Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

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

David P.L. Simons[1], **Mariëlle I.A. Stoelinga**[2] *

[1] Philips Research Laboratories Eindhoven,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
david.simons@philips.com

[2] Computing Science Institute, University of Nijmegen,
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
marielle@cs.kun.nl

Received: date / Revised version: date

Abstract. This paper reports on the mechanical verification of the IEEE 1394 root contention protocol. This is an industrial leader election protocol, in which timing parameters play an essential role.

A manual verification of this protocol using I/O automata has been published in [25]. We improve the communication model from that paper. Using the Uppaal2k tool, we investigate the timing constraints on the parameters which are necessary and sufficient for correct protocol operation: by analyzing large numbers of protocol instances with different parameter values, we derive the required timing constraints.

We explore the use of model checking in combination with stepwise abstraction. That is, we show that the implementation automaton correctly implements the specification via several intermediate automata, using Uppaal to prove the trace inclusion in each step.

1 Introduction

Various recent studies have evidenced the maturity of automated tools for the verification of realistic applications [22,5]. Several case studies are reported in which automatic verification tools are used to analyze IEEE standards. The first and probably best–known example is the verification of the IEEE Futurebus+ standard by Clarke and his students using SMV [21]. This verification revealed several errors that were previously undetected. In [23], using the Caesar/Aldebaran tool set, a deadlock was revealed in the draft IEEE 1394 standard. In this work, we investigate the applicability of the Uppaal2k version [14,19] to analyze the IEEE 1394 root contention protocol [13], a real–time leader election protocol for two processes. We examine to what extent Uppaal can used to do parametric analysis and verification via stepwise abstraction. As far as we know, our case study is the first time that a timed–automata–based tool is used to analyze a part of a (draft) IEEE standard. Given the importance of timing constraints in many of these standards, we believe this to be a significant step forward. Although our analysis did not reveal any errors, we did discover a number of minor points where the standard is incomplete.

Timing parameters play an essential role in the root contention protocol. For certain parameter values, the protocol is correct, and for other values it fails. We are interested in deriving the precise constraints that ensure correctness. There are currently three tools available that (at least in some cases, see [3]) can do parametric analysis of timed systems: HyTech [10], PMC [8] and – very recently – the tool described in [4]. Whereas HyTech and PMC can currently analyse and derive linear parameter constraints, [4] 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. By analyzing numerous instances of the protocol for different values of the parameters, Uppaal allowed us to do an approximate parameter analysis. Uppaal has already been used successfully in various verifications [9,6]. 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.

We would probably have obtained the same results if, in stead of Uppaal, we had used the model checker KRONOS [27], which has been applied successfully in a number of case studies for timed systems, such as [7].

* Research supported by PROGRESS Project TES4199, Verification of Hard and Softly Timed Systems (HaaST).

A manual verification of the root contention protocol has been carried out in [25], where two timing constraints where inferred that ensure protocol correctness. We have mechanically checked the proof invariants in the model from [25] and our Uppaal experiments affirm that all invariants hold under the conditions mentioned in [25]. However, we found informal documents [26] and [18] on the web, that derived (two slightly different) stricter constraints for the protocol. A close examination of the IEEE 1394 specification [13] revealed that the model in [25] is not completely conform the standard and resulted in a new, enhanced protocol model, which, we believe, does carefully reflect the IEEE specification. The main difference between the models from [25] and the present work is the way in which the communication between the processes is modeled: by a package mechanism in the former versus by continuous–time signals in the latter.

Using the enhanced model, we investigate the correct operation of the root contention protocol with Uppaal. The constraints that we deduce by approximate parameter analysis are exactly those from [18]. Since the probabilistic phenomena in [25] and in this work are basically the same, we do not reconsider probabilistic aspects of the protocol here.

Unlike most other Uppaal case studies, we carry out the verification using stepwise abstraction, similarly to [25]. This method allows for separation of concerns and is common practice in automaton–based verification, but not in the context of model checkers. In this method, one relates the implementation and the specification automaton via trace inclusion, using several intermediate automata. In order to do so, we need several constructions on Uppaal models. The work [16,15] discusses a different approach to this problem. The contributions of this work over [25] consist of: (1) a realistic model of the communication within the protocol, (2) the investigation of timing constraints with Uppaal, (3) the application of stepwise abstraction together with Uppaal, and several constructions on Uppaal models to verify trace inclusion (in certain cases) with Uppaal.

The modeling and verification effectively took us approximately one month. Most of this time has been spent in understanding the IEEE 1394 standard and in several prototype–model improvements. Some time could have been saved by automating tasks, such as certain syntactic operations on the automaton models and repeated verification of the same properties for different parameter values.

The rest of this paper is organized as follows. Section 2 informally describes the root contention protocol. In Section 3, we describe how Uppaal can be used in verifying system correctness via stepwise abstraction. Section 4 presents the new protocol model and contains the associated verification results. Finally, in Section 5, conclusions are given.

2 The IEEE 1394a Root Contention Protocol

The IEEE 1394 standard [12] (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 camera's. It 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.

The IEEE standard provides a layered, OSI–style description, defining four protocol layers. The root contention protocol is part of Tree–identify phase, present in the lowest, physical layer (PHY). This layer provides the electrical and mechanical interface for data transmission across the bus. Furthermore, it handles bus configuration, arbitration, and data transmission. At this moment, the IEEE 1394a supplement [13] to the standard is the latest approved standard that concerns root contention and includes several clarifications, extensions, and performance improvements over earlier standards.

A 1394 network consist of several nodes (devices), having one or more ports. Each port may be connected to one other node's port, via a bi–directional cable. Nodes should be connected in a tree–like network topology, without cycles. First, bus configuration is performed. This is done automatically upon a bus reset: after power up and after device addition or removal. Bus configuration proceeds in three phases. It starts with bus initialization. This is followed by the Tree–identify phase (Tree ID). 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 (Self ID), 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.

The Tree ID phase works as follows. 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 neighbor node, or the node can leave the line undriven (IDLE). The PN signal is to ask the receiving node to become parent (connecting closer to the root) of the sending node (then connecting further away from the root) and is acknowledged by a CN signal. The receipt of a CN signal on a port in its turn, is 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 other ports. Thus, initially, only the leaf nodes 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 ID, 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 contention protocol is initiated to elect one of the two nodes as the root of the tree.

2.1 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. Note that root contention is detected by each of the two contending nodes ($\mathtt{Node_1}$ and $\mathtt{Node_2}$) individually. Upon detection of root contention, a node backs off by removing the PN signal, and 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, whereas, if the random bit is zero, it will wait for a shorter time ROOT_CONTEND_FAST. Figure 1 lists the wait times as specified in the latest draft version of the IEEE 1394a standard [13].

When its timer expires, a node samples its contention port once again. If it sees IDLE, then it starts sending 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 will send the CN signal back as an acknowledgement and becomes the root.

If both nodes pick different random bits, then the slowest (picking 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 is the case, then the slower node becomes the root. Secondly, if the nodes with the same speed, then there is a chance of root contention again: each node may see an IDLE signal when its timer expires and they both start sending PN signals. In this case, both nodes will detect renewed root contention and the whole process is repeated until one of them becomes root. Eventually (with probability one), both nodes will pick different random bits, in which case root contention certainly is resolved.

2.2 Protocol Timing Constraints and their Implications

The timing parameters that are used in the protocol include the wait times as listed above, and the *delay* parameter, which corresponds to the maximal total time from sending a signal by one node to receiving it by the other node. It includes the cable propagation delay, and the time to process the cable line states by the hardware and software layers at the ports of the two nodes. For the protocol to work correctly, two constraints on these timing parameters are essential (Equations 1 and 2).

