\mathtt{msg_{ij}} = \mathit{req} \wedge \mathsf{Wait}(\mathtt{i}) \Rightarrow \mathtt{snt_j} = \mathit{req} \wedge \mathtt{x_i} \leq \mathtt{x_{ij}} \tag{7.13}$$
$$\mathtt{snt_i} = \perp \wedge \mathtt{rec_i} = \mathit{req} \Rightarrow \mathtt{snt_j} = \mathit{req} \wedge \mathtt{rec_j} = \perp \wedge \mathtt{x_j} \geq \mathsf{rc\_min_j} \tag{7.14}$$
$$\mathtt{rec_i} = \mathit{ack} \Rightarrow \mathtt{snt_j} = \mathit{ack} \tag{7.15}$$
$$\mathtt{msg_{ij}} = \mathit{ack} \Rightarrow \mathtt{snt_i} = \mathit{ack} \tag{7.16}$$
$$\mathtt{snt_i} = \mathit{ack} \Rightarrow \mathtt{rec_i} = \mathtt{snt_j} = \mathit{req} \wedge \mathtt{rec_j} \neq \mathit{req} \wedge \mathtt{x_j} \geq \mathsf{rc\_min_j} \tag{7.17}$$

Assertions (7.1)-(7.7) are local invariants, which can be proven straightforwardly for automata $\mathtt{Node(i)}$ and $\mathtt{Wire(i,\ j)}$ in isolation. Most of the time nodes 1 and 2 are either both in contention or both not in contention. Assertion (7.8) states that in these cases the values of the clocks of the two nodes differ by at most delay. Assertion (7.9) expresses that the only case where node $\mathtt{i}$ is in contention but the other node $\mathtt{j}$ is not occurs when $\mathtt{j}$ has just flipped a coin but the request message that $\mathtt{j}$ sent to $\mathtt{i}$ has not yet arrived or been processed. If a channel contains a message then nothing has been received at the end of this channel

(7.10). 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 (7.11). 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 (7.12), or there is contention and i has just flipped a coin (7.13). If i has received a request message without having sent anything, then j has sent this message but has not received anything (7.14). The last three invariants deal with situations where there is an acknowledgement somewhere in the system (7.15)-(7.17). 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 automaton $\mathtt{Impl}^{\mathtt{A}^-}$ and the invariants were translated into Uppaal's input language. This allowed us to check the invariants mechanically. This was done for a large number of parameter values, just as in the approach taken in Section 7.5.

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

Lemma 7.4.10 *Suppose state s satisfies assertions (7.1)-(7.17) and $s \xrightarrow{\mathtt{Send(i,\,m)}} s'$. Then $s \models \mathtt{msg_{ij}} = \mathtt{rec_j} = \perp$ and $s' \models \mathsf{Cont}(\mathtt{i}) \Leftrightarrow \mathsf{Cont}(\mathtt{j})$.*

Lemma 7.4.11 *Assertions (7.1)-(7.17) hold for all reachable states of* $\mathtt{Impl}^{\mathtt{A}}$.

Remark 7.4.12 The constraint Eq_0 on the timing parameters is used to prove Lemma 7.4.10 and ensures that there can never be two messages traveling 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. Since Lemma 7.4.10 is used to prove Lemma 7.4.11, the first constraint is also used in our proof of Lemma 7.4.11.

The parameter constraint Eq_2 is not used in either of the proofs above.

The first intermediate automaton

Intermediate automaton $\mathtt{I}_1^{\mathtt{A}}$ is displayed in Figure 7.11.

This probabilistic timed I/O automaton records the status for each of the two nodes to be either *init*, *head*, *tail*, or *done*. In addition $\mathtt{I}_1^{\mathtt{A}}$ 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 i is *head* and for j *tail*. A Retry(c) transition may occur if both nodes have flipped c. Clock x is used to express that both nodes flip their coin within time *delay* after the (re-)start of the protocol. In addition it ensures that subsequently (depending on the outcome of the coin flips) at least *rc_fast_min* − *delay* or *rc_slow_min* − *delay* time and at most *rc_fast_max* or *rc_slow_max* time will elapse before either a Root(i) or a Retry(c) action occurs.

automaton $\mathtt{I}_1^{\mathtt{A}}$
    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** $\mathtt{d} > 0$
    transitions
        internal Flip(i)
            pre $\mathtt{phase}[\mathtt{i}] = init$
            eff $\mathtt{phase}[\mathtt{i}] := \begin{cases} head & \frac{1}{2} \\ tail & \frac{1}{2} \end{cases}$;
                if $\mathtt{phase}[\mathtt{next}(\mathtt{i})] \neq init$ **then** $x := 0$
        output Root(i)
            pre $\{\mathtt{phase}[1], \mathtt{phase}[2]\} \subseteq \{head, tail\}$
                $\wedge \neg(\mathtt{phase}[\mathtt{i}] = head \wedge \mathtt{phase}[\mathtt{next}(\mathtt{i})] = tail)$
                $\wedge\ x \geq \mathsf{rc\_min}(\mathtt{phase}[\mathtt{i}]) - delay$
            eff $\mathtt{phase} := \mathsf{constant}(done)$
        internal Retry(c)
            pre $\mathtt{phase} = \mathsf{constant}(\mathtt{c})$
                $\wedge\ x \geq \mathsf{rc\_min}(\mathtt{c})$
            eff $\mathtt{phase} := \mathsf{constant}(init)$;
                $x := 0$
        delay Time(d)
            pre $init \in \{\mathtt{phase}[1], \mathtt{phase}[2]\} \Rightarrow x + \mathtt{d} \leq delay$
                $\wedge \{\mathtt{phase}[1], \mathtt{phase}[2]\} \subseteq \{head, tail\} \Rightarrow$
                    $x + \mathtt{d} \leq \max(\mathsf{rc\_max}(\mathtt{phase}[1]), \mathsf{rc\_max}(\mathtt{phase}[2]))$
            eff $\mathtt{x} := \mathtt{x} + \mathtt{d}$

Figure 7.11: Intermediate automaton $\mathtt{I}_1^{\mathtt{A}}$.

Proposition 7.4.13 $\mathtt{Impl}^{\mathtt{A}} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_1^{\mathtt{A}}$. *More specifically the conjunction, for* $\mathtt{i} \in \mathtt{P}$, *of*

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

determines a probabilistic step refinement from $\mathtt{Impl}^{\mathtt{A}}$ *to* $\mathtt{I}_1^{\mathtt{A}}$.

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

Remark 7.4.14 The constraint Eq_2 on the timing parameters is used in the proof of Proposition 7.4.13 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).

The first constraint Eq_0 is also used in our proof, since our proof of Proposition 7.4.13 uses Lemma 7.4.11, which in its turn uses the first constraint. However, we conjecture that if we prove the inclusion between $\mathtt{Impl}^{\mathtt{A}}$ and $\mathtt{Spec}$ without using intermediate automata, we do not need the first constraint. In fact, this constraint is an implicit assumption in the model, needed to justify the modeling of the communication channels by a one–place buffer.

Remark 7.4.15 In [And98] 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 $\mathtt{I}_1^{\mathtt{A}}$ 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 request message of node $\mathtt{i}$ arrives at node $\mathtt{j}$ before the timing delay of node $\mathtt{j}$ 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 section.

Remark 7.4.16 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 *rc_fast_min* and the other node j flips tail and selects a timing delay of *rc_slow_max*, our version elects a leader within at most *rc_fast_min* $+ 3\mathit{delay}$, whereas in the current version this upper bound is *rc_slow_max* $+ 3\mathit{delay}$.

The second intermediate automaton

In Figure 7.12 the second intermediate automaton $\mathtt{I}_2^{\mathtt{A}}$ is described. $\mathtt{I}_2^{\mathtt{A}}$ is a probabilistic I/O automaton that is identical to $\mathtt{I}_1^{\mathtt{A}}$ except that all real-time information has been abstracted away; instead a (trivial) task partition is included. The proof of the following Proposition 7.4.17 is easy: the projection function π from $\mathtt{I}_1^{\mathtt{A}}$ to $\mathtt{I}_2^{\mathtt{A}}$ trivially is a probabilistic step refinement (after hiding of the time delays).

```
automaton I_2^A
  states
    phase : Array[P, Phase] := constant(init)
  signature
    output Root(i: P)
    internal Flip(i: P),
             Retry(c: Toss)
  transitions
    internal Flip(i)
      pre phase[i] = init
      eff phase[i] := { head  1/2
                      { tail  1/2
    output Root(i)
      pre {phase[1], phase[2]} ⊆ {head, tail}
          ∧ ¬(phase[p] = head ∧ phase[next(p)] = tail)
      eff phase := constant(done)
    internal Retry(c)
      pre phase = constant(c)
      eff phase := constant(init)
  tasks
    One block
```

Figure 7.12: Intermediate automaton $\mathtt{I}_2^{\mathtt{A}}$.

Proposition 7.4.17 $\mathtt{I}_1^{\mathtt{A}} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_2^{\mathtt{A}}$.

