A Proof-checked Verification of a Real-Time Communication Protocol [*]

Indra Polak
Department of Philosophy, Utrecht University
P.O.Box 80126,3508 TC Utrecht, The Netherlands
ipolak@phil.ruu.nl

Abstract

We present an analysis of a protocol developed by Philips to connect several components of an audio-system. The verification of the protocol is carried out using the timed I/O-automata model of Lynch and Vaandrager. The verification has been partially proof-checked with the interactive proof construction program Coq. The proof-checking revealed an error in the correctness proof (not in the protocol!).

1 Introduction

The verification of distributed systems is considered to be an important topic of current research. Especially *real time systems* where discrete and continuous quantities play important roles are interesting since these systems are very difficult to build error-free. Where many aspects of human life are becoming more and more dependent on such systems, the importance of the correctness of these increases.

To help us human beings build good systems, the support of the computer can be very worthwile. One approach is to formally check a hand-made correctness proof with the computer. There are currently a number of programs that can be used to support this process. One of these is the type theory based system Coq [5].

In this case study we will report on our experiences when proof-checking a real-time communication protocol with the Coq system.

This article builds on [4], where the specification and correctness proof were shown. We add the proof-checking in this article. Sections 4 and 5 are almost directly taken from [4]. We use a slightly different formal model that was easier to implement in Coq.

The protocol we analyzed is known as the *Manchester protocol* and is in this case being used in audio systems. We will use a variant of the *timed I/O-automata model* of Lynch and Vaandrager to specify the protocol. The proof itself is stated in many sorted predicate logic.

The article proceeds as follows. In section 2 the protocol is introduced. In section 3 timed I/O automata are explained. Section 4 shows the specification of the protocol while section 5 deals with the correctness proof. In section 6 we introduce Coq and treat the proof-checking. The focus will be on section 6 since the others are well treated in [4].

[*]Part of this work took place in the context of ESPRIT Basic Research Action 6453: 'Types for Proofs and Programs'.

2 The Manchester protocol

The protocol we analyzed is known as the bi-phase or *manchester* encoding. Imagine a sender-receiver pair that is to communicate messages consisting of a (finite) number of bits on a bi-state communication channel. Both parties have a real-time clock. They have agreed upon the following protocol:

- Divide the time-axis in equal parts called *bit-slots*.
- The first bit of a message is always 1.
- An upgoing edge in the middle of a bit-slot means a 1.
- A downgoing edge in the middle of a bit-slot means a 0.
- The other up- and downgoing edges are done on the boundaries of the bit-slots.
- The state of the channel is low when not used.

See figure 1.

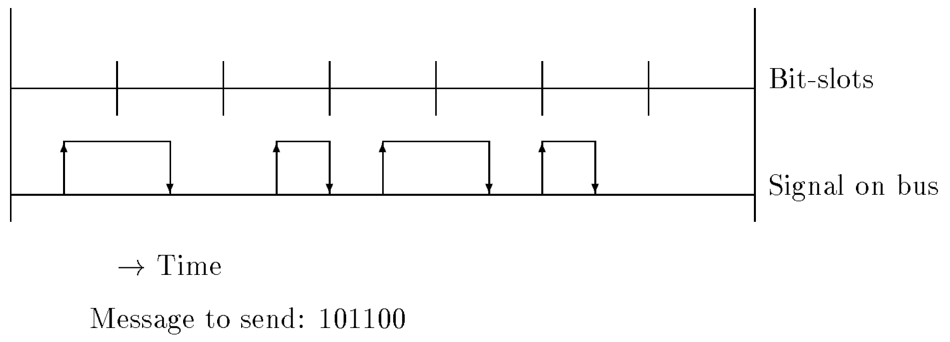

Figure 1: Bi-phase protocol.

For reasons of economy, the upgoing edges are excuted better than the downgoing ones. Therefore, the receiver only sees the upgoing edges, since these are sharp, while the downgoing edges are not and are therefore difficult to clock. By measuring the time between the *upgoing* edges, the receiver can in most cases still reconstruct the sent message. A problem occurs when the message has the form $x10$. The receiver can not distinguish this message from the message $x1$, since the last zero bit is communicated with a down-going edge. To solve this problem, the following convention is used: if the last bit received by the receiver is 1, and if the number of bits received is even, the receiver assumes that a 0 bit was sent too, and outputs a message ending with 10 to the environment. Now all messages having odd length can be properly detected with the above protocol.