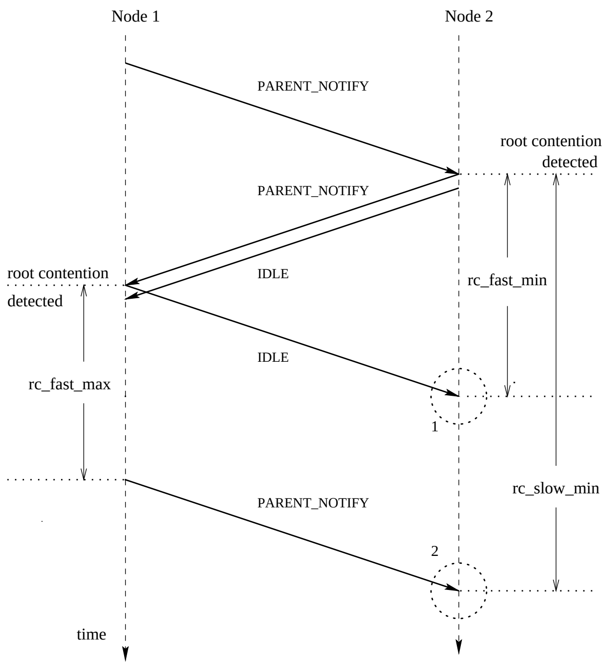

Fig. 2. Visualization of the protocol timing constraints

$$2 * delay < rc\_fast\_min. \tag{1}$$

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

Throughout the verification, we assume three basic assumptions

$$0 \leq rc\_slow\_min \leq rc\_slow\_max, \tag{B1}$$

$$0 \leq rc\_fast\_min \leq rc\_slow\_min \text{ and} \tag{B2}$$

$$rc\_fast\_min \leq rc\_fast\_max. \tag{B3}$$

We do not assume $rc\_fast\_max \leq rc\_slow\_max$ beforehand, but note that this follows from Constraint 2. The origin of these equations is visualized in Figure 2 and explained below.

Ad Equation 1: In case of Node 2 selecting the short waiting period, constraint Equation 1 ensures that the IDLE signal from Node 1 arrives at Node 2 before the waiting period of Node 2 ends (See circle 1 in Figure 2). 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 makes itself root. When Node 1 ends its waiting period, however, it will see the IDLE signal from Node 2, as if nothing happened, and send a PN to Node 2. Awaiting the response it will time out, which leads to a bus reset. Therefore, constraint Equation 1 is required for correct protocol operation.

Ad Equation 2: This constraint ensures that root contention is always resolved in case of one node (say Node

	minimum	maximum
ROOT_CONTEND_FAST	*rc_fast_min*(760 ns)	*rc_fast_max*(850 ns)
ROOT_CONTEND_SLOW	*rc_slow_min*(1590 ns)	*rc_slow_max*(1670 ns)

Fig. 1. Root contend wait times from IEEE 1394a

1) selecting the short waiting period and the other (Node 2) selecting the long waiting period. More precisely, constraint Equation 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 2). Otherwise, Node 2 might still see the IDLE signal from Node 1, and start sending a new 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 timers. Therefore, this equation ensures that renewed root contention can only occur for if both nodes pick equal random bits.

This analysis above is based on informal notes [18, 26] to the IEEE P1394a Working Group. The equations from [18] match ours, whereas [26] incorrectly cites [18] and contains some errors. In the model of [25], the above scenario cannot be mimicked, due to an imperfection in its model, and weaker constraint equations are found.

Note that these timing constraints do not appear in the IEEE 1394 specification [12,13]. However, the root contention wait times from the specification (see Figure 1) meet these constraints. For these values of the parameters, the timing constraint Equation 2, (which implies Equation 1 for these parameter values) implies $delay < 370$ ns.

The cable propagation delay is specified to be less or equal than 5.05 ns/m. Unfortunately, any additional processing delays are not explicitly specified in the standard. If we disregard such extra delays, then a maximum cable length between two nodes of $\frac{370\text{ ns}}{5.05\text{ ns/m}} \approx 73$ m is allowed. In the IEEE 1394a standard the cable length is, at the moment, limited to 4.5 m by the worst case round trip propagation delay during bus arbitration. However, the above timing constraints require the root contention times to be longer, when cable lengths increase significantly. This may be disadvantageous to future applications, and alternative root contention protocols may become necessary.

On the other hand, if processing delays are significant, the current maximum cable length allows for a maximal processing delay of $\frac{370-4.5\cdot 5.05}{2}$ ns ≈ 173 ns on each side of the wire.

3 Verification of trace inclusion with Uppaal

Uppaal [14,19] 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. If the property is not satisfied, a diagnostic trace is provided showing how the property is violated.

Section 3.1 describes how a real–time system can be modeled in Uppaal and which properties can be checked automatically. Section 3.2 gives the semantics of a Uppaal model and Section 3.3 tells how Uppaal can be used to verify *trace inclusion*, *i.e.* that one model in Uppaal is a correct implementation of another one.

3.1 Model checking with Uppaal

This section gives an informal introduction to Uppaal and is based on [19]. A system in Uppaal is modeled as a network of nondeterministic processes 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. Thus, nondeterministic choices can be specified within Uppaal. Note that time can only elapse at the locations (conform invariants). Transitions are taken instantaneously, i.e. no time elapses during transitions. Three special types of locations are available:

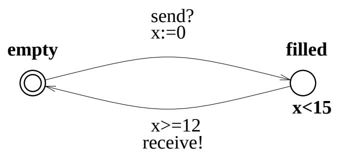

Fig. 3. An example Uppaal process automaton (`Buffer`)

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.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 features in our verification, we do not describe them here.

Consider the Buffer process automaton displayed in Figure 3. 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 (= initial location where all clocks are 0) and in which the property p holds. Dually, $\mathrm{A}\Box p$ denotes that p holds for all reachable states, *i.e.* that p is an invariant of the automaton.

This logic is sufficient to specify reachability properties, invariants, and bounded liveness properties. For the latter, see [1]. However, general liveness and fairness properties, *e.g.* whether an event occurs infinitely often, cannot be expressed in Uppaal.

3.1.1 Notational conventions

Throughout this paper we 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, 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 $\mathrm{Act} = \mathrm{Names}? \cup \mathrm{Names}! \cup \{\tau\}$. 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. See also Appendix B for a glossary of symbols.

3.2 The semantics of Uppaal models

In our verification, we express the behaviour 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 $\backslash$ 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.

Now, we give the open system semantics of an automaton A by its underlying *timed labeled transition system* (TLTS). We may assume that A consists of one component, because we give the semantics of the parallel composition later in this section.

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. 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 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 automaton 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. The initial state of the TLTS is given by the initial locations in the automaton and the clock valuations that assign 0 to each variable.

Example 1. The TLTS underlying the system in Figure 3 consists of the states (empty, u) and (filled, u) for $u \geq 0$ (where u denotes the valuation that assigns the value u to the clock x), the initial state $(\mathit{empty}, 0)$ and the transitions

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

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

Definition 1. 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 ending in that state.
4. A *trace* (of either an automaton or a TLTS) arises from a execution by omitting the states and internal actions. The sets of traces of a TLTS or an automaton A are both denoted by tr_A. We write $A \sqsubseteq_{\mathrm{TR}} B$ if $\mathrm{tr}_A \subseteq \mathrm{tr}_B$.

Example 2. 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.

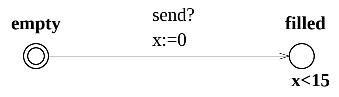

Fig. 4. Action restriction

The following theorem expresses the main result by Alur and Dill [2], upon which Uppaal and other model checkers based on timed automata are built.