Proposition 7.4.18 *If $\alpha \in execs(\mathtt{I}_1^{\mathtt{A}})$ is diverging and π relates α and β, then β is fair.*

The result formulated in the Proposition 7.4.18 above follows by the fact that a diverging execution of $\mathtt{I}_1^{\mathtt{A}}$ either contains infinitely many Retry actions, or contains an infinite suffix with a $\mathtt{Root(p)}$ transition followed by an infinite number of delay transitions. Now the Claim 7.4.8 implies $\mathtt{I}_1^{\mathtt{A}} \sqsubseteq_{\mathrm{DFTF}} \mathtt{I}_2^{\mathtt{A}}$.

The third intermediate automaton

Figure 7.13 gives the IOA code for the probabilistic I/O automaton $\mathtt{I}_3^{\mathtt{A}}$. This automaton abstracts from $\mathtt{I}_2^{\mathtt{A}}$ since it only has a single probabilistic transition. Within automaton $\mathtt{I}_3^{\mathtt{A}}$, $init$ 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_{p} indicates that the results are different and the outcome of p equals tail. In state $same$ both coin flips have yielded the same result.

Proposition 7.4.19 $\mathtt{I}_2^{\mathtt{A}} \sqsubseteq_{\mathrm{TD}} \mathtt{I}_3^{\mathtt{A}}$. *More specifically, the following function r from (reachable) states of $\mathtt{I}_2^{\mathtt{A}}$ to discrete probability spaces over states of $\mathtt{I}_3^{\mathtt{A}}$ is a probabilistic hyperstep refine-*

automaton $\mathtt{I}_3^{\mathtt{A}}$
    type $\mathsf{Loc} =$ **enumeration of** $init, win_1, win_2, same, done$
    states
        $\mathtt{loc} : \mathsf{Loc} := init$
    signature
        output `Root(p: P)`
        internal `Flips`,
                `Retry`
    transitions
        internal `Flips`
            pre $\mathtt{loc} = init$
            eff $\mathtt{loc} := \begin{cases} win_1 & \frac{1}{4} \\ win_2 & \frac{1}{4} \\ same & \frac{1}{2} \end{cases}$
        output `Root(p)`
            pre $\mathtt{loc} \in \{win_{\mathtt{p}}, same\}$
            eff $\mathtt{loc} := done$
        internal `Retry`
            pre $\mathtt{loc} = same$
            eff $\mathtt{loc} := init$
    tasks
        One block

Figure 7.13: Intermediate automaton $\mathtt{I}_3^{\mathtt{A}}$.

ment from $\mathtt{I}_2^{\mathtt{A}}$ *to* $\mathtt{I}_3^{\mathtt{A}}$ *(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 7.4.20 and 7.4.21 can be found in [Sto99b]. 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: $\mathtt{I}_2^{\mathtt{A}}$ has 10 states, $\mathtt{I}_3^{\mathtt{A}}$ has 5 states, and `Spec` has 2 states.

Proposition 7.4.20 $\mathtt{I}_2^{\mathtt{A}} \sqsubseteq_{\mathrm{FTD}} \mathtt{I}_3^{\mathtt{A}}$.

Proposition 7.4.21

1. $\mathtt{I}_3^{\mathtt{A}} \sqsubseteq_{\mathrm{TD}}$ Spec. *More specifically, the function determined by the predicate done* $\Leftrightarrow$ loc $=$ *done is a probabilistic step refinement from* $\mathtt{I}_3^{\mathtt{A}}$ *to* Spec.

2. $\mathtt{I}_3^{\mathtt{A}} \sqsubseteq_{\mathrm{FTD}}$ Spec.

7.4.4 Concluding Remarks

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 (see Chapter 2) rather than the more general but also complex simulation relations (especially in the timed case!) which have been proposed by Segala and Lynch [Seg95b, SL95]. The complexity of the definitions in [Seg95b, SL95] 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 section, 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.

The idea to introduce auxiliary automata in a simulation proof has been studied in many papers, see for instance [AL91]. The verification reported in this section 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.

7.5 Second Approach: Mechanical Verification of the Root Contention Protocol

In this section, we present a mechanical verification of RCP. We improve the protocol model from the previous section and then use the model checker Uppaal to investigate the timing constraints of the new protocol model. Since the probabilistic phenomena in the new protocol model are basically the same as before, we do not reconsider probabilistic aspects in this section.

As pointed out in Section 7.2.3, timing parameters play an essential role in the root contention protocol. There are currently three tools[5] available that (at least in some cases, see [AHV93]) can do parametric analysis of timed systems: HyTech [HHW97], PMC [gDUoT] and – very recently – the tool described in [AAB00]. Whereas HyTech and PMC can currently analyze and derive linear parameter constraints, [AAB00] describes a prototype implementation which can also deal with nonlinear constraints. Since the performance of HyTech is limited, and we expected the protocol to be too complex for it, and since only prototypes of the other two tools are currently available, we decided to use the Uppaal tool. Uppaal has already been used successfully in various verifications [HSLL97, DKRT97b]. It can model real–time systems with a finite control structure. A limited class of properties, viz. reachability properties, can be checked automatically and (relatively) efficiently. Our protocol models fit naturally into its input language.

[5] At the time we carried out this research, the prototype extension of Uppaal was in a very preliminary stage.

Using the enhanced model, we investigate the correct operation of the root contention protocol with Uppaal. By analyzing numerous instances of the protocol for different values of the parameters, Uppaal allowed us to do an approximate parameter analysis. The constraints that we deduce are exactly those from [LaF97].

Unlike most other Uppaal case studies, we carry out the verification using intermediate automata, as in Section 7.4. The intermediate automata used in this section are similar to the ones in the previous section, but have been adapted to match the new protocol model. To do so, we need several constructions on Uppaal models. The works [Jen99, JLS00] discuss an other approach to this problem. The difference with our work is that [Jen99, JLS00] consider timed automata without invariants but with shared variables and urgent channels and require the specification automaton to be deterministic. We use the notion of step refinement to deal with nondeterministic specifications in certain cases.

The rest of this section is organized as follows. In Section 7.5.1, we describe how Uppaal can be used in verifying system correctness via stepwise abstraction. Section 7.5.2 gives the semantics of a Uppaal model. Moreover, this section explains how Uppaal can be used to verify *trace inclusion*, i.e. that one model in Uppaal is a correct implementation of another one. Section 7.5.3 presents the new protocol model and Section 7.5.4 is concerned with the verification and analysis of these models. Finally, in Section 7.5.5, conclusions are given.

7.5.1 Verification of Trace Inclusion with Uppaal

Uppaal [Upp, LPY97] is a tool box for modeling, simulation and automatic verification of real–time systems, based on timed automata. In the present case study, we used the Uppaal2k version. Uppaal can simulate a model, i.e. it can provide a particular execution of the model step by step, and it can automatically check whether a given reachability property is satisfied and, if so, provide a trace showing how the property is satisfied.

Model checking with Uppaal

This section gives an informal introduction to Uppaal and is based on [LPY97]. A system in Uppaal is modeled as a network of nondeterministic processes (called automata) with a finite control structure and real–valued clocks. Communication between the processes takes place via channels (via binary synchronization on complementary actions). 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.

Basically, a process is a finite state machine (or labeled transition system) extended with clock variables. The nodes of an automaton describe the control locations. Each location can be decorated with an invariant: a number of clock bounds expressing the range of clock values that are allowed in that location. The edges of the transition system represent changes in control locations. Each edge can be labeled by a *guard* g, an *action (label)* a, and a collection r of *clock resets*. All three types of labels are optional. A guard is a boolean combination of inequalities over clock variables, expressing when the transition is enabled, i.e. when it can be taken. Upon taking a transition, the clock resets, if present, are executed. The action label, if present, enforces binary synchronization. This means that exactly one of the other processes has to 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 location from which more than one transition is enabled with the same action. In this way, nondeterministic choices can be specified within Uppaal. Note that time can only

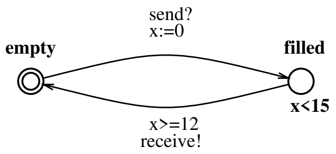

Figure 7.14: An example Uppaal process automaton (`Buffer`)

elapse at the locations (conform invariants). Transitions are taken instantaneously, i.e. no time elapses during transitions. Three special types of locations are available:

1. *Initial locations*, denoted by (O), define the initial state of the system (exactly one per process).

2. *Urgent locations*, denoted by (U), are locations in which no time can be spent, hence a shorthand notation for a location that satisfies the virtual invariant $x \leq 0$. The (fresh) clock x is reset on all transitions to the urgent location.

3. *Committed locations*, denoted by (C), are used to make the incoming and the outgoing transition atomic. Being in a committed location, the process execution cannot be interrupted and no time elapses. We used these locations to encode multi–way synchronization in Uppaal (see Section A.1.2).

Moreover, channels can declared as being urgent and shared integer variables can be used to communicate between the processes. Since we did not use these Uppaal features in our verification, we do not describe them here.

Consider the Buffer process automaton displayed in Figure 7.14. This automaton models a one–place buffer which delivers a message with a time delay between 12 and 15 time units.