Alas, the sender and the receiver have clocks that drift. How much is unknown. This drift is expressed as a tolerance N on the clocks, what this means exactly will be explained later on. The main problem of the designer can now be stated as follows :

Find the maximum tolerance N for which the above protocol functions correctly.

Correct functioning means in this case that whenever the sender sends a bit-string s, the receiver will always receive the same bit-string s.

The above protocol is used by Philips in certain audio equipment to provide communication between audio sets. This gives the system more possibilities with respect to embedding intelligence in the system. Now in order to record a tape from a Compact Disc, only one button needs to be pushed instead of several more.

The control information is sent over a tiny network that connects the various devices. The bus communication on this network is ruled basically by the above protocol, so Philips needs to know how precise their clocks must be. Otherwise the messages need not be communicated correctly, and instead of playing your tape, you might end up with an erased one. Of course Philips did test their systems rigorously for a certain tolerance, but it remains interesting to know how high the tolerance may be in general. If the found tolerance is higher than the one used, cheaper clocks can be put in the product instead.

3 Timed I/O-automata

In this section we introduce some preliminaries on a variant of timed I/O-automata.

An *action signature* S is a triple $(in(S), out(S), int(S))$ of three disjoint sets of respectively *input actions*, *output actions* and *internal actions*. We derive from these the sets of *external actions*, *locally controlled actions* and *actions of* S as follows:

$$\begin{aligned} ext(S) &= in(S) \cup out(S) \\ local(S) &= out(S) \cup int(S) \\ acts(S) &= in(S) \cup out(S) \cup int(S). \end{aligned}$$

Definition 3.1 *A* timed I/O-automaton A *consists of five components*

1. *a nonempty set* $states(A)$ *of states,*
2. *a nonempty set* $start(A) \subseteq states(A)$ *of startstates,*
3. *a nonempty set* $accepting(A)$ *of acceptingstates,*[1]
4. *an action signature* $sig(A)$*,*
5. *a set* $steps(A) \subseteq states(A) \times (acts(A) \cup \mathbb{R}^+) \times states(A)$ *of transitions.*

We will let $s \overset{a}{\rightarrow}_A s'$ abbreviate $(s, a, s') \in steps(A)$. We will often omit the subscript A when no confusion can arise. The possible actions are divided in normal actions that do not influence time and explicit time actions that add an amount of time to the current time. The

[1]In this respect our timed I/O-automata differ from the usual ones. We need this to express the correctness criterion more conveniently and point out the relation with Timed Büchi Automata

intuition is that every normal action has a *time stamp* that records the time that the action occurred. We have the following restraint:

Restraint Each input action is enabled in each state.

This restraint makes the assumption that input actions are controlled by the environment and can therefore not be prevented by the system.

3.1 Composition, Hiding

The composition of two (or more) timed I/O-automata is constructed with a Cartesian product. This achieves synchronizing on common events and interleaving on others. To make two automata $S1$ and $S2$ compatible we rename actions in such a way that $out(S_1) \cap out(S_2) = \emptyset$, $int(S_1) \cap acts(S_2) = \emptyset$, and $int(S_2) \cap acts(S_1) = \emptyset$. The *composition* $S_1 \parallel S_2$ of a pair of compatible action signatures S_1, S_2 is defined to be the action signature S, with

$$\begin{aligned} in(S) &= (in(S_1) \cup in(S_2)) - (out(S_1) \cup out(S_2)), \\ out(S) &= out(S_1) \cup out(S_2), \\ int(S) &= int(S_1) \cup int(S_2). \end{aligned}$$

Now the composition of two timed automata A_1 and A_2 is the timed automaton A with

1. $states(A) = states(A_1) \times states(A_2)$,
2. $start(A) = start(A_1) \times start(A_2)$,
3. $accepting(A) = accepting(A_1) \times accepting(A_2)$,
4. $sig(A) = sig(A_1) \parallel sig(A_2)$,
5. $steps(A)$ is the set of triples $((s_1, s_2), a, (s_1', s_2'))$ in $states(A) \times (acts(A) \cup \mathbb{R}^+) \times states(A)$ such that, for $i \in \{1, 2\}$, if $a \in acts(A_i) \cup \mathbb{R}^+$ then $s_i \xrightarrow{a}_{A_i} s_i'$ else $s_i = s_i'$.