Theorem 1. *The set of reachable states of a timed 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 2. 1. A TLTS is called *deterministic* if there are no τ labeled transitions and for every state s and every action $a \in \mathrm{Act}$, there is at most one state t such that $s \xrightarrow{a} t$.
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.

Definition 3. 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 3. The automaton `Buffer`$\setminus\{$`receive`$\}$ is shown in Figure 4.

The parallel composition operator we consider is basically the one from CCS. 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 synchronising on complementary actions $a?$ and $a!$. Synchronisation yields an invisible action in the composition (and hence no other components can synchronise with the same action).

Definition 4. 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 $\mathrm{Inv}(q_1) \wedge \mathrm{Inv}(q_2)$.

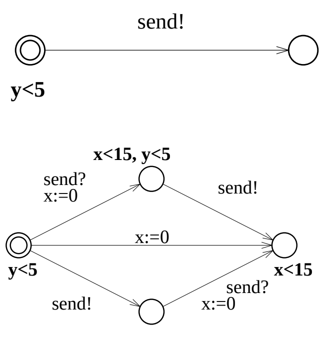

Fig. 5. `Sender` and `Sender` || `Buffer` \ {`receive`}

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 4. Figure 5 shows the automaton `Sender` and the automaton `Sender` || (`Buffer` \ {`receive`}).

Compositionality results for trace inclusion have been proven in several settings, see e.g. [20]. It also holds for automata we consider.

Proposition 1. *Trace inclusion is compositional i.e.*

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

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.

Proposition 2. $\mathrm{Act}_{A\parallel B} = \mathrm{Act}_A \cup \mathrm{Act}_B$ *and* $\mathrm{Act}_{A\setminus C} = \mathrm{Act}_A \setminus (C? \cup C!)$.

Uppaal interprets networks of automata as closed systems. Thus, the semantics $\mathcal{N}(A_1, \ldots A_n)$ Uppaal assigns to 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 5. In Figure 6 presents the the Uppaal semantics $\mathcal{N}$(`Sender`, `Buffer` \ {`receive`}) of the network constiting of the components `Sender` and `Buffer`\{`receive`}.

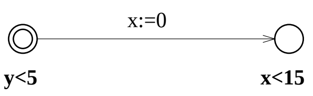

Fig. 6. $\mathcal{N}$(`Sender`, `Buffer` \ {`receive`})

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 synchronises with every visible action. See Appendix A, Section A.3 for the details.

3.3 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 $A \sqsubseteq_{\mathrm{TR}} B$. [1]

We assume that the visible actions of A are included in those of B. Although it is in general undecidable whether $A \sqsubseteq_{\mathrm{TR}} B$, Alur and Dill [2] have shown (in a timed automaton setting without location invariants) that deciding whether $A \sqsubseteq_{\mathrm{TR}} B$ 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 is that we need in the verification is that

$$A \sqsubseteq_{\mathrm{TR}} B \iff \textit{error} \text{ is not reachable in } A \parallel B^{err}.$$

The reader is referred to the Appendix A, Section A.4.1 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

[1] In fact, a 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.

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.4.2 for the details.

4 The Enhanced Protocol Model

A manual verification of the root contention protocol is described in [25]. However, a major difference between the model in [25] and the enhanced model presented in this work lies in the way in which communication between the nodes across the wires is handled. In [25], this is modeled as the transfer of single messages (PN or CN) that are sent only once, and can be overwritten and lost. This abstraction is inappropriate, since in IEEE 1394 communication is done via signals continuously being driven across the wire. These signals persist at the input port of the receiving node, until the sending node changes its output port signal. An important difference between communication via messages 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 paper 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 3 to verify this model and we show that – tacitly assuming the basic constraints B1, B2 and B3 – the constraints 1 and 2 are both necessary and sufficient for the correctness of the protocol. These constraints were given beforehand in the informal note [18]. However, the works [25] and [26] also claimed to 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.

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.

4.1 The Protocol Model Automata

Figures 7 and 8 display the Uppaal templates of the `Wire` and `Node` automata of the enhanced model. These template automata are instantiated to a total system (`Impl`) 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 synchronisation between the action that model communication between the nodes and wire, but no synchoronisation is required between *root* and *child*. Thus, model of the root contention protocol is given by

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

where C is the set of actions used to send signals over the wire, that is $C = \{\mathit{snd\_req}_{1,2}, \mathit{snd\_ack}_{1,2}, \mathit{snd\_idle}_{1,2}, \mathit{snd\_req}_{2,1}, \mathit{snd\_ack}_{2,1}, \mathit{snd\_idle}_{2,1}, \mathit{rec\_req}_{1,2}, \mathit{rec\_ack}_{1,2}, \mathit{rec\_idle}_{1,2}, \mathit{rec\_req}_{2,1}, \mathit{rec\_ack}_{2,1}, \mathit{rec\_idle}_{2,1}\}$. The Uppaal model files can be found at our web site [24].

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 1. 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 [25], such that they can now transmit PN (*req*), CN (*ack*), and IDLE (*idle*) messages. These messages mark the leading edge of a new signal being driven across the wire. Until a new message arrives, signals continue to be driven across the wire. Furthermore, the wires now comprise a two–place buffer, such that two messages at the same time can travel across a wire. 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 1) has been established by checking for reachability of the locations where either of the wires contains two signals. The sufficiency result (Experimental result 2) 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 $q \xrightarrow{a} \mathit{unexp\_input}$ for all the locations $q = \mathit{rec\_ack\_idle}$, $\mathit{rec\_ack\_req}$, $\mathit{rec\_req\_ack}$,

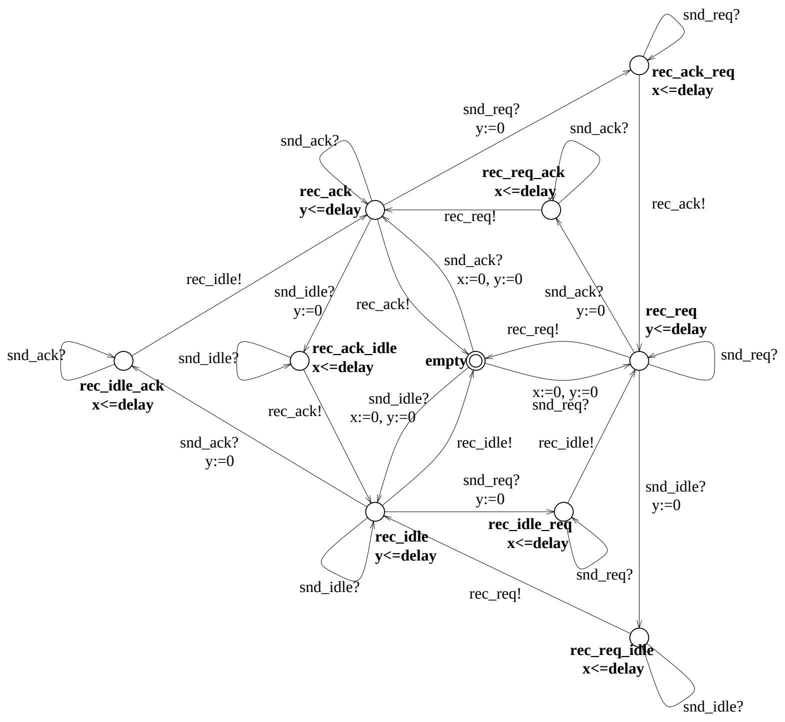

Fig. 7. The Uppaal Wire automaton template

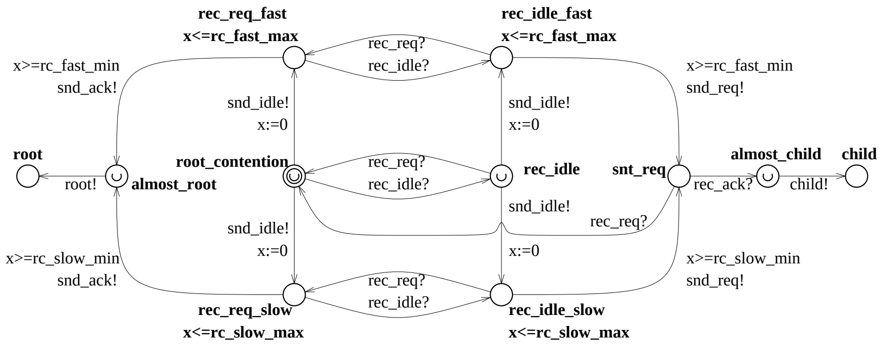

Fig. 8. The Uppaal Node automaton template

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.3.1.