Uppaal is able to analyze reachability properties automatically. These properties must be of the form $\mathrm{E}\Diamond p$ or $\mathrm{A}\Box p$, where p is Boolean expression over clock constraints and locations of the automata. For example, $\mathrm{E}\Diamond \texttt{Buffer.filled} \wedge x > 13$ is a property over the automaton `Buffer`. Informally, $\mathrm{E}\Diamond p$ denotes that there must be some state (= location + clock values) which is reachable from the initial state and which satisfies p. This logic is sufficient to specify reachability properties, invariants, and bounded liveness properties. For the latter, see [ABK98]. However, general liveness and fairness properties, e.g.whether an event occurs infinitely often, cannot be expressed in Uppaal.

Notational conventions

Throughout this chapterwe use the following conventions for automata.

We assume that the automata do not have urgent channels, committed locations, and shared variables and we assume that any two components in a network use different clocks.

We denote the absence of a transition label by a special symbol τ. When convenient, we assume that all labels on a transition are present, interpreting the absence of a guard or invariant by true and the absence of a reset set by $\emptyset$. We denote the invariant of a location q by $\mathrm{Inv}(q)$.

Moreover, we assume a fixed set of action names, Names, such that $\tau \notin \mathrm{Names}$, and for any subset of $N \subseteq \mathrm{Names}$, we write $N! = \{a! \mid a \in N\}$, $N? = \{a? \mid a \in N\}$. Then the set of discrete actions is given by $act = \mathrm{Names}? \cup \mathrm{Names}! \cup \{\tau\}$. The action τ is called the *invisible* or *internal* action; the other actions are called *visible*. The set of *visible actions* occurring in automaton A is the denoted by Act_A. By abuse of notation, we let a, b range over act, but in the expressions $a?$ and $a!$, a ranges over action names.

7.5.2 The Semantics of Uppaal Models

Similarly to the verification in the previous section, we express the behavior of a system by the set of its traces and use trace inclusion to express that one system correctly implements another one. The Uppaal tool interprets networks of automata as *closed systems*, which are systems that cannot interact with their environment. In closed systems we cannot express many sets of traces. (The traces of closed systems consists of their time passage actions.) Therefore, we provide – in addition to the standard Uppaal semantics – an interpretation of Uppaal models as *open systems*, which still have the possibility to interact with the environment. Then we define two operators, parallel composition $\|$ and action restriction $\setminus$ to express the closed system semantics in terms of the open system semantics. Finally, we give a translation of each open model to a closed model with equivalent reachability properties. In this way, we are able to verify all reachability properties of open systems (and in particular trace inclusion) with Uppaal.

We use the same (Uppaal) syntax for open and for closed systems. We use the word *automaton* if we interpret the model as an open system and the term *network of automata* if we use the standard closed system interpretation.

Open system semantics

Now, we give the open system semantics of an automaton A by its underlying timed labeled transition system. We may assume that A consists of one component, because we give the semantics of the parallel composition later in this section. Our treatment here is rather informal. For more rigorous definitions, the reader is referred to [AD94].

A *timed labeled transition system* (TLTS) consists of a set of states, a set of actions comprising the set of positive real numbers and a transition relation which relates certain states and actions to other states. The TLTS associated to an automaton A is given as follows.

The states (q, v) of the TLTS consist of a location q of A and a *clock valuation* v. The latter is a function that assigns a value in $\mathbb{R}^{\geq 0}$ to each clock variable of the automaton. We require that each state (q, v) of the TLTS underling A meets the location invariant of q. The initial state of the TLTS, if any, is given by the initial location q_0 in the automaton and the clock valuation v_0 that assigns 0 to each clock variable. (Note that there is no initial state if v_0 does not meet the location invariant of q_0.)

The transitions $s \xrightarrow{\ell} s'$ of the TLTS, labeled with a discrete or time passage action, indicate the following: Being in state $s = (q, v)$, that is, the system is in location q and the clocks have the values as described by v, the system can move from s to a new state s', in which the location and clock variables have been updated according to the delay or discrete step taken in the automaton. The *time passage actions* $s \xrightarrow{d} s'$ of the TLTS, where $d \in \mathbb{R}^{>0}$, augment all clocks with d time units and leave the locations unchanged. Such a transition is enabled if augmenting the clocks with d time units is allowed by $\mathrm{Inv}(q)$. The *discrete actions* $s \xrightarrow{a} s'$ change the state as specified by the transition in the network of automata. This means that the guard of the transition involved is met in s, that the invariant of the destination of the transition involved is met in s' and that the clock variables are set according to the transition involved. We label the transition in the TLTS by a special symbol τ if the corresponding transition in the automaton is unlabeled.

Hence, the TLTS has an infinite set of states, actions and (for non–trivial automata) transitions.

Example 7.5.1 The TLTS underlying the system depicted in Figure 7.14 consists of the states (empty, u) and (filled, u') for $u \geq 0$ and $u' < 15$ (where u and u' denote the valuations that assign respectively the values u and u' to the clock x), the initial state $(\mathit{empty}, 0)$ and the transitions

$$\begin{aligned}
&(\mathit{empty}, u) \xrightarrow{\texttt{send?}} (\mathit{filled}, 0), \text{for } u \geq 0,\\
&(\mathit{filled}, u) \xrightarrow{\texttt{receive!}} (\mathit{empty}, u), \text{for } 12 \leq u \leq 15,\\
&(\mathit{empty}, u) \xrightarrow{d} (\mathit{empty}, u + d), \text{for } d > 0,\\
&(\mathit{filled}, u) \xrightarrow{d} (\mathit{filled}, u + d), \text{for } d > 0, u + d < 15.
\end{aligned}$$

When interpreted as a closed system, the TLTS underlying `Buffer` would not have any transitions, because the actions `receive` and `send` cannot synchronize.

Definition 7.5.2

1. A *timed execution* of a TLTS is a possibly infinite sequence s_0, a_1, s_1, $a_2, \ldots$ such that s_0 is the initial state and s_i is a state, a_i a (discrete or delay) action, and $s_i \xrightarrow{a_{i+1}} s_{i+1}$ a transition.
2. A *timed execution* of an automaton is a timed execution of its underlying TLTS – a sequence (q_0, v_0), a_1, (q_1, v_1), a_2, $(q_2, v_2), \ldots$ where q_i and v_i, are a location and clock valuation respectively.
3. A state is *reachable* if there exists an execution passing that state.
4. A *trace* (of either an automaton or a TLTS) arises from an execution by omitting the states and internal actions. The sets of traces of a TLTS or an automaton A are both denoted by $\mathit{traces}(A)$. We write $A \sqsubseteq_{\mathrm{TR}} B$ if $\mathit{traces}(A) \subseteq \mathit{traces}(B)$.

Example 7.5.3 The state $(\mathit{filled}, 13)$ is reachable in the automaton `Buffer`. An execution passing it by is $(\mathit{empty}, 0)$, 18, $(\mathit{empty}, 18)$, `send?`, $(\mathit{filled}, 0)$, 10, $(\mathit{filled}, 10)$, 3, $(\mathit{filled}, 13)$. The states $(\mathit{filled}, 15)$ and $(\mathit{filled}, 16)$ are not reachable in this automaton.

The following theorem expresses the main result by Alur and Dill [AD94], upon which Uppaal and other model checkers based on timed automata are built.

Theorem 7.5.4 *The set of reachable states of an automaton is decidable.*

An important class of automata and TLTS is formed by the deterministic systems. Intuitively, determinism means that, given the current state and the action to be taken, the next state (if any) is uniquely determined.

Definition 7.5.5

1. A TLTS is called *deterministic* if there are no τ labeled transitions and for every state s and every action $a \in \mathit{act}$, there is at most one state t such that $s \xrightarrow{a} t$.

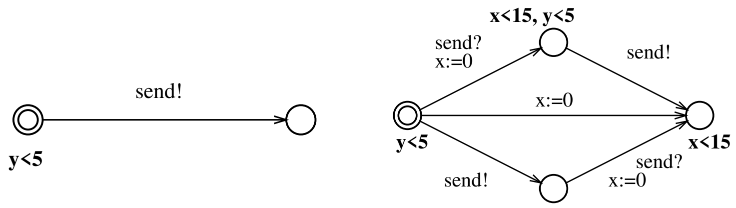

Figure 7.15: `Sender` and `Sender` $\|$ `Buffer` $\setminus$ $\{$`receive`$\}$

2. An automaton is called *deterministic* if there are no unlabeled transitions and $q \xrightarrow{a\ g\ r} q'$ and $q \xrightarrow{a\ g'\ r'} q''$ implies that $q' = q'' \wedge r = r'$ or that $g \wedge \mathrm{Inv}(q)$ and $g' \wedge \mathrm{Inv}(q)$ are disjoint, (i.e. $g \wedge g' \wedge \mathrm{Inv}(q)$ is unsatisfiable).

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

Parallel composition of automata

The parallel composition operator $\|$ which we consider is basically the composition as in CCS [Mil80]. This means that the automaton $A_1 \parallel A_2$ contains the control locations of both automata. A transition in the parallel composition corresponds to either one of the components taking a transition and the other one remaining in the same location or to both components taking a transition simultaneously, while synchronizing on complementary actions $a?$ and $a!$. Synchronization yields an invisible action in the composition (and hence no other components can synchronism with the same action).