Note that since time actions are synchronized, time can only pass if both automata can let time pass with the same amount.

Suppose $H \subseteq out(S)$ for some action signature S. We define the action signature **HIDE** H **IN** S as the triple $(in(S), out(S) - H, int(S) \cup H)$.

Now if A is an I/O-automaton and $H \subseteq out(A)$, then **HIDE** H **IN** A is the I/O-automaton obtained from A by replacing $sig(A)$ by **HIDE** H **IN** $sig(A)$, and leaving the other components unchanged.

3.2 traces

An *execution fragment* of a timed automaton A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 ...$ of states and actions of A, beginning with a state, and if it is finite also ending with a state, such that for all i, $s_i \overset{a_{i+1}}{\to} s_{i+1}$. Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 ...$ is an execution fragment of A. Let γ bet the sequence consisting of the actions in α: $\gamma = \alpha_1 \alpha_2 ....$ Then $trace(\alpha)$ is defined to be the sequence γ. We have a function $h : int(A) \to \{\tau\}$ that hides internal actions by renaming them to τ. A *computation* is an infinite trace that starts in a start state and infinitely often visits accepting states, where all internal actions are renamed into τ by h . We will consider two computations equal iff the computations without τ actions are equal.

In this way our timed I/O-automata resemble *timed Büchi automata*, see [2]. So our model differs from usual timed I/O-automata ([10]) in the following ways:

- We do not really make a distinction between normal actions and time actions.
- We allow abnormal traces like *Zeno*-traces to occur, since they do not influence the correctness of the protocol.

But there are a number of aspects that make our model similar to timed I/O-automata:

- We use a notion of input-enabledness
- We use the same precondition/effect style and state variables

so in need of a better name we have chosen to call our model a variant of timed I/O-automata.

3.3 simulations

Definition 3.2 A weak timed forward simulation *from A to B is a relation $R \subseteq states(A) \times states(B)$ that satisfies the following conditions:*

1. *If $s \in$ start(A), then there exists a state s' $\in$ start(B) such that $R(s, s')$.*
2. *If $s \xrightarrow{a}_A s'$, and $R(s, u)$ for some $u \in states(B)$, then there exists a state u' of B such that $u \xrightarrow{a}_B u'$ and $R(s', u')$, or from state s' we can never reach an accepting state anymore.*
3. *If s is an accepting state of A, and $R(s, s')$, then s' is an accepting state of B.*

Lemma 3.3 *If there exists a weak timed forward simulation from A to B, then $computations(A) \subseteq computations(B)$.*

Proof: By induction on the length of the trace. ■

This lemma states that if we have a weak timed forward simulation from A to B, then all computations of A are contained in that of B. We will say that A behaves correct w.r.t. B. We will use a simulation to express the correctness criterion of our specification.[2]

Another problem is a notion of fairness. As it stands now, the specification does not guarantee fairness. However we believe that in this protocol the only unfair behavior results from so-called *zeno-traces* that do not occur in the real world. As these traces do not violate the correctness, we let them be in the specification.

[2]This criterion coincides with the correctness criterion used with Timed Buchi Automata, where only infinite words that infinitely often visit members of a set of accepting states are considered.

4 Specification

In this section we will show a formal specification of the protocol. We will first explain the syntax we will use in defining the timed I/O-automata. In general we follow the same patterns as in [4].

4.1 Precondition/effect notation

We will use the characteristic precondition/effect style of notation for I/O-automata. The advantage of this notation is mainly that we can reason about many states and transitions conveniently.

In this notation, states are represented by state variables. Each variable has an abstract data type as domain. This data type is defined using standard equations over a signature Σ and a many sorted Σ-Algebra A.

Each action is defined using a precondition over the state-variables and an effect that gives the state variables new values after the action. A primed variable denotes the variable after the action. Only if a precondition is true for a certain state, then the transition is possible. For the *TIME* action we use an *action formula* that states the effect on all variables slightly different. Iff the action formula is true, then we can do the *TIME* action.

Now to make our presentation more readable, we introduce notational sugar for an *if then else* and a *case* construct. We also use a standard notation for logical connectives.

We make a distinction between *Input actions*, *Output actions*, *Internal actions*, *Discrete variables*, *Continuous variables* and *Initial formulas.* The initial formula represents the start states of the timed I/O-automaton.