Experimental result 1. *For all parameter values, the wire may contain two signals simultaneously at some point in time, i.e. 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* `Impl`.

Experimental result 2. *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 Equation 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 3 however states that this never happens in the protocol, *i.e.* that no other input can occur than the input specified in `Node` provided that Equation 1 holds. This implies equivalence between $\mathtt{Node}^{\mathtt{u\_i}}$ automaton and the `Node` automaton for param-

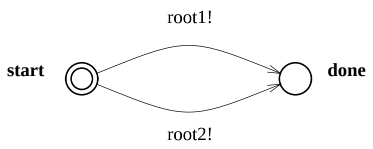

Fig. 9. The specification automaton of the protocol

eter values meeting Equation 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.3.1 for a more formal treatment.

Experimental result 3. *The unexp_input locations in* $(\mathtt{Node}_1^{\mathtt{u-i}} \parallel \mathtt{Wire}_{1,2}^{\mathtt{u-i}} \parallel \mathtt{Wire}_{2,1}^{\mathtt{u-i}} \parallel \mathtt{Node}_2^{\mathtt{u-i}}) \setminus C$ *are unreachable if and only if Equation 1 holds.*

4.2 Verification of the Protocol

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 9. We demonstrate that Impl (the parallel composition of the two Node and Wire automata) is a correct implementation of Spec, provided that Impl meets the timing constraint Equations 1 and 2 from Section 2.2.

Following the lines in [25], we do not prove $\mathtt{Impl} \sqsubseteq_{\mathrm{TR}}$ Spec at once but introduce three intermediate automata $\mathtt{I}_1$, $\mathtt{I}_2$, and $\mathtt{I}_3$ in our verification. We use Uppaal and the methods described in Section 3.2 to derive from numerous instances of the protocol for different parameter values that

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

if the parameters meet the timing constraints. Furthermore, we argue that Impl is not a correct implementation of Spec if the parameters do not satisfy the constraints.

The method of introducing intermediate automata in a correctness proof is called *stepwise abstraction.* It is a widely used method in automaton–based verification, because it allows among other things for separation of concerns.

Here, $\mathtt{I}_1$ is a timed automaton, which abstracts from all message passing in Impl while preserving the timing information of Impl. The automaton $\mathtt{I}_2$ is obtained from $\mathtt{I}_1$ by removing all timing information. In $\mathtt{I}_3$ internal choices are further contracted. Since timing aspects are only present in Impl and $\mathtt{I}_1$, the timing parameters only play a role in the first inclusion ($\mathtt{Impl} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1$).

4.3 The First Intermediate Automaton

The intermediate automaton $\mathtt{I}_1$ is displayed in Figure 10. It is a Uppaal equivalent of the timed I/O automaton model from [25], 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, $\mathtt{I}_1$ 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 $[\mathit{rc\_fast\_min} - \mathit{delay}, \mathit{rc\_fast\_max}]$ or $[\mathit{rc\_slow\_min} - \mathit{delay}, \mathit{rc\_slow\_max}]$, depending on the random bit, either a root is selected ($\mathit{root}_1!$ or $\mathit{root}_2!$) or a restart of the protocol occurs.

The method described in Section 3.3 allowed us to establish trace inclusion between Impl and $\mathtt{I}_1$. Figure 11 describes how unlabeled transitions in $\mathtt{I}_1$ and Impl are relabeled, yielding $\mathtt{Impl}^r$ and $\mathtt{I_1}^r$. (See also Section 3.3 and Appendix A, Section A.4.2.) This relabeling of transitions has been constructed from the step refinement from Impl to $\mathtt{I}_1$ given in [25]. The transitions relabeled with *retry* synchronize with an auxiliary automaton 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 $\mathit{snd\_req}_{i,j}$ and $\mathtt{Node}_\mathtt{i}$, $\mathtt{Wire}_{\mathtt{i,j}}$ and $\mathtt{I}_1$ on $\mathit{snd\_idle}_{ij}$. We encoded multiway synchronization in Uppaal as described in Section A.2.

Manual parameter analysis shows that the error location is unreachable in $\mathtt{Impl}^r \parallel \mathtt{I_1}^r$, if the constraint Equations 1 and 2 hold. Now, Experimental result 4 is a direct consequence of Lemma 3.

Experimental result 4. *If the timing parameters in* Impl *satisfy Equations 1 and 2, then* $\mathtt{Impl} \sqsubseteq_{\mathrm{TR}} \mathtt{I}_1$.

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 Equation 2, *c.f.* Section 2.2. 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.

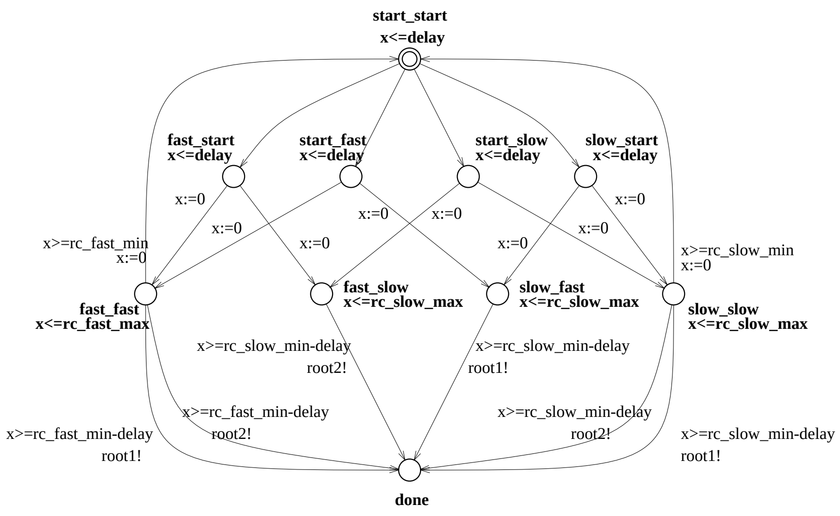

Fig. 10. The Uppaal $\mathtt{I}_1$ automaton of the root contention protocol

$\mathtt{I}_1$	Impl
start_start $\longrightarrow$ *fast_start*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_fast_1$
start_start $\longrightarrow$ *start_fast*	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
start_start $\longrightarrow$ *start_slow*	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_slow_2$
start_start $\longrightarrow$ *slow_start*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
start_fast $\longrightarrow$ *slow_fast*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_2} rec\_req\_slow_1$
slow_start $\longrightarrow$ *slow_fast*	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
start_slow $\longrightarrow$ *slow_slow*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
fast_start $\longrightarrow$ *fast_fast*	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_fast_2$
start_fast $\longrightarrow$ *fast_fast*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_fast_1$
start_slow $\longrightarrow$ *slow_slow*	$root\_contention_1 \xrightarrow{snd\_idle12}_{\mathtt{Node}_1} rec\_req\_slow_1$
fast_start $\longrightarrow$ *fast_slow*	$root\_contention_2 \xrightarrow{snd\_idle21}_{\mathtt{Node}_2} rec\_req\_slow_2$
fast_fast $\longrightarrow$ *start_start*	$one\_req \xrightarrow{retry}_{\mathtt{EchoRetry}} start$
slow_slow $\longrightarrow$ *start_start*	$one\_req \xrightarrow{retry}_{\mathtt{EchoRetry}} start$

Fig. 11. Relabeling $\mathtt{I}_1$ and `Impl`.