Definition 7.5.6 The *parallel composition* of two automata, A_1 and A_2 is the automaton $A_1 \parallel A_2$ such that

1. The locations of $A_1 \parallel A_2$ are the pairs whose first element is a location of A_1 and its second element one of A_2.
2. The invariant of the location (q_1, q_2) is $\mathrm{Inv}(q_1) \wedge \mathrm{Inv}(q_2)$.
3. The initial location is the pair with the initial location of A_1 and the initial location of A_2.
4. The step $(q_1, q_2) \xrightarrow{a,g,r} (q_1', q_2')$ is a transition in the parallel composition if $q_1 \xrightarrow{a,g,r} q_1'$ is a transition in A_1 and $q_2 = q_2'$, or if $q_2 \xrightarrow{a,g,r} q_2'$ is a transition in A_2 and $q_1 = q_1'$. The step $(q_1, q_2) \xrightarrow{\tau, g_1 \wedge g_2, r_1 \cup r_2} (q_1', q_2')$ is a transition in the parallel composition if $q_1 \xrightarrow{a, g_1, r_1} q_1'$ is a transition in A_1 and $q_2 \xrightarrow{b, g_2, r_2} q_2'$ is a transition in A_2, and a and b are complementary actions.

Example 7.5.7 Figure 7.15 shows the automaton `Sender` and the automaton `Sender` $\|$ (`Buffer` $\setminus$ $\{$`receive`$\}$).

Compositionality results for trace inclusion have been proven in several settings, see e.g. [LV96a]. It also holds for automata we consider.

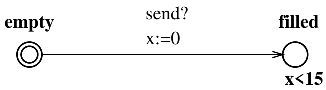

Figure 7.16: Action restriction

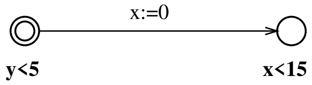

Figure 7.17: $\mathcal{N}(\texttt{Sender}, \texttt{Buffer} \setminus \{\texttt{receive}\})$

Proposition 7.5.8 *Trace inclusion is compositional, that is,*

$$A_1 \sqsubseteq_{\mathrm{TR}} A_2 \implies A_1 \parallel B \sqsubseteq_{\mathrm{TR}} A_2 \parallel B.$$

It is crucial here that the automata we consider do not contain committed locations, urgent channels and shared variables. However, the auxiliary automata that we use to establish trace inclusion within Uppaal do contain committed locations in some cases. This, of course, does not affect the compositionality result. The following proposition is an immediate consequence of the definitions.

Closed systems semantics

Definition 7.5.9 Let C be a set of action names and A be an automaton. Then $A \setminus C$ is the automaton obtained from A by disabling all action with names in C, (i.e. by removing all transitions labeled by an action in $C? \cup C!$.)

Example 7.5.10 The automaton $\texttt{Buffer} \setminus \{\texttt{receive}\}$ is shown in Figure 7.16.

Proposition 7.5.11 $\mathrm{Act}_{A\parallel B} = \mathrm{Act}_A \cup \mathrm{Act}_B$ *and* $\mathrm{Act}_{A\setminus C} = \mathrm{Act}_A \setminus (C? \cup C!)$.

Uppaal interprets a network of automata as a closed system. This means that the semantics $\mathcal{N}(A_1, \ldots, A_n)$ Uppaal assigns to a network consisting of components $A_1, \ldots, A_n$ is given by the automaton

$$\mathcal{N}(A_1, \ldots A_n) = (A_1 \parallel \ldots \parallel A_n) \setminus \mathrm{Names}$$

Example 7.5.12 In Figure 7.17 presents the Uppaal semantics

$$\mathcal{N}(\texttt{Sender}, \texttt{Buffer} \setminus \{\texttt{receive}\})$$

of the network consisting of the components $\texttt{Sender}$ and $\texttt{Buffer} \setminus \{\texttt{receive}\}$.

Thus, the network does not have any visible actions, and therefore its traces only contain time passage actions. Since we are interested in describing sets of traces, we cannot do with closed systems only. With Uppaal we can, of course, only verify closed systems. However, the reachability problems of automata can be expressed in terms of reachability properties of closed systems, by simply adding an automaton to the open system that synchronizes with every visible action. See Appendix A, Section A.1.3 for the details.

Verification of trace inclusion

Within automaton–based verification, it is common practice to describe both the implementation and the specification of a system as automata. Then an automaton A is said to be a (correct) *implementation* of another one B if $A \sqsubseteq_{\mathrm{TR}} B$. [6]

Although it is in general undecidable whether $A \sqsubseteq_{\mathrm{TR}} B$, Alur and Dill [AD94] have shown (in a timed automaton setting without location invariants) that deciding whether or not $A \sqsubseteq_{\mathrm{TR}} B$ can be reduced to reachability checking, provided that B is deterministic. The basic step in this reduction is the construction of an automaton which we call B^{err} here. In our setting, B^{err} is constructed by adding a location $error$ to B and transitions $q \xrightarrow{a\ g} error$ for all locations q and action labels a in such a way that this transition is enabled if no other a–transition is enabled from q. Furthermore, an internal transition from q to $error$ with the guard $\neg\mathrm{Inv}(q)$ is added and all location invariants are removed. The basic result that we need in the verification is that, if the visible actions of A are included in those of B, then

$$A \sqsubseteq_{\mathrm{TR}} B \iff error \text{ is not reachable in } \mathcal{N}(A, B^{err}).$$

The reader is referred to the Appendix A, Section A.1.5 for an elaboration of this.

Moreover, if B is not deterministic, we can try to make it so by renaming its actions and apply the method above. We can use a, what is called, *step refinement* f (or a conjectured one) for this relabeling. To put it very briefly, a step refinement is basically a function from the states of A to the states of B that induces a function from the a–transitions of A to the a–transitions of B. Thus, we can give a transition in A and its image in B the same fresh label and remove all sources of nondeterminism in B. This yields automata A^f and B^f such that

$$A^f \sqsubseteq_{\mathrm{TR}} B^f \implies A \sqsubseteq_{\mathrm{TR}} B,$$

(but not conversely). We refer the reader to Appendix A, Section A.1.5 for the details.

7.5.3 The Enhanced Protocol Model

The enhanced protocol model is presented in Figures 7.18 and 7.19 and are explained below. A major difference between the model in Section 7.4 and the model presented in here lies in the way in which communication between the nodes across the wires is handled. In Section 7.4 this is modeled as the transfer of single messages (PN or CN) that are sent only once, and can be overwritten and lost. As we have ecplained before, 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. An important difference between communication via messages

[6] In fact, coarser notions exist, viz. timed trace and timed trace inclusion, which abstract from certain irrelevant timing aspects that are still present in the traces. Timed trace inclusion has a rather technical definition and trace inclusion implies timed trace inclusion. Therefore, we deal with trace inclusion in the remainder rather than with timed trace inclusion.

and via channels is that one can distinguish two subsequent messages with the same contents, but it is not possible to distinguish two subsequent signals that are equal. Besides driving PN and CN signals, the wire can be left undriven (IDLE). Since the enhanced model presented in this section closely follows the draft IEEE 1394a standard, we believe that our model now adequately reflects the root contention protocol as specified in the IEEE standard. However, we can never formally prove this because the specification is partly informal.

We use the constructions and techniques from Section 7.5.1 to verify this model and we show that – tacitly assuming the basic constraints B_1, B_2 and B_3 – the constraints Eq_1 and Eq_2 are both necessary and sufficient for the correctness of the protocol. Recall that the parameters take values in $\mathbb{R}^{\geq 0}$. These constraints were given beforehand in the informal note [LaF97]. However, Section 7.4 and [Nyu97] also give (different) constraints that ensure protocol correctness. Moreover, we consider in each step of our analysis the constraints needed for the correctness of this step.

This section presents the Uppaal automata modeling RCP. We will see that Equation Eq_1 is needed to ensure that these models faithfully represent the protocol. The next section is concerned with the verification of these models. In the sequel, the term "experimental results" is used for results that have been obtained by checking a number of instances with Uppaal, rather than by a rigorous proof.

The Protocol model automata

Figures 7.18 and 7.19 display the Uppaal templates of the `Wire` and `Node` automata of the enhanced model. These template automata are instantiated to a total system ($\mathtt{Impl}^{\mathtt{B}}$) of two nodes $\mathtt{Node}_1$ and $\mathtt{Node}_2$, connected by bi–directional communication channels ($\mathtt{Wire}_{1,2}$ for messages from $\mathtt{Node}_1$ to $\mathtt{Node}_2$, and $\mathtt{Wire}_{2,1}$ for the opposite directions). We require synchronization between the action that model communication between the nodes and wire, but no synchronization is required for the actions $root$ and $child$. Thus, model of the root contention protocol is given by

$$\mathtt{Impl}^{\mathtt{B}} \equiv (\mathtt{Node}_1 \parallel \mathtt{Wire}_{1,2} \parallel \mathtt{Wire}_{2,1} \parallel \mathtt{Node}_2) \setminus C,$$

where C is the set of actions used to send signals over the wires, that is $C = \{snd\_req_{1,2}$, $snd\_ack_{1,2}$, $snd\_idle_{1,2}$, $snd\_req_{2,1}$, $snd\_ack_{2,1}$, $snd\_idle_{2,1}$, $rec\_req_{1,2}$, $rec\_ack_{1,2}$, $rec\_idle_{1,2}$, $rec\_req_{2,1}$, $rec\_ack_{2,1}$, $rec\_idle_{2,1}\}$.