4.2 Data types

We start the specification of the protocol with a description of the various data types that we will need. We assume a many-sorted signature Σ and a Σ-algebra $\mathcal{A}$ which consists of the following components:

- a type **Nat** of natural numbers, with constant symbol zero, successor function symbol succ, and a predicate symbol odd, all with the usual interpretation. Also, there is an embedding $\iota : \mathbf{Nat} \to \mathbf{Real}$ of the natural numbers into the reals. We will suppress ι's in terms.

- a type **Bit** of bits that the protocol has to transmit, with constants symbols 0 and 1. Again there is an embedding $\iota : \mathbf{Bit} \to \mathbf{Real}$, which we will suppress in terms.

- a type **List**, with as domain the collection of finite lists of bits. There is a constant symbol ϵ for the empty list, an embedding $\langle . \rangle : \mathbf{Bit} \to \mathbf{List}$, and a binary function symbol $\hat{}$, denoting concatenation of lists. Besides these constructors, there are function symbols

$\mathsf{head} : \mathbf{List} \to \mathbf{Bit}$	$\mathsf{last} : \mathbf{List} \to \mathbf{Bit}$
$\mathsf{tail} : \mathbf{List} \to \mathbf{List}$	$\mathsf{last\_two} : \mathbf{List} \to \mathbf{List}$
$\mathsf{length} : \mathbf{List} \to \mathbf{Nat}$	

head takes the first element of a list (defined arbitrarily as 0 in case of the empty list), tail returns the remainder of a list after removal of the first element, last gives the last element of a list, $\mathsf{last\_two}$ gives the last two elements of a list, and length returns the length of a list. These operations are fully characterized by the axioms (here m is a variable of type **List**, and d, e are variables of type **Bit**):

$$\begin{array}{lcllcl}
\mathsf{head}(\epsilon) & = & 0 & \mathsf{last\_two}(\epsilon) & = & \epsilon \\
\mathsf{head}(\langle d\rangle\hat{\ } m) & = & d & \mathsf{last\_two}(\langle d\rangle) & = & \langle d\rangle \\
\mathsf{tail}(\epsilon) & = & \epsilon & \mathsf{last\_two}(m\hat{\ }\langle de\rangle) & = & \langle de\rangle \\
\mathsf{tail}(\langle d\rangle\hat{\ } m) & = & m & \mathsf{length}(\epsilon) & = & \mathsf{zero} \\
\mathsf{last}(\epsilon) & = & 0 & \mathsf{length}(\langle d\rangle\hat{\ } m) & = & \mathsf{succ}(\mathsf{length}(m)) \\
\mathsf{last}(m\hat{\ }\langle d\rangle) & = & d & & &
\end{array}$$

Here (and elsewhere) we write $\langle de\rangle$ for $\langle d\rangle\hat{\ }\langle e\rangle$. Finally, we need an operation $\mathsf{finalize}$: **List** $\rightarrow$ **List** defined by:

$$\mathsf{if}\ \mathsf{last}(m){=}1 \wedge \mathsf{odd}(\mathsf{length}(m))\ \mathsf{then}\ \mathsf{finalize}(m){=}m\ \mathsf{else}\ \mathsf{finalize}(m){=}m\hat{\ }\langle 0\rangle.$$

- a type **Bool** of booleans with constant symbols true and false. We view boolean valued terms as formulas and use b as an abbreviation of $b{=}\mathsf{true}$.
- a function symbol min : **Real** $\times$ **Real** $\rightarrow$ **Real**, with the obvious interpretation, and two constant symbols Q and T of type **Real**:
  - Q denotes one quarter of the length of a bit slot in the Manchester encoding. In the Philips specifications Q equals $222\mu s$.
  - T gives the tolerance on the timing of the sender and receiver in the protocol. Philips allows a maximum tolerance of $\pm\frac{1}{20}$.

  In this paper we will assume $\mathsf{Q} > 0$ and $0 \leq \mathsf{T} < 1$.