Experimental result 5. *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.*

4.4 The Second Intermediate Automaton

The intermediate automaton $\mathtt{I}_2$ is identical to $\mathtt{I}_1$, except that all timing information has been removed. Since weakening the guards and invariants in an automaton yields an automaton with more traces, we get Proposition 3, as expected.

Proposition 3. $\mathtt{I}_1 \sqsubseteq_{\mathrm{TR}} \mathtt{I}_2$.

4.5 The Third Intermediate Automaton

Figure 12 shows intermediate automaton $\mathtt{I}_3$, in which internal choices have been further contracted. Selection of the two random bits is no longer represented via separate, subsequent transitions, but done at once via a single transition.

Since neither $\mathtt{I}_2$ nor $\mathtt{I}_3$ contains timing information, trace inclusion can be checked with standard methods, see [17]. Since we are interested in the applicability of the relabeling method, we use this one for establishing $\mathtt{I}_2 \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3$. Again, we added labels to certain unlabeled transitions in $\mathtt{I}_2$ and $\mathtt{I}_3$, to obtain ${\mathtt{I}_2}^f$ from $\mathtt{I}_2$ and (the deterministic automaton) ${\mathtt{I}_3}^f$ from $\mathtt{I}_3$. Figure 13 lists the corresponding transitions in $\mathtt{I}_3$ and $\mathtt{I}_2$ that should

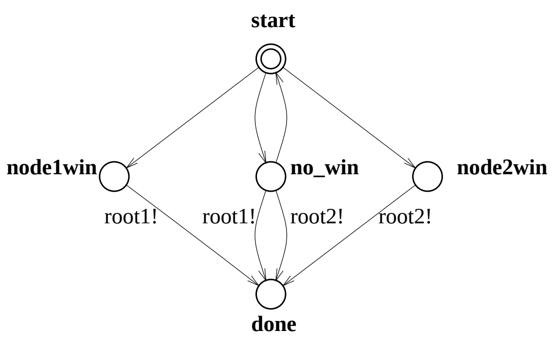

Fig. 12. The Uppaal $\mathtt{I}_3$ automaton of the root contention protocol

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

Fig. 13. Corresponding transitions getting the same label

get the same (fresh) labels after relabeling. Transitions not mentioned the table keep the same label. In particular, the transitions in $\mathtt{I}_2$ leaving from *start_start* remain unlabeled.

It has been established by Uppaal that $\mathtt{I}_2{}^f \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3{}^f$. Now, Proposition 4 is an immediate corollary of Lemma 3.

Proposition 4. $\mathtt{I}_2 \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3$.

Since the specification automaton Spec is deterministic, we only need to check for reachability of the *error* location in the automaton $\mathtt{Spec}^{err}$ to obtain Proposition 5. (As in the previous case ($\mathtt{I}_2 \sqsubseteq_{\mathrm{TR}} \mathtt{I}_3$) trace inclusion and timed trace inclusion are the same. But now, because Spec is deterministic, the method we use to establish timed trace inclusion this is exactly the usual method for establishing trace inclusion.)

Proposition 5. $\mathtt{I}_3 \sqsubseteq_{\mathrm{TR}} \mathtt{Spec}$.

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

Experimental result 6. *If Equations 1 and 2 are met by* Impl*, then* Impl $\sqsubseteq_{\mathrm{TR}}$ Spec.

Combining the Results 2, 3, 5 and 6 yields the final conclusion.

Experimental result 7. *The root contention protocol is correct if and only if the timing parameters satisfy Equations 1 and 2.*

5 Conclusions

This paper reports a mechanical verification of the IEEE 1394 root contention protocol. This is an industrial protocol in which timing parameters play an essential role.

In this case study, we used the Uppaal2k tool and stepwise verification of trace inclusion to investigate the timing constraints of the protocol. We analysed 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 [18]. Although these experiments do not ensure correctness, we are 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 [26, 18].

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 [11] has given the full evidence of several experimental results in this paper. By feeding Equations 1 and 2 and the basic assumptions $B1$, $B2$ and $B3$ to a prototype parametric extension of Uppaal, the Experimental result 4 has been established. Due to lack of memory, the necessity of the constraints could only be established partially by the tool.

We experienced that 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 it is already known for some time [2] how to reduce language inclusion to a reachability problem.

Thirdly, the fact that Uppaal does not support multi-synchronisation 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.

Acknowledgements The authors like to thank Judi Romijn for a helpful discussion and the anonymous referees of this journal for a remark that has led to a clearer distinction between networks and automata.

A Appendix

A.1 Notational conventions

Besides the conventions adopted in the paper (Section 3.1.1), we find it convenient to use general boolean expressions in guards and invariants, whereas Uppaal only allows the use of a conjunction in these. Therefore, we use $q \xrightarrow{a,g \vee g',r} q'$ as an abbreviation for the two transitions $q \xrightarrow{a,g,r} q'$ and $q \xrightarrow{a,g',r} q'$; the guard $g \implies g'$ stands as an abbreviation for $\neg g \vee g'$; $\neg g$ stands as an abbreviation for the guard obtained by replacing every $>$ in g by $\leq$, $\geq$ by $<$, $\geq$ by $<$, $\leq$ by $<$ and $\wedge$ by $\vee$ and $\vee$ by $\wedge$.

A.2 Encoding multi-way synchronization in Uppaal

As explained before, Uppaal only provides binary synchronization. We can use the concept of committed location and renaming to enforce synchronization between more than two action labels. If we want to have an $a!$-action in A_0 synchronize with n $a?$-actions in $A_1 \ldots A_n$ ($n > 1$), then the idea is as follows. Relabel a in A_i into – a fresh label – a_i, $i \geq 0$. Whenever A_0 performs an $a_0!$ action, an auxiliary automaton 'catches' it and 'distributes' it over the other automata. More precisely, the auxiliary automaton synchronizes on $a_0!$ and enforces – sequentially but without delay nor interruption – synchronization with $a_1? \ldots a_n?$ in $A_1 \ldots A_n$ respectively.

Example 6. Consider the automata in Figure 14. Being in their initial locations, either A_2 or A_3 takes an $a?$-action to synchronize with the $a!$-action in A_1. Synchronization of the three automata on $a!$, $a?$ and $a?$, can be mimicked by introducing an auxiliary automaton and renaming of actions, see Figures 15 and 16. It is also possible to integrate the auxiliary automaton within the other automata.