First of all, the PN message is now called req (parent request), and the CN message ack (acknowledgement). 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 Figure 7.5. The actions 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.

Starting in the root contention location, a node picks a random bit (slow or fast). Simultaneously, a timer x is reset, and an $idle$ message is sent to the `Wire`, which models the removal of the PN signal. Independently, but at about the same time, the other contending `Node` also sends an $idle$, possibly followed by a renewed req. Therefore, the receipt 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 Section 7.4 such that they can now transmit PN (req), CN (ack), and IDLE ($idle$) messages. These messages mark the leading

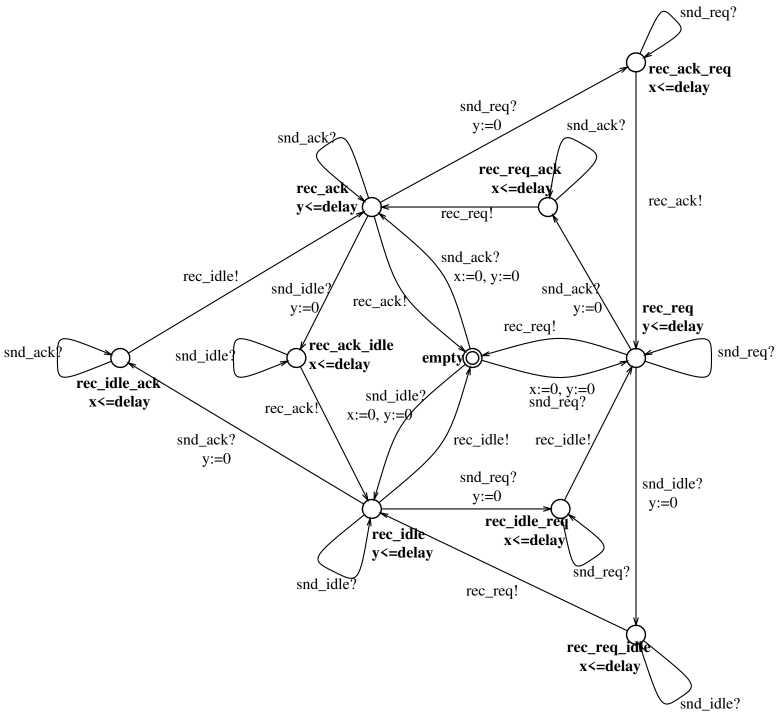

Figure 7.18: The Uppaal Wire automaton template

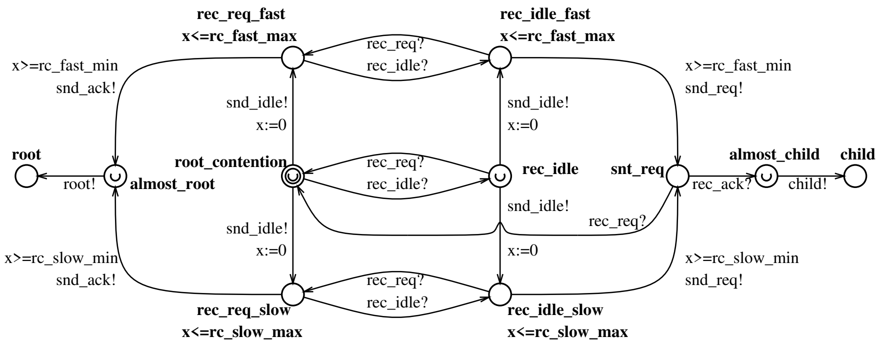

Figure 7.19: The Uppaal Node automaton template

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. The IEEE standard does not specify how many signals can be in the wire simultaneously. However, the following experimental results shows that two–place buffer is necessary and sufficient to model the communication channels. The necessity result (Experimental result 7.5.13) has been established by checking for reachability of the locations where either of the wires contains two signals. The sufficiency result (Experimental result 7.5.14) has been established by checking that no input to the wire occurs if it already contains two messages. Technically, we constructed an automaton $\mathtt{Wire}^{\mathtt{u\_i}}$ by adding locations $unexp\_input$ (unexpected input) to each of the wires and transitions $q \xrightarrow{a} unexp\_input$ for all the locations $q = rec\_ack\_idle$, rec_ack_req, rec_req_ack, rec_idle_req, rec_idle_ack, rec_req_idle and $a = snd\_idle$, snd_req, snd_ack. Then we checked that the location $unexp\_input$ is unreachable indeed. See Appendix A, Section A.1.4.

Experimental result 7.5.13 *For all parameter values, the wire may contain two signals simultaneously at some point in time, that is, one of the locations rec_ack_idle, rec_ack_req, rec_req_ack, rec_idle_req, rec_idle_ack or*
rec_req_idle is reachable in $\mathtt{Impl}^{\mathtt{B}}$.

Experimental result 7.5.14 *The* $unexp\_input$ *locations in* $(\mathtt{Node}_1 \parallel \mathtt{Wire}^{\mathtt{u\_i}}_{1,2} \parallel \mathtt{Wire}^{\mathtt{u\_i}}_{2,1} \parallel \mathtt{Node}_2) \setminus C$ *are unreachable if and only if* Eq_1 *holds.*

The Node automaton is not input enabled, which means that it might block input actions (rec_ack, rec_req or rec_idle) in certain locations by being unable to synchronize. Experimental result 7.5.15, however, states that this never happens in the protocol. That is, provided that Eq_1 holds, no other input can occur than the input specified in Node. This implies equivalence between $\mathtt{Node}^{\mathtt{u\_i}}$ automaton and the Node automaton for parameter values meeting Equation Eq_1. This result has been established by adding a location called $unexp\_input$ to each component and synchronization transitions to this location from all locations in which input (via rec_idle, rec_req and rec_ack) would otherwise be blocked. See Appendix A, Section A.1.4 for a more formal treatment.

Experimental result 7.5.15 *The* $unexp\_input$ *locations in* $(\mathtt{Node}^{\mathtt{u\_i}}_1 \parallel \mathtt{Wire}^{\mathtt{u\_i}}_{1,2} \parallel \mathtt{Wire}^{\mathtt{u\_i}}_{2,1} \parallel \mathtt{Node}^{\mathtt{u\_i}}_2) \setminus C$ *are unreachable if and only if* Eq_1 *holds.*

7.5.4 Verification and Analysis

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 7.20. This automaton is exactly the same as the automaton Spec from the previous section (Section 7.4), it is only expressed in a different syntax and the (trivial) task partition used in the previously to model fairness has been omitted.

We demonstrate that $\mathtt{Impl}^{\mathtt{B}}$ (the parallel composition of the two Node and Wire automata) is a correct implementation of Spec, provided that $\mathtt{Impl}^{\mathtt{B}}$ meets the timing constraint Equations Eq_1 and Eq_2 from Section 7.2.3.

Following the lines in Section 7.4 we do not prove $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{Spec}$ at once but introduce three intermediate automata $\mathtt{I}^{\mathtt{B}}_1$, $\mathtt{I}^{\mathtt{B}}_2$, and $\mathtt{I}^{\mathtt{B}}_3$ in our verification. We use Uppaal and the methods

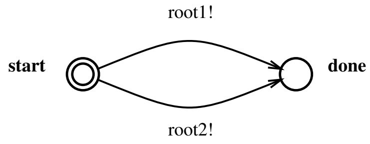

Figure 7.20: The specification automaton of the protocol

described in Section 7.5.2 to derive from numerous instances of the protocol for different parameter values that

$$\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_2^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{Spec}$$

if the parameters meet the timing constraints. Furthermore, we argue that $\mathtt{Impl}^{\mathtt{B}}$ is not a correct implementation of Spec if the parameters do not satisfy the constraints.

Here, $\mathtt{I}_1^{\mathtt{B}}$ is a timed automaton, which abstracts from all message passing in $\mathtt{Impl}^{\mathtt{B}}$ while preserving the timing information of $\mathtt{Impl}^{\mathtt{B}}$. The automaton $\mathtt{I}_2^{\mathtt{B}}$ is obtained from $\mathtt{I}_1^{\mathtt{B}}$ by removing all timing information. In $\mathtt{I}_3^{\mathtt{B}}$ internal choices are further contracted. Notice that the intermediate automata $\mathtt{I}_1^{\mathtt{B}}$, $\mathtt{I}_2^{\mathtt{B}}$ and $\mathtt{I}_3^{\mathtt{B}}$ are obtained from $\mathtt{Impl}^{\mathtt{B}}$ in the same way as $\mathtt{I}_1^{\mathtt{A}}$, $\mathtt{I}_2^{\mathtt{A}}$ and $\mathtt{I}_3^{\mathtt{A}}$ are obtained from $\mathtt{Impl}^{\mathtt{A}}$. In fact, we will see that, when restricted to the reachable states, $\mathtt{I}_2^{\mathtt{B}}$ is basically the automaton $\mathtt{I}_2^{\mathtt{A}}$ where all probabilistic information has been omitted, that $\mathtt{I}_3^{\mathtt{A}}$ is basically $\mathtt{I}_3^{\mathtt{B}}$ where the probabilistic information was left out. but that $\mathtt{I}_1^{\mathtt{B}}$ and $\mathtt{I}_1^{\mathtt{A}}$ differ in their timing behavior. Moreover, remark that, since timing aspects are only present in $\mathtt{Impl}^{\mathtt{B}}$ and $\mathtt{I}_1^{\mathtt{B}}$, the timing constraints only play a role in the first inclusion ($\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$).