4.3 The sender

We now define the system S, which models the sender of the protocol. The discrete variables of S are a variable *list*, which records the bit string still to be transmitted, a boolean *wire_high* to keep track of the voltage on the wire, and a boolean *transmitting* which records whether the sender is busy transmitting. There is also a continuous variable x which represents a drifting clock with tolerance T that is reset in the middle of each bit slot. The input action $IN(m)$ corresponds to a request by the environment to transmit a bit string m. Upon the occurrence of such an action in the initial state, S immediately does an *UP*-action, which represents an upgoing edge on the bus. Depending on whether the second bit in the string is absent, 0 or 1, a *DOWN*-action occurs $2\mathsf{Q}$ or $4\mathsf{Q}$ time units after the first *UP*, according to the local clock of S. An action *DOWN* represents a downgoing edge on the bus. Subsequent actions *UP* and *DOWN* are generated as required by the Manchester encoding, and when the transmission is finished the protocol returns to its initial state. *IN*-actions that occur before the transmission finishes are ignored.

Inputs	IN : **List**		
Outputs	UP		
Internals	$DOWN$		
Discrete	$transmitting$: **Bool**	**Init**	$\wedge\ \neg transmitting$
	$wire\_high$: **Bool**		$\wedge\ \neg wire\_high$
	$list$: **List**		$\wedge\ list{=}\epsilon$
Accepting	$\wedge\ \neg transmitting$		
	$\wedge\ \neg wire\_high$		
	$\wedge\ list{=}\epsilon$		
Continuous	x : **Real**		

```
$IN(m)$
    **Precondition**
        $\wedge\ \mathsf{head}(m){=}1$
        $\wedge\ (\mathsf{odd}(\mathsf{length}(m)) \vee \mathsf{last\_two}(m){=}\langle 00\rangle)$
    **Effect**
        if $\neg transmitting \wedge \neg wire\_high \wedge list{=}\epsilon$ then  [$list := m$
                                                                      $x := 0$]

$UP$
    **Precondition**
        $\wedge\ \neg wire\_high$
        $\wedge\ list{\neq}\epsilon$
        $\wedge$ if $transmitting$ then (if $\mathsf{head}(list){=}1$ then $x{=}4\mathsf{Q}$ else $x{=}2\mathsf{Q}$) else $x{=}0$
    **Effect**
        $transmitting := \mathsf{true}$
        $wire\_high := \mathsf{true}$
        if $\mathsf{head}(list){=}1$ then  [$list := \mathsf{tail}(list)$
                                    $x := 0$]

$DOWN$
    **Precondition**
        $\wedge\ wire\_high$
        $\wedge$ if $list{\neq}\epsilon \wedge \mathsf{head}(list){=}0$ then $x{=}4\mathsf{Q}$ else $x{=}2\mathsf{Q}$
    **Effect**
        if $list{=}\epsilon \vee list{=}\langle 0\rangle$ then  [$transmitting := \mathsf{false}$]
        $wire\_high := \mathsf{false}$
        if $list{\neq}\epsilon \wedge \mathsf{head}(list){=}0$ then  [$list := \mathsf{tail}(list)$
                                              $x := 0$]

$TIME(t)$
    **Action formula**
        $\wedge\ t > 0$
        $\wedge\ 1 - \mathsf{T} \leq \frac{x'-x}{t} \leq 1 + \mathsf{T}$
```

4.4 The receiver

Next we define system R, which models the receiver of the protocol. System R has two state variables: a discrete variable *list*, which gives the bit string received thus far, and a continuous variable x, which represents a drifting clock with tolerance T that is reset whenever an upgoing edge is detected. There are two actions: an action UP that corresponds to the detection of an upgoing edge, and an action OUT by which the receiver passes a received string on to the environment. The action predicates for UP and OUT are straightforward formalizations of the informal specifications by Philips of the receiver algorithm.