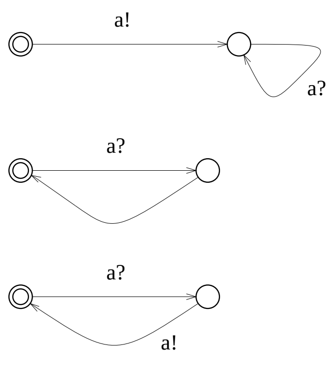

Fig. 14. Three automata: A_1, A_2, A_3

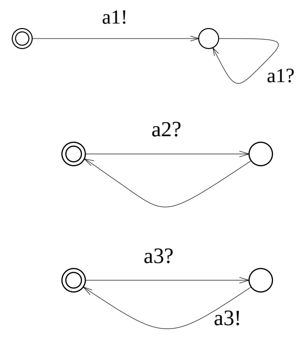

Fig. 15. Encoding multisynchronization with binary synchronization

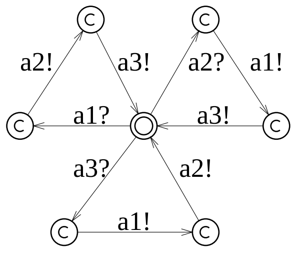

Fig. 16. Auxilliary automaton

A.3 Reducing reachability properties of automata to reachability properties of networks

Definition 5. Let C be a set of action names. Define the automaton Snc_C as the automaton with one location q_C and transitions $q_C \xrightarrow{a} q_C$ for every a in $C? \cup C!$. For an automaton A, define $Snc_A = Snc_C$, where C are the action names occuring in A, and write q_A for q_C.

The following result allows us to check the reachability properties of an automaton A via the reachability properties a network $\mathcal{N}(A \parallel Snc_A)$, which can be checked by Uppaal. Its proof is straightforward.

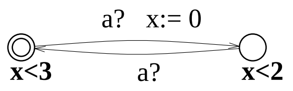

Fig. 17. An input enabled automaton

Proposition 6. *Let q be a location and v a clock valuation of A. Then (q, v) is reachable in A if and only if $((q, q_A), v)$ is reachable in $(A \parallel Snc_A) \setminus$ Names.*

If A is given in terms of restriction and parallel composition over the components $A_1, A_2, \ldots, A_n$, then we like to carry out the construction with Snc_A above on A's components, rather than first computing A explicitly and then performing the construction.

In our verification, A is given by $(A_1 \parallel A_2) \setminus C$ and we have checked its reachability properties via the network $\mathcal{N}(A_1, A_2, Snc_{C'})$. This is justified by the following argument, where C is a set of action names, $C' = \text{Names} \setminus C$, $s = (q, v)$ is a state of A and $\overline{s} = ((q, q_{C'}), v)$.

$$\begin{aligned}
&\overline{s} \text{ reachable in } \mathcal{N}(A_1, A_2, Snc_{C'}) \iff \\
&\overline{s} \text{ reachable in } (A_1 \parallel A_2 \parallel Snc_{C'}) \setminus \text{Names} \iff \\
&\overline{s} \text{ reachable in } (A_1 \parallel A_2 \parallel Snc_{C'}) \setminus C \cup C' \iff \\
&\overline{s} \text{ reachable in } ((A_1 \parallel A_2) \setminus C) \parallel Snc_{C'}) \setminus C' \iff \\
&s \text{ reachable in } (A_1 \parallel A_2) \setminus C.
\end{aligned}$$

A.3.1 Input enabling in Uppaal

The *input actions* of an automaton A are the actions of the form $a?$. We call A *input enabled* if synchronization on input actions is always (in any reachable state) possible. More precisely, if the $a?$-transitions in A leaving from a location q are given by

$$q \xrightarrow{a?, g_1, r_1} q_1 \ldots, q \xrightarrow{a?, g_n, r_n} q_n,$$

then A is input enabled iff the expression $\text{Inv}(q) \implies \bigvee_{i=1}^{n}(g_i \wedge \text{Inv}(q_i)[r_i])$ is equivalent to true, where we use the convention that $\bigvee_{i=1}^{0} \cdots$ yields false. Here, $I[r]$ denotes the invariant that is obtained by replacing each clock variable in I that occurs in (the reset set) r by 0. We state that an automaton is input enabled if and only if its underlying automaton is so, using the standard notion of input enabledness for TLTSs.

Example 7. The automaton in Figure 17 is input enabled. Notice that $(x \leq 2)[x := 0]$ yields true. The construction to make an automaton input enabled is shown in Figure 18. Notice that $\neg(x < 3 \implies 1 \leq x \wedge x < 2)$ is equivalent to $x < 1 \vee x \geq 2$.

A non-input enabled automaton may block input, by being unable to synchronize on it. This is often considered as a bad property, since a component is usually not

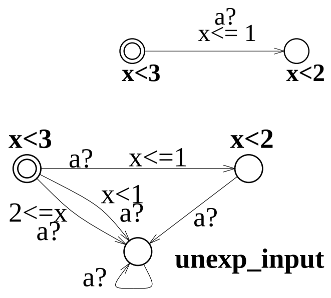

Fig. 18. Constructing an input enabled automaton

able to prevent the environment from providing inputs and this might indicate a modelling error. Therefore, it is relevant whether blocking of inputs can actually occur in a network of automata, i.e. whether or not the situation can occur that one component could provide an input to another one, while the latter automaton is not able to synchronise on it. In order to check this, we make every component A input enabled by directing every input that would otherwise be blocked by the component to a fresh location, called *unexp_input*$_A$ and check for reachability of this location. More precisely, we construct the automaton $A^{\text{u-i}}$ from A as follows. If the outgoing $a?$-transitions leaving from q are as above, then we add the transition $q \xrightarrow{a?, g} \mathit{unexp\_input}_A$, where g is equivalent to $\neg(\text{Inv}(q) \implies \bigvee_{i=1}^{n}(g_i \vee \text{Inv}(q_i)[r_i]))$. (In particular, if q does not have any outgoing $a?$-transitions, then we add $q \xrightarrow{a?} \mathit{unexp\_input}_A$.) We also add the transition $\mathit{unexp\_input}_A \xrightarrow{a?} \mathit{unexp\_input}_A$ for every input action $a?$ of A.

Proposition 7. *The automaton $A^{\text{u-i}}$ is input enabled.*

By checking reachability of the location *unexp_input*, we can check whether A and $A^{\text{u-i}}$ are semantically equivalent, i.e. whether their underlying TLTSs are exactly the same when restricted to the reachable states.

Proposition 8. *The network $\mathcal{N}(A_1, \ldots, A_n)$ is semantically equivalent to the network $\mathcal{N}(A_1^{\text{u-i}}, \ldots, A_n^{\text{u-i}})$ if and only if none of the locations unexp_input$_{A_i}$ is reachable in the network $\mathcal{N}(A_1^{\text{u-i}}, \ldots, A_n^{\text{u-i}})$.*

A.4 Verification of trace inclusion

The rest of this section describes how Uppaal can be used in some cases to check whether or not $A \sqsubseteq_{\text{TR}} B$, via the construction of B^{err}, A^r and B^r.

Throughout this section, we assume that the visible actions of A are included in those of B. Recall that we assume that B does not contain committed locations or urgent channels.

A.4.1 The construction of B^{err}

If we want to check $A \sqsubseteq_{\mathrm{TR}} B$, for a deterministic automaton B, then we build an automaton B^{err}, which is an adaption of construction by [2]. The automaton 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 such 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.

Definition 6. The automaton B^{err} is defined as follows.

1. The locations of B^{err} are the locations of B together with the (fresh) location *error*,
2. the initial location of B^{err} is the initial location of B,
3. there are no location invariants in B^{err},
4. the transitions of the automaton B^{err} are given as follows, where q and q' range over locations in B, and a over visible actions in B.

$$\begin{aligned}\cup_{q,a,q'} (&\{q \xrightarrow{a,g\wedge I,r}_{B^{err}} q' \mid q \xrightarrow{a,g,r}_B q', I = \mathrm{Inv}(q)\} \cup \\ &\{q \xrightarrow{\neg I} error \mid I = \mathrm{Inv}(q)\} \cup \\ &\{q \xrightarrow{a,g_{aq}\wedge I} error \mid I = \mathrm{Inv}(q), \\ &\quad g_{aq} = \neg \bigvee\{g \mid q \xrightarrow{a,g,r}_B q'\}\} \cup \\ &\{error \xrightarrow{a} error\}).\end{aligned}$$

For a construction of B^{err} in the presence of urgent channels and shared variables, see [16,15]. This work also describes how to encode, what are called, timed ready simulation relations as reachability properties. However, we claim that our construction also works, if committed locations are only used to encode multisynchronisation between B and components of A.