The first intermediate automaton

The intermediate automaton $\mathtt{I}_1^{\mathtt{B}}$ is displayed in Figure 7.21. It is a Uppaal equivalent of the timed I/O automaton model from Section 7.4 restricted to the reachable locations. It abstracts from the communication between the nodes and records the status (start, fast, slow, or done) for each of the two nodes. Also, $\mathtt{I}_1^{\mathtt{B}}$ has a clock x to impose timing constraints on 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 $\mathtt{Node}_1$ having picked the fast random bit, and $\mathtt{Node}_2$ still being 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. The invariants on clock x cause both nodes to pick a random bit within a time interval $delay$ after the protocol (re–)start. Also, within an interval $[rc\_fast\_min - delay, rc\_fast\_max]$ or $[rc\_slow\_min - delay, rc\_slow\_max]$, depending on the random bit, either a root is selected ($root_1!$ or $root_2!$) or a restart of the protocol occurs.

The method described in Section 7.5.2 allowed us to establish trace inclusion between $\mathtt{Impl}^{\mathtt{B}}$ and $\mathtt{I}_1^{\mathtt{B}}$. Figure 7.22 describes how unlabeled transitions in $\mathtt{I}_1^{\mathtt{B}}$ and $\mathtt{Impl}^{\mathtt{B}}$ are relabeled, yielding $\mathtt{Impl}^{\mathtt{B}^r}$ and $\mathtt{I}_1^{\mathtt{B}^r}$. (See also Section 7.5.2 and Appendix A, Section A.1.5.) This relabeling of transitions has been constructed from the step refinement from $\mathtt{Impl}^{\mathtt{B}}$ to $\mathtt{I}_1^{\mathtt{B}}$ given in Section 7.4 The transitions relabeled with $retry$ synchronize with an auxiliary automaton

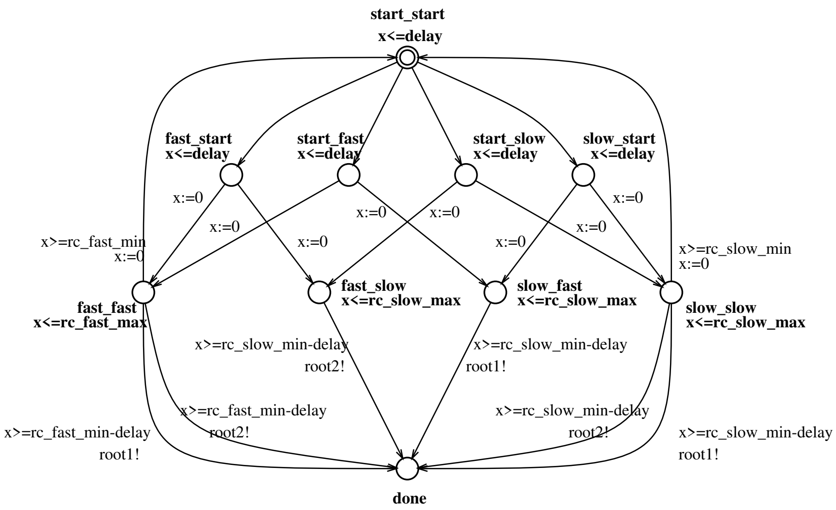

Figure 7.21: The Uppaal $\mathtt{I}_1^{\mathtt{B}}$ automaton of the root contention protocol

called EchoRetry, which takes this action as soon as root contention re–occurs, i.e. as soon as both $\mathtt{Node}_1$ and $\mathtt{Node}_2$ have taken their *snd_req*! transitions to the *snt_req* location. This requires the automata $\mathtt{Node}_\mathtt{i}$, $\mathtt{Wire}_{\mathtt{i,j}}$ and EchoRetry all to synchronize on the action $snd\_req_{i,j}$ and $\mathtt{Node}_\mathtt{i}$, $\mathtt{Wire}_{\mathtt{i,j}}$ and $\mathtt{I}_1^{\mathtt{B}}$ on $snd\_idle_{i\,j}$. We encoded multiway synchronization in Uppaal as described in Section A.1.2.

Manual parameter analysis shows that the error location is unreachable in $\mathtt{Impl}^{\mathtt{B}^r} \parallel \mathtt{I}_1^{\mathtt{B}^r}$, if the constraint *Eq1* and *Eq2* hold. Now, Experimental result 7.5.16 is a direct consequence of Lemma A.1.14.

Experimental result 7.5.16 *If the timing parameters in* $\mathtt{Impl}^{\mathtt{B}}$ *satisfy Equations Eq1 and Eq2, then* $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$.

In order to show the necessity of Equations 2, we need a liveness argument. The key liveness property is that eventually a leader is elected with probability one. Therefore, it is essential that root contention is resolved within the same pass (i.e. without renewed root contention) if both nodes pick different random bits. This is guaranteed by *Eq2*, c.f. Section 7.2.3. Since the probability to pick different random bits is strictly greater than zero in each pass, the nodes will eventually pick different bits, and thus elect a root, with probability one.

Experimental result 7.5.17 *Assume that the two nodes pick different random bits. Then the root contention protocol is resolved within one pass if and only if Equation 2 is satisfied.*

$\mathtt{I}_1^{\mathtt{B}}$	Impl
$start\_start \rightarrow fast\_start$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_fast_1$
$start\_start \rightarrow start\_fast$	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
$start\_start \rightarrow start\_slow$	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_slow_2$
$start\_start \rightarrow slow\_start$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
$start\_fast \rightarrow slow\_fast$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_2} rec\_req\_slow_1$
$slow\_start \rightarrow slow\_fast$	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
$start\_slow \rightarrow slow\_slow$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
$fast\_start \rightarrow fast\_fast$	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
$start\_fast \rightarrow fast\_fast$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_fast_1$
$start\_slow \rightarrow slow\_slow$	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
$fast\_start \rightarrow fast\_slow$	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_slow_2$
$fast\_fast \rightarrow start\_start$	$one\_req \xrightarrow{retry}_{\mathtt{EchoRetry}} start$
$slow\_slow \rightarrow start\_start$	$one\_req \xrightarrow{retry}_{\mathtt{EchoRetry}} start$

Figure 7.22: Relabeling $\mathtt{I}_1^{\mathtt{B}}$ and $\mathtt{Impl}^{\mathtt{B}}$.

The second intermediate automaton

The intermediate automaton $\mathtt{I}_2^{\mathtt{B}}$ is identical to $\mathtt{I}_1^{\mathtt{B}}$, except that all timing information has been removed. Thus, $\mathtt{I}_2^{\mathtt{B}}$ is the exactly same as the automaton $\mathtt{I}_2^{\mathtt{A}}$ from the previous section, if all probabilistic information, the task partition and the unreachable states are omitted from the latter automaton. Since weakening the guards and invariants in an automaton yields an automaton with more traces, we get Proposition 7.5.18, as expected.

Proposition 7.5.18 $\mathtt{I}_1^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_2^{\mathtt{B}}$.

The third intermediate automaton

Figure 7.23 shows intermediate automaton $\mathtt{I}_3^{\mathtt{B}}$, 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. Thus, $\mathtt{I}_3^{\mathtt{B}}$ is the exactly same as $\mathtt{I}_3^{\mathtt{A}}$, if we delete all probabilistic information, the task partition and the unreachable states in the latter automaton.

Since neither $\mathtt{I}_2^{\mathtt{B}}$ nor $\mathtt{I}_3^{\mathtt{B}}$ contains timing information, trace inclusion can be checked with standard methods, see [KS90]. Since we are interested in the applicability of the relabeling method, we use this one for establishing $\mathtt{I}_2^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3^{\mathtt{B}}$. Again, we added labels to certain unlabeled transitions in $\mathtt{I}_2^{\mathtt{B}}$ and $\mathtt{I}_3^{\mathtt{B}}$, to obtain $\mathtt{I}_2^{\mathtt{B}f}$ from $\mathtt{I}_2^{\mathtt{B}}$ and (the deterministic automaton) $\mathtt{I}_3^{\mathtt{B}f}$ from $\mathtt{I}_3^{\mathtt{B}}$. Figure 7.24 lists the corresponding transitions in $\mathtt{I}_3^{\mathtt{B}}$ and $\mathtt{I}_2^{\mathtt{B}}$ that should get the same (fresh) labels after relabeling. Transitions not mentioned the table keep the same label. In particular, the transitions in $\mathtt{I}_2^{\mathtt{B}}$ leaving from $start\_start$ remain unlabeled.

It has been established by Uppaal that $\mathtt{I}_2^{\mathtt{B}f} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3^{\mathtt{B}f}$. Now, Proposition 7.5.19 is an immediate corollary of Lemma A.1.14.

Proposition 7.5.19 $\mathtt{I}_2^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3^{\mathtt{B}}$.