Inputs	UP		
Outputs	OUT : **List**		
Discrete	$list$: **List**	**Init**	$list{=}\epsilon$
Accepting	$list{=}\epsilon$		
Continuous	x : **Real**		

```
UP
  Precondition
    true
  Effect
    case
      $list{=}\epsilon$ $\Rightarrow list := \langle 1\rangle$
      $\mathsf{last}(list){=}0 \Rightarrow$ case
            $x < 3\mathsf{Q}$ $\Rightarrow list := \epsilon$
            $3\mathsf{Q} \le x < 5\mathsf{Q} \Rightarrow list := list\hat{}\langle 0\rangle$
            $5\mathsf{Q} \le x$ $\Rightarrow list := list\hat{}\langle 01\rangle$
      $\mathsf{last}(list){=}1 \Rightarrow$ case
            $x < 3\mathsf{Q}$ $\Rightarrow list := \epsilon$
            $3\mathsf{Q} \le x < 5\mathsf{Q} \Rightarrow list := list\hat{}\langle 1\rangle$
            $5\mathsf{Q} \le x < 7\mathsf{Q} \Rightarrow list := list\hat{}\langle 0\rangle$
            $7\mathsf{Q} \le x$ $\Rightarrow list := list\hat{}\langle 01\rangle$
    $x := 0$
```

```
$OUT(\mathsf{finalize}(list))$
  Precondition
    $\wedge\ list{\neq}\epsilon$
    $\wedge$ if $\mathsf{last}(list){=}0$ then $x{=}7\mathsf{Q}$ else $x{=}9\mathsf{Q}$
  Effect
    $list := \epsilon$
```

```
$TIME(t)$
  Action formula
    $\wedge\ t > 0$
    $\wedge\ 1 - \mathsf{T} \le \frac{x'-x}{t} \le 1 + \mathsf{T}$
```

4.5 The Full Protocol

The full protocol can now be defined as the composition of automata S and R, with communication between these components hidden:

$$Impl \ \triangleq\ \mathsf{HIDE}\ \{UP\}\ \mathsf{IN}\ (S \| R)$$

4.6 The correctness criterion

System P defines the collection of allowed behaviors of $Impl$. It has the same input and output actions as $Impl$, but no internal actions. In P each action $IN(m)$ is followed by an action $OUT(m)$ within time

$$\frac{(4\ \mathsf{length}(m) + 5)\ \mathsf{Q}}{1 - \mathsf{T}}$$

However, if the environment offers another IN-action before the system has generated a corresponding OUT-action, P moves to a state of chaos in which anything is possible. This means that in such a situation any behavior of *Impl* is allowed. In the next section we will prove that the *Impl* is indeed a correct implementation of P.

Inputs	IN : **List**		
Outputs	OUT : **List**		
Discrete	$list$: **List**	**Init**	$\wedge\ list{=}\epsilon$
	$chaos$: **Bool**		$\wedge\ \neg chaos$
Accepting	$\wedge\ list{=}\epsilon$		
	$\wedge\ \neg chaos$		
Continuous	x : **Real**		

```
IN(m)
  Precondition
    $\wedge$ $\mathsf{head}(m)=1$
    $\wedge$ $(\mathsf{odd}(\mathsf{length}(m)) \vee \mathsf{last\_two}(m)=\langle 00\rangle)$
  Effect
    if $list=\epsilon$ then  [$list := m$
                             $x := 0$]
    if $list\neq\epsilon$ then  [$chaos := \mathsf{true}$]

OUT(list)
  Precondition
    $\vee$ $(list\neq\epsilon \wedge (1-\mathsf{T})x \leq (4\,\mathsf{length}(list)+5)\,\mathsf{Q})$
    $\vee$ $chaos$
  Effect
    $list := \epsilon$

TIME(t)
  Action formula
    $\wedge$ $t > 0$
    $\wedge$ $x' - x = t$
```

5 Correctness Proof

In this section we will establish that there exists a weak timed forward simulation from the implementation to the specification. We first gain insight into the reachable states by presenting a number of *invariants*, i.e., properties that hold initially and that are preserved by the transitions. We have omitted all proofs, which are mostly routine and tedious; the creative part is finding the right invariants and the order in which to prove them.

From now on, we will assume that the tolerance T is *less than* $\frac{1}{17}$. The following scenario shows what goes wrong if $\mathsf{T} \geq \frac{1}{17}$. Assume that the sender's clock progresses *maximally slow* and the receiver's clock *maximally fast*. Now the sender and receiver are at rest and the message to be sent is "101". Immediately after the *IN* of this message the sender will output an *UP* to the receiver. Both clocks are (re)set to 0, the buffer of the sender contains "01" and that of the receiver "1". The receiver can output "1" at 9Q local receiver's time, before the last *UP* arrives at 8Q local sender's time, if

$$9\mathsf{Q} \cdot \tfrac{1-\mathsf{T}}{1+\mathsf{T}} \leq 8\mathsf{Q}.$$

And this is, as the reader can verify, when $\mathsf{T} \geq \frac{1}{17}$.