Example 8. Figure 19 illustrates an automaton and its error-construction.

In order to decide whether $A \sqsubseteq_{\mathrm{TR}} B$, we check whether the *error*-location is reachable in the composition of A and B^{err}. In order to enforce synchronisation between corresponding actions in A and B^{err}, we change the actions $a?$ in B^{err} into $a!$ and $a!$ into $a?$. Note that B^{err} is not deterministic, even not if B is so.

Lemma 1. *Let B be a deterministic automaton. Every finite sequence over $\mathrm{Act}_B\backslash \cup \mathbb{R}^{>0}$ is a trace of B^{err}. Such a sequence is a trace of B if and only if none of the executions in B^{err} with this trace reach the error location.*

Proposition 9. *Assume that B is a deterministic automaton. Then $A \sqsubseteq_{\mathrm{TR}} B \iff$ error is not reachable in $A \parallel B^{err}$.*

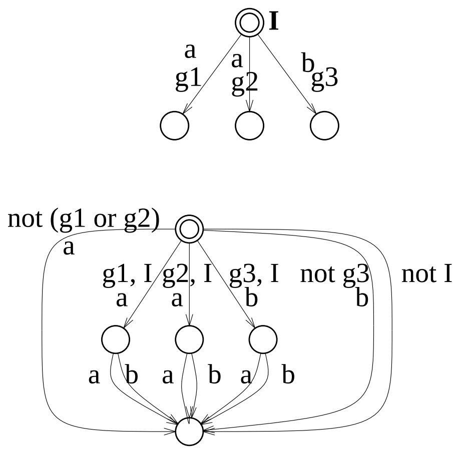

Fig. 19. An automaton and its error-construction

A.4.2 The construction of A^r and B^r

If B is non-deterministic, then we can try to make it deterministic by renaming its labels and then use the above method to verify trace inclusion.

Definition 7. A *renaming function* is a function h : Names $\to$ Names. For an automaton (resp. a TLTS) A, we denote by A^h the automaton (resp. the TLTS) obtained by replacing every visible action $a?$ in A by $h(a)?$ and $a!$ by $h(a)!$.

Lemma 2. *Let A, B be automata (resp. TLTSs) and let h be a renaming function on A. Then*

$$A \sqsubseteq_{\mathrm{TR}} B \implies A^h \sqsubseteq_{\mathrm{TR}} B^h.$$

The previous result shows that, for proving $A_1 \sqsubseteq_{\mathrm{TR}} B_1$, it suffices to find A_2, B_2 and a renaming function h such that: B_2 is deterministic, ${A_2}^h = A_1$, ${B_2}^h = B_1$ and $A_2 \sqsubseteq_{\mathrm{TR}} B_2$. The latter can be verified by Uppaal. Notice that $A_2 \not\sqsubseteq_{\mathrm{TR}} B_2$ does not imply $A_1 \not\sqsubseteq_{\mathrm{TR}} B_1$ and that even if $A_1 \sqsubseteq_{\mathrm{TR}} B_1$, it is not always possible to find the required A_2, B_2 and h. (For instance if A_1 consists of the transition $s \xrightarrow{a} s'$ and B_1 of $t \xrightarrow{\tau} t'$ and $t' \xrightarrow{a} t''$, then no such a h exists.)

In our verification, we have constructed h, A_2 and B_2 from A_1, B_1 and a given (conjectured) step refinement f. First, we describe this construction for TLTSs L_1 and L_1, yielding TLTSs L_2 and L'_2. Informally, a step refinement is a function f from the states of A to the states of B such that every transition $s \xrightarrow{a} s'$ of L can be mimicked in L' by the transition $f(s) \xrightarrow{a}_B f(s')$, where we also allow internal transitions in L to be mimicked by remaining in the same state in L'. Formally, a *step refinement* from L to L' is a function mapping the state space of L to the state space of L' such that the initial state of L is mapped to the initial state of L' and for all transitions $s \xrightarrow{a}_L t$ of L leaving from a reachable state s we have either that $f(s) \xrightarrow{a}_{L'} f(t)$ or that a is internal and $f(s) = f(t)$. A step refinement from an automaton A to an automaton B is a step refinement from the underlying TLTS of A to the underlying TLTS of B.

The idea behind the construction of L_2 and L'_2 (on the level of TLTSs) from f and L_1 and L'_1 is as follows: by the step refinement we know for each transition in L'_1 exactly the transitions in A_1 with which it can synchronize. Therefore, we give all the actions occurring in these transitions the same, fresh label, thus removing the nondeterministic choices in L'_1. The TLTSs obtained in this way from L_1 and L'_1 are denoted by ${L_1}^f$ and ${L'_1}^f$ respectively.

Formally, the TLTSs L^f and L'^f are obtained from L and L' as follows: We start with the same state space and transition relation as in L and L' respectively. Then for all sources of nondeterminism in L' *i.e.* all transitions $t \xrightarrow{a}_{L'} t'$ in L' such that either $a = \tau$ or $\exists t'' \neq t'[t \xrightarrow{a}_{L'} t'']$, we change the label a into $c_{t,a,t'}$, which yields $t \xrightarrow{c_{t,a,t'}}_{L'^f} t'$ in L'^f. Then, for all transitions in A, we replace $s \xrightarrow{a} s'$ by $s \xrightarrow{c_{f(s),a,f(s')}}_{A^f} s'$ in A^f if and only if $f(s) \xrightarrow{c_{f(s),a,f(s')}}_{B^f} f(s')$. We require for the relabeling function $c_{\cdot,\cdot,\cdot}$ that

1. the label $c_{s,a,t}$ is not a visible action of A and B,
2. $c_{s,a,t} \neq c_{s,a,t'}$ for $t \neq t'$,
3. $c_{s,a,t} \neq \tau$, and
4. $c_{s,a,s'} \neq c_{t,b,t'}$ for $a \neq b$.

Requirement 1 ensures that $c_{s,a,t}$ is fresh. The second requirement ensures that outgoing transitions of the same state get different labels and the third that no actions are relabeled into τ; together they ensure that the relabeled automaton obtained by the construction above is deterministic. The last requirement ensures that transitions with different labels get different labels after relabeling.

If $c_{s,a,t}$ is contains actions that are also used for synchronization between components of A, we achieve that B also takes part in these synchronizations by using three-way synchronization as described in A.2.

Example 9. Consider the TLTSs in Figure 20, where we omitted time passage actions $s \xrightarrow{d} s$, where d can be arbitrary. It is clear that the function $s_i \mapsto t_i$ for $i = 1,2,5,6$ and $s_3 \mapsto t_1, s_4 \mapsto t_2$ is a step refinement form A to B and that both the transitions $s_0 \xrightarrow{a} s_1$ and $s_0 \xrightarrow{a} s_3$ should synchronize with the transition $t_0 \xrightarrow{a} t_1$. Hence these transitions get the same action label by the renaming. Similarly, $s_0 \xrightarrow{a} s_1$ and $s_0 \xrightarrow{a} s_3$ should synchronize and get the same labels (but different the three previously mentioned) after renaming.

Now, take h to be the identity on the names of A and B (including τ) and $h : (c_{s,a,t}) = a$ for the new names. In the example, h is given by $\{a_1 \mapsto a, a_2 \mapsto a, b \mapsto b, c \mapsto c\}$. Now, we have the following.