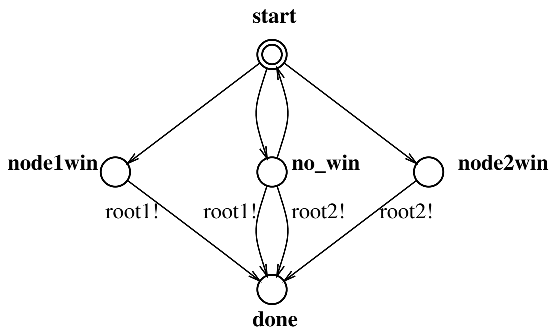

Figure 7.23: The Uppaal $\mathtt{I}_3^{\mathtt{B}}$ automaton of the root contention protocol

$\mathtt{I}_3^{\mathtt{B}}$	$\mathtt{I}_2^{\mathtt{B}}$
$start \xrightarrow{win_2?} node2win$	$start\_fast \xrightarrow{win_1!} slow\_fast$
$start \xrightarrow{win_1?} node1win$	$slow\_start \xrightarrow{win_2!} slow\_fast$
$start \xrightarrow{win_1?} node1win$	$fast\_start \xrightarrow{win_2!} fast\_slow$
$start \xrightarrow{win_2?} node2win$	$start\_slow \xrightarrow{win_1!} slow\_fast$
$start \xrightarrow{no\_win?} no\_win$	$start\_slow \xrightarrow{no\_win!} slow\_slow$
$start \xrightarrow{no\_win?} no\_win$	$start\_slow \xrightarrow{no\_win!} slow\_slow$
$start \xrightarrow{no\_win?} no\_win$	$fast\_start \xrightarrow{no\_win!} slow\_slow$
$start \xrightarrow{no\_win?} no\_win$	$start\_fast \xrightarrow{no\_win!} fast\_fast$

Figure 7.24: Corresponding transitions getting the same label

Since the specification automaton $\mathtt{Spec}$ is deterministic, we only need to check for reachability of the *error* location in the automaton $\mathtt{Spec}^{err}$ to obtain Proposition 7.5.20. (As in the previous case ($\mathtt{I}_2^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3^{\mathtt{B}}$) trace inclusion and timed trace inclusion are the same. But now, because $\mathtt{Spec}$ is deterministic, the method we use to establish timed trace inclusion this is exactly the usual method for establishing trace inclusion.)

Proposition 7.5.20 $\mathtt{I}_3^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{Spec}$.

By transitivity of $\sqsubseteq_{\mathrm{TR}}$ we get that the Equations 1 and 2 are sufficient.

Experimental result 7.5.21 *If Equations 1 and 2 are met by* $\mathtt{Impl}^{\mathtt{B}}$*, then* $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{Spec}$.

Combining the Results 7.5.14, 7.5.15, 7.5.17 and 7.5.21 yields the final conclusion.

Experimental result 7.5.22 *The root contention protocol is correct if and only if the timing parameters satisfy Equations Eq_1 and Eq_2.*

7.5.5 Concluding Remarks

We analyzed a large number of protocol instances with Uppaal. From these experiments, we derived that the constraints which are necessary and sufficient for correct protocol operation are exactly those from [LaF97]. Although these experiments do not ensure correctness, we are quite convinced that the constraints we derived are exactly those required.

Some minor points of incompleteness have been found: The IEEE specification only specifies the propagation delay of signals but not the delay needed to process incoming and outgoing signals. Moreover, the IEEE standard only provides specific values for the timing parameters and not the general parameter constraints, although these give some useful insight in the correctness of the protocol and in restrictions on future applications. We also found some small errors in the informal notes [Nyu97, LaF97].

The fact that the modeling and verification took us a relatively short time illustrates once again that model checkers can be used effectively in the design and evaluation of industrial protocols. Especially the iterative modeling via trial and error is valuable when it comes to understanding the properties of a model. Our case study has added that this also holds in the presence of parameters: once an appropriate model and conjectured timing constraints have been obtained with Uppaal, rigorous parameter analysis could be tackled with another tool or method.

In our case, the very recent work [HRSV01] has given the full evidence of several experimental results in this section. By feeding Eq_1 and Eq_2 together with the basic assumptions B_1, B_2 and B_3 to a prototype parametric extension of Uppaal, the Experimental result 7.5.16 has been established. Due to lack of memory, the necessity of the constraints could only be established partially by the tool.

In our experience, using the current Uppaal2k implementation to establish trace inclusion suffers from several disadvantages. The practical modeling and verification is, as pointed out above, not a problem. However, the proof that the properties we verified indeed established trace inclusion, involved several technicalities. Firstly, due to its closed world interpretation, timed languages cannot be described in Uppaal directly. It is however no conceptual problem to extend Uppaal such that this would be possible. Secondly, the check for language inclusion ($A \sqsubseteq_{\mathrm{TR}} B$, B deterministic) is not implemented in Uppaal. However, we are not aware of any other freely available timed model checker which can do this. This might be remarkable since verification often involves checking language inclusion and it is already known for some time [AD94] how to reduce language inclusion to a reachability problem.

Thirdly, the fact that Uppaal does not support multisynchronization enforces the need of committed locations and this makes the underlying theory more complicated. Relatively small adaptations of Uppaal would overcome these problems and make the verification of trace inclusion a lot easier.

7.6 Third approach: parametric model checking of the Root Contention Protocol

The last verification in this chapter reports on the verification of RCP using the prototype parametric extension of Uppaal from Chapter 6. We analyze the models $\mathtt{Impl}^{\mathtt{A}}$ and $\mathtt{Impl}^{\mathtt{B}}$ from Sections 7.4 and 7.5, where we deal with all five parameters in the system. Unlike the verifications in those sections, we now express the correctness of the protocol by a safety property in Uppaal's logic and find the constraints needed and sufficient to satisfy this property. In addition, we model check the inclusion $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$ parametrically. The purpose of the research presented in this section is not only to prove the correctness of RCP but also to investigate the performance and the usability of the new prototype extension of Uppaal.

When verifying the properties mentioned, we profit from the fact that the parameter constraints have already been found in the preceding sections. So, rather than having the tool generating the constraints, which is a time and memory consuming task, we establish the sufficiency of the constraints by providing these as initial constraints to the tool and show that they imply the property under investigation. Necessity is established by showing that the property does not hold for the negated constraints. As in Chapter 6, significant speed ups are gained by eliminating parameters with the techniques developed in Section 6.4.

All experiments were performed on a 366 MHz Celeron, except the check for $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$, which was performed in a 333 MHz SPARC Ultra Enterprise. The reader is referred to Fig. 7.25 for an overview of results.

7.6.1 Protocol Specification

A crucial safety property for the system is that at most one of the nodes becomes root and at most one of the nodes becomes child. This is expressed by the invariance property Φ below. Note that Φ is similar to the specification automaton `Spec` from the previous sections.

$$\Phi \equiv \forall\Box\ .\ (\neg(\mathrm{Node}_1.\mathrm{root} \wedge \mathrm{Node}_2.\mathrm{root}) \wedge \neg(\mathrm{Node}_1.\mathrm{child} \wedge \mathrm{Node}_2.\mathrm{child}))$$

Of course, the protocol is also required to eventually elect a leader, but here, we will restrict our attention to the safety property.

As we saw in the previous two verifications, there are three parameter equations that play a role in the protocol correctness.

$$delay < rc\_fast\_min. \qquad (Eq_0)$$

$$2 \cdot delay < rc\_fast\_min. \qquad (Eq_1)$$

$$2 \cdot delay < rc\_slow\_min - rc\_fast\_max. \qquad (Eq_2)$$

As before, we assume a basic constraint C_0, given by

$$C_0 \equiv rc\_fast\_min \leq rc\_fast\_max \leq rc\_slow\_min \leq rc\_slow\_max.$$

Note that this is exactly the initial constraint from Section 7.4, but slightly more restrictive than in Section 7.5. Whenever a property is checked with the prototype parametric model checker, C_0 is given as an initial constraint to the tool.

7.6.2 Verification and analysis

Safety for `Impl`$^{\mathtt{A}}$

The manual verification reported on in Section 7.4 implies that the safety property Φ holds for the model $\mathtt{Impl}^{\mathtt{A}}$ if the parameters meet the Equation Eq_0. This has been re–established mechanically with the prototype extension of Uppaal. For this, we provided the equation Eq_0 as an initial constraint, together with the basic constraint C_0, to the prototype tool and then we checked that Φ holds. (Uppaal models of $\mathtt{Impl}^{\mathtt{A}}$ were available from the research reported on in Section 7.4.) Thus, we have the following result.

Proposition 7.6.1 *If* $v \models C_0 \wedge Eq_0$*, then* $[\![\mathcal{A}]\!]_v \models \Phi$.

Moreover, Remark 7.4.14 on page 199, suggests that the Eq_0 is probably not needed for the safety property to hold and that Eq_2 is needed for liveness only, not for safety. This raises the question whether Φ holds for all parameter values. As Φ is an invariance property and this question is just an instance of the universality problem. Since $\mathtt{Impl}^{\mathtt{A}}$ is an L/U automaton ($L = \{rc\_fast\_min, rc\_slow\_min\}, U = \{rc\_fast\_max, rc\_slow\_max, delay\}$), universality problems can be reduced to non–parametric model checking (see Corollary 6.4.6 and the text just above it). This is done by substituting 0 for each parameter in L and ∞ for all parameters in U. Standard Uppaal established the following result.