The variables in the invariants are prefixed with their origin, e.g., $S.x$ for sender's clock x. Besides the variables present in the sender and the receiver, we add to *Impl* a (discrete) boolean history variable *error* that indicates whether *Impl* is in an erroneous or chaotic state. We will need this variable to express that a premature input has occurred. Variable *error* is defined by adding a clause $\neg error$ to the initialization condition of *Impl*, and a clause

$$\mathsf{if}\ R.list{\neq}\epsilon\ \mathsf{then}\ error := \mathsf{true}$$

to the effect of *IN* in *Impl*. All the other actions, including *TIME*, leave *error* unchanged. With [8] we know that this is a harmless extension by which, as one can easily verify, the set of traces of *Impl* is not changed. We have a predicate *living* that ranges over states to express the property of being able to reach an accepting state. Since the simulation relation only says something about living states, we can ignore the other states.

We start with a few invariants about the state space of the sender. The first invariant reflects the observation that the sender is always busy (i.e., transmitting) if the bus is high.

Lemma 5.1 *The following property holds for all reachable states of Impl :*

$$S.wire\_high \quad \rightarrow \quad S.transmitting.$$

The second invariant gives upper bounds for the the various stages of progress of the sender: in the first conjunct the sender is at rest and ready to accept any input, in the second conjunct the sender has received its message but has not yet begun to transmit, in the third conjunct it is waiting to send the next "1" etc.

Lemma 5.2 *The following property holds for all reachable, living states of Impl :*

$$\begin{array}{ll}
\vee & init(S) \\
\vee & \neg S.wire\_high \wedge S.list{\neq}\epsilon \wedge \neg S.transmitting \wedge S.x{=}0 \\
\vee & \neg S.wire\_high \wedge S.list{\neq}\epsilon \wedge S.transmitting \wedge \mathsf{head}(S.list){=}1 \wedge S.x \leq 4\mathsf{Q} \\
\vee & \neg S.wire\_high \wedge S.list{\neq}\epsilon \wedge S.transmitting \wedge \mathsf{head}(S.list){=}0 \wedge S.x \leq 2\mathsf{Q} \\
\vee & S.wire\_high \wedge S.list{\neq}\epsilon \wedge \mathsf{head}(S.list){=}0 \wedge S.x \leq 4\mathsf{Q} \\
\vee & S.wire\_high \wedge (S.list{=}\epsilon \vee \mathsf{head}(S.list){=}1) \wedge S.x \leq 2\mathsf{Q}.
\end{array}$$

The next invariant gives an upper bound for the clock of the receiver.

Lemma 5.3 *The following property holds for all reachable, living states of Impl :*

$$R.list{=}\epsilon \vee \mathsf{if}\ \mathsf{last}(R.list){=}0\ \mathsf{then}\ R.x \leq 7\mathsf{Q}\ \mathsf{else}\ R.x \leq 9\mathsf{Q}.$$

We now give invariants for relations between the states of the sender and the receiver. The next invariant tells us that for a good working of the implementation an input of a new message can only happen when the receiver is at rest.

Lemma 5.4 *The following property holds for all reachable, living states of Impl :*

$$\neg S.wire\_high \wedge S.list{\neq}\epsilon \wedge \neg S.transmitting \rightarrow R.list{=}\epsilon.$$

We want to reason about the two clocks in the implementation as if they were not drifting, but precise. For this reason we introduce a new symbol $\approx$ with an intended meaning of "almost equal". With this we abstract from the amount of drifting.

Notation 5.5 *Let e and f be expressions of type* **Real**. *We define:*

$$e \approx f \quad \triangleq \quad \frac{1-\mathsf{T}}{1+\mathsf{T}} e \leq f \leq \frac{1+\mathsf{T}}{1-\mathsf{T}} e.$$

Notice that if T=0 we can read $\approx$ again as $=$. Also note that $e \approx f$ if and only if $f \approx e$.

Lemma 5.6 *Let s, s' be states of Impl and e, f terms of type* **Real** *in which no continuous variables occur. Suppose $s \models S.x + e \approx R.x + f$, and suppose that $s \xrightarrow{d} s'$, for some $d \in \mathsf{R}^+$. Then $s' \models S.x + e \approx R.x + f$.*

The most important observations about the implementation are those in which the distance between the clocks is related to the contents of the buffers of sender and receiver. We start with the possible distances and then give a more detailed description.