Lemma 3. *Let A,B be TLTSs, f a function from the state space of A to the state space of B, and h as defined in the preceding text. Then*

1. $A^{f^h} = A$ and $B^{f^h} = B$.
2. $A^f \sqsubseteq_{\mathrm{TR}} B^f \implies A \sqsubseteq_{\mathrm{TR}} B$.

Moreover, we claim that $A^f \sqsubseteq_{\mathrm{TR}} B^f$ if f is a step refinement. (Then Lemma 3 yields that $A \sqsubseteq_{\mathrm{TR}} B$, which is a basic result for step refinements.) We do not use this claim in the verification.

In order to construct A^f componentwise, remark that $(A_1 \parallel A_2)^f = {A_1}^f \parallel {A_2}^f$. In order to build the automaton, $((A_1 \parallel A_2) \setminus C)^f$, we put ${A_1}^f$, ${A_2}^f$ and B^f in parallel, but we enforce multisynchronisation between A_1, A_2 and B on all actions $c_{s,a,t} \in C \cap \mathrm{Act}_{A_1} \cap \mathrm{Act}_{A_2}$, i.e. on those actions that are used for synchronisation between A_1 and A_2 on which we want (as $c_{s,a,t} \in C$) B also to synchronise.

The problem is how to lift this construction from TLTSs to automata, *i.e.* if L is the TLTS underlying A and L' the one underlying B, we wish to find automata A^f and B^f whose underlying TLTSs are L^f and L'^f respectively.

In general, we conjecture that this construction can be lifted to the level of automata, provided that r is given by a (finite) expression over the clocks and locations of the automaton. Although the existence of a step refinement already implies trace inclusion, this construction (on automata) would yield a method to check this inclusion with Uppaal. Although a relabeling procedure can be complex in general, in our case study it is much faster than checking whether the function r is a step refinement.

B Appendix

a, b	action (names)
f	step refinements
g	guards
h	renaming funcion
q	locations
r	reset sets
s	states
v	clock valuations
x, y	clocks
A	automaton
B	deterministic automaton
C	sets of action names
I	location invariants
L	TLTSs

References

1. L. Aceto, A. Burgueño, and K.G.Larsen. Model checking via reachabilty testing for timed automata. In *Proc. of*

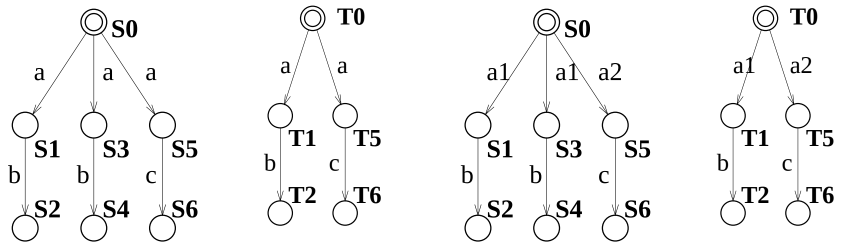

Fig. 20. The automata A, B and A^r and B^r

the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer-Verlag, 1998.

2. R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.
3. R. Alur, T.A. Henzinger, and M.Y. Vardi. Parametric real-time reasoning. *25th Annual ACM Symposium on Theory of Computing (STOC 1993)*, pages 592–601, 1993.
4. A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proceedings of the International Confence on Computer Aided Verification (CAV00)*, Chicago, USA, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434. Springer-Verlag, 2000.
5. E.M. Clarke and J. Wing. Formal methods: State of the art and future directions. In *ACM Computing Surveys*, volume 28(4), 1996.
6. P.R. D'Argenio, J.-P. Katoen, T.C. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In *Proc. of the 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Enschede, The Netherlands, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer-Verlag, 1997.
7. C. Daws and S. Yovine. Two examples of verification of multirate automata using KRONOS. In *16th Annual IEEE Real-Time Systems Symposium,* Pisa, Italy, December 1995, pages 66–75. Computer Society Press, 1995.
8. TVS group Delft University of Technology. PMC: Prototype Model Checker. `http://tvs.twi.tudelft.nl/toolset.html`.
9. K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modelling and analysis of an audio/video protocol: An industrial case study using uppaal. In *Proc. of the 18th IEEE Real-Time Systems Symposium,* San Francisco CA, USA, pages 2–13, 1997.
10. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Software Tools for Technology Transfer*, volume 1, 1997. Also available via `http://www-cad.eecs.berkeley.edu/ ~tah/HyTech/`.
11. T.S. Hune, J.M.T. Romijn, M.I.A. Stoelinga, and F.W. Vaandrager. Linear parametric model checking of timed automata. To appear in Proceedings TACAS'2001.
12. IEEE Computer Society. IEEE Standard for a High Performance Serial Bus. Std 1394-1995, August 1996.
13. IEEE Computer Society. P1394a Draft Standard for a High Performance Serial Bus (Supplement). Draft 3.0, September 1999.
14. Basic Research in Computer Science at Aalborg University and Department of Computer Systems (DoCS) at Uppsala University. `http://www.docs.uu.se/docs/rtmv/uppaal/`.
15. H. E. Jensen, K.G. Larsen, and A. Skou. Scaling up Uppaal - automatic verification of real-time systems using compositionality and abstraction. In J. Mathai, editor, *Proceedings of the 6th International School and Symposium on Formal Techniques and Fault Tolerant Systems (FTRTFT00),* Pune, India, September 2000, volume 1926 of *Lecture Notes in Computer Science*, pages 19–30. Springer-Verlag, 2000.
16. H.E. Jensen. *Abstraction-Based Verification of Distributed Systems.* PhD thesis, Department of Computer Science, Aalborg University, Denmark, June 1999.
17. P. Kannelakis and S. Smolka. Ccs expressions, finite state processes and three problems of equivalence. *Information and Computation*, pages 43–68, 1990.
18. D. LaFollette. SubPHY Root Contention, Overhead transparencies, August 1997. Available through URL `ftp://gatekeeper.dec.com/pub/standards/io/1394/P1394a/Documents/97-043r0.pdf`.
19. Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *Springer International Journal of Software Tools for Technology Transfer*, 1(1+2):134–152, 1997.
20. N.A. Lynch and F.W. Vaandrager. Action transducers and timed automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.
21. K.L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem.* Kluwer Academic Publishers, 1993.
22. J.M.T. Romijn. *Analysing industrial protocols with formal methods.* PhD thesis, University of Twente, October 1999. Available via `http://www.cs.kun.nl/~judi`.
23. M. Sighireanu and R. Mateescu. Verification of the link layer protocol of the IEEE 1394 serial bus (FireWire): an experiment with E-LOTOS. *Springer International Journal on Software Tools for Technology Transfer*, 2(1):68–88, 1998.
24. M.I.A. Stoelinga. `http://www.cs.kun.nl/~marielle/uppaal/`.
25. M.I.A. Stoelinga and F.W. Vaandrager. Root contention in IEEE 1394. In J.-P. Katoen, editor, *Proceedings of 5th AMAST Workshop on Real-Time and Probabilistic*

Systems (ARTS'99) Bamberg, Germany, May 1999, volume 1601 of *Lecture Notes in Computer Science*, pages 53–75. Springer-Verlag, 1999. Also, Technical Rapport CSI-R9905, Computing Science Institute, University of Nijmegen, May 1999.

26. Takayuki Nyu. Modified Tree-ID Process for Long-haul Transmission and Long PHY_DELAY, Overhead transparencies, 1997. Available through URL `ftp://gatekeeper.dec.com/pub/standards/io/1394/P1394a/Documents/97-051r1.pdf`.
27. S. Yovine. Kronos: A verification tool for real-time systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1/2,), October 1997.