Proposition 7.6.2 $\mathcal{A}[(0, \infty)] \models \Phi$.

Now Proposition 6.4.3 yields the desired result.

Corollary 7.6.3 *For all parameter valuations* v, $[\![\mathcal{A}]\!]_v \models \Phi$.

Safety for $\mathtt{Impl}^{\mathtt{B}}$

We check the property Φ for $\mathtt{Impl}^{\mathtt{B}}$. From the investigation in Section 7.5, it follows that Φ is satisfied provided that the parameters meet the equations Eq_1 and Eq_2. Several experiments with the prototype for different initial constraints yielded that the property is already satisfied for Eq_0. Recall, however, that Eq_2 is needed for liveness and that Eq_1 guarantees that the two–place buffers are an appropriate model of the communication channel and also that unexpected input signals do not to occur.

We established the following with our prototype model checker.

Proposition 7.6.4 *If* $v \models C_0 \wedge Eq_0$*, then* $[\![\mathtt{Impl}^{\mathtt{B}}]\!]_v \models \Phi$.

Figure 7.25 shows three performance measures for this result. The first numbers, 1.6 minutes and 36 MB, refer to the direct verification of the Proposition 7.6.4 above for the model $\mathtt{Impl}^{\mathtt{B}}$.

The same result can be checked more efficiently (11 seconds, 13 MB) by substituting ∞ for rc_fast_max and for rc_slow_max and substituting rc_fast_min for rc_slow_min. The new model, together with the initial constraint $C_0 \wedge Eq_0$, satisfies the invariant property; As $\mathtt{Impl}^{\mathtt{B}}$ is an L/U automaton, Proposition 6.4.3 now implies 7.6.4.

Further performance improvements can be made by reducing the verification of the property to nonparametric model checking. In order to do so, we consider the model $\mathtt{Impl}^{\mathtt{B}^<}$. In this model, we substitute $rc\_fast\_max = rc\_slow\_max = \infty$ and $delay = rc\_fast\_min = rc\_fast\_max$. The model we thus obtain has no constants and only one parameter. This means we can check any property for this model by substituting $1 = \mathtt{d} = rc\_fast\_min = rc\_fast\_max$. The model is now completely parametric and satisfies Φ. Application of the Propositions 6.4.8, 6.4.12 and6.4.3 imply Proposition 7.6.4.

7.6.3 $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$

Experimental result 7.5.16 from Section 7.5 states that $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$, provided that the timing parameters in $\mathtt{Impl}^{\mathtt{B}}$ satisfy Eq_1 and Eq_2. This was established experimentally by checking the statement for a large number of protocol instances. Here, the inclusion $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$

model	*property*	*initial constr*	*reduced?*	*Uppaal*	*time*	*memory*
$\mathtt{Impl}^{\mathtt{A}}$	Φ	$C_0 \wedge Eq_0$	no	par	2.9 h	185 Mb
$\mathtt{Impl}^{\mathtt{A}}$	Φ	—	yes	std	1 s	800 Kb
$\mathtt{Impl}^{\mathtt{B}}$	Φ	C_0	no	par	1.6 m	36 Mb
$\mathtt{Impl}^{\mathtt{B}}$	Φ	$C_0 \wedge Eq_0$	partly	par	11 s	13 Mb
$\mathtt{Impl}^{\mathtt{B}}$	Φ	—	yes	std	1 s	800 Kb
$\mathtt{Impl}^{\mathtt{B}}$	$\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$	$C_0 \wedge Eq_0 \wedge Eq_1$	no	par	2.6 h	308 Mb

Figure 7.25: Experimental results for the root contention protocol

was reduced to the nonreachability of error states added to $\mathtt{I}_1^{\mathtt{B}}$. The prototype model checker allows to verify this result formally. We take the same models as before. So in particular, we use the same reduction to nonreachability of error states.

Proposition 7.6.5 *If* $v \models C_0 \wedge Eq_1 \wedge Eq_2$, *then* $[\![\mathtt{Impl}^{\mathtt{B}}]\!]_v \sqsubseteq_{\mathrm{TR}} [\![\mathtt{I}_1^{\mathtt{B}}]\!]_v$.

Moreover, we established that the error states mentioned above are reachable in the following cases: if the parameters obey $C_0 \wedge (2 \cdot delay = rc\_fast\_min \wedge Eq_2)$ or if they satisfy $C_0 \wedge Eq_1 \wedge 2 \cdot delay = rc\_slow\_min - rc\_fast\_max$. Recall that the constructions presented in Section 7.5.2 only yield sufficient conditions for trace inclusion. This means that even though error states are reachable, we might still have $\mathtt{Impl}^{\mathtt{B}} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1^{\mathtt{B}}$. Thus, a necessity result for the equations is not given here.

Moreover, $\mathtt{I}_1^{\mathtt{B}}$ falls outside the class of L/U automata, and, therefore, we cannot apply Proposition 6.4.12 to derive a more general result for the equations to be necessary for the unreachability of error states. We conjecture that a more general theory of parametric automata, where parameters are partitioned into lower bound, upper bound and mixed parameters, would be capable of doing so.

7.7 Conclusions

When looking back on the three verification exertions in this chapter, we notice the following points.

Although RCP is not a very complex protocol in itself, many different aspects play a role in its correctness. We tackled probabilistic, real–time, fair and parametric behavior, using three different methods. To keep the analysis process manageable, we separated concerns as much as possible. In doing so, the technique of stepwise abstraction via intermediate automata — not completely unexpectedly — turned out to be very advantageous.

We saw that parametric aspects play a role already in the modeling of the protocol. For some parameter values, the model reflects the IEEE 1394 standard appropriately, for other it does not: the one–place buffers in model $\mathtt{Impl}^{\mathtt{A}}$ are appropriate if Eq_0 holds, the two–place buffers if $\mathtt{Impl}^{\mathtt{B}}$ are so if Eq_1 holds. Moreover, Eq_1 ensures that no other inputs occur than those modeled in $\mathtt{Impl}^{\mathtt{B}}$. The automaton $\mathtt{Impl}^{\mathtt{A}}$ is input–enabled, so all inputs possibly occurring are modeled for sure. Proving that no unexpected inputs occur, as we did for $\mathtt{Impl}^{\mathtt{B}}$, is in fact very similar to proving invariants, which we did for $\mathtt{Impl}^{\mathtt{A}}$. Thus, some of the protocol invariants are encoded in $\mathtt{Impl}^{\mathtt{B}}$, whereas they are properties of $\mathtt{Impl}^{\mathtt{A}}$. The latter may be more elegant, but the input enabled variant of $\mathtt{Impl}^{\mathtt{B}}$ requires many extra transitions, whereas $\texttt{Impl}^{\texttt{B}}$ is a small and easy to understand automaton. In general, this problem would be overcome by adding extra syntactic construct to Uppaal. In the particular case of $\texttt{Impl}^{\texttt{B}}$, the use of variables instead of locations — which can already been done with the current Uppaal version — would already help.

The use of Uppaal to verify trace inclusion was a nice scientific exercise. It showed that trace–based verification with Uppaal is feasible, but also that in order to do it efficiently, an extension of Uppaal with several syntactic constructs would be necessary. This was of course to be expected, because Uppaal is not designed for trace based verification. The strong point of the Uppaal verification was that we could analyze the timing constraints in a quick manner.

As a matter of fact, the experimental verifications with Uppaal were a direct inspiration for the class of lower bound/upper bound automata in Chapter 6: while model checking the protocol for parameter instances, it became clear we could derive firm conclusions for the parametric version of the protocol.

The third verification, where we expressed the protocol specification directly in Uppaal's logic, was more straightforward than the automaton based approach. However, this approach does not support separations of concerns.

Another important remark is that the automata $\texttt{Impl}^{\texttt{A}}$ and $\texttt{Spec}$ are not bisimilar; there is only a one–way simulation from $\texttt{Impl}^{\texttt{A}}$ to $\texttt{Spec}$. This is because $\texttt{Spec}$ can elect process i as a leader with probability one, whereas $\texttt{Impl}^{\texttt{B}}$ can do so with probability $\frac{3}{4}$ at most. The deprobabilized version $\texttt{Impl}^{\texttt{A}^-}$ is, however, bisimilar to $\texttt{Spec}$, provided that the timing actions are hidden. The case for $\texttt{Impl}^{\texttt{B}}$, which is entirely non–probabilistic, is similar. It is bisimilar to $\texttt{Spec}$ if we hide timing actions. If we add probabilities, then there is a simulation from $\texttt{Impl}^{\texttt{B}}$ to $\texttt{Spec}$, but no bisimulation.

This shows that, although bisimulations are useful in many case, they are not sufficient for the analysis of probabilistic algorithms.

Acknowledgements We thank Judi Romijn for her explanation of some subtle points in IEEE 1394, her constructive criticism on early versions of our I/O automata model and other helpful remarks. We also thank the anonymous referees of [SS01] for a remark that has led to a clearer distinction between networks and automata.