Lemma 5.7 *The following property holds for all reachable, living states of Impl :*

$$\begin{array}{lll}
\wedge\ S.transmitting \wedge \neg S.wire\_high & \rightarrow & \vee\ R.x \approx S.x + 4\mathsf{Q} \\
 & & \vee\ R.x \approx S.x + 2\mathsf{Q} \\
 & & \vee\ R.x \approx S.x \wedge \mathsf{head}(S.list){=}1 \\
\wedge\ S.transmitting \wedge S.wire\_high & \rightarrow & \vee\ R.x \approx S.x \\
 & & \vee\ R.x \approx S.x - 2\mathsf{Q} \wedge S.list{\neq}\epsilon \wedge \mathsf{head}(S.list){=}0.
\end{array}$$

Lemma 5.8 *The following property holds for all reachable, living states of Impl :*

$$\begin{array}{ll}
\wedge\ R.list{\neq}\epsilon & \wedge\ S.transmitting \rightarrow \\
 & \vee\ \mathsf{last}(R.list){=}1 \wedge R.x \leq \frac{1+\mathsf{T}}{1-\mathsf{T}}4\mathsf{Q} \wedge R.x \approx S.x \\
 & \vee\ \mathsf{last}(R.list){=}1 \wedge \frac{1-\mathsf{T}}{1+\mathsf{T}}4\mathsf{Q} \leq R.x \leq \frac{1+\mathsf{T}}{1-\mathsf{T}}8\mathsf{Q} \wedge R.x \approx S.x + 4\mathsf{Q} \\
 & \vee\ \mathsf{last}(R.list){=}0 \wedge R.x \leq \frac{1+\mathsf{T}}{1-\mathsf{T}}2\mathsf{Q} \wedge R.x \approx S.x - 2\mathsf{Q} \\
 & \vee\ \mathsf{last}(R.list){=}0 \wedge \frac{1-\mathsf{T}}{1+\mathsf{T}}2\mathsf{Q} \leq R.x \leq \frac{1+\mathsf{T}}{1-\mathsf{T}}6\mathsf{Q} \wedge R.x \approx S.x + 2\mathsf{Q}) \\
\wedge\ R.list{\neq}\epsilon & \wedge\ init(S) \rightarrow \\
 & \vee\ \mathsf{last}(R.list){=}1 \wedge R.x \leq 9\mathsf{Q} \wedge R.x \approx S.x \\
 & \vee\ \mathsf{last}(R.list){=}1 \wedge \frac{1-\mathsf{T}}{1+\mathsf{T}}4\mathsf{Q} \leq R.x \leq 9\mathsf{Q} \wedge R.x \approx S.x + 4\mathsf{Q} \\
 & \vee\ \mathsf{last}(R.list){=}0 \wedge \frac{1-\mathsf{T}}{1+\mathsf{T}}2\mathsf{Q} \leq R.x \leq 7\mathsf{Q} \wedge R.x \approx S.x + 2\mathsf{Q} \\
\wedge\ R.list{=}\epsilon & \rightarrow\ \neg S.transmitting \wedge \neg S.wire\_high.
\end{array}$$

The following invariant implies that, with our additional assumption that $\mathsf{T} < \frac{1}{17}$, the above defective scenario is not possible: an output of a message by the receiver cannot happen when the sender is still busy.

Lemma 5.9 *The following property holds for all reachable, living states of Impl :*
$S.list{\neq}\epsilon \wedge ((R.list{\neq}\epsilon \wedge \mathsf{last}(R.list){=}0 \wedge R.x = 7\mathsf{Q}) \vee (\mathsf{last}(R.list){=}1 \wedge R.x = 9\mathsf{Q})) \rightarrow error.$

The last invariant gives an obvious property of the specification automaton.

Lemma 5.10 *The following property holds for all reachable states of P:*

$$P.list{=}\epsilon \vee (\mathsf{head}(P.list) = 1 \wedge (\mathsf{odd}(P.list) \vee \mathsf{last\_two}(P.list) = \langle 00 \rangle).$$

We have now collected enough invariants to establish a weak timed forward simulation from the implementation to the specification. Besides a part needed to deal with premature inputs, the simulation consists of two parts: a part relating the buffers of the sender and the receiver to the buffer of the specification and a part relating the clocks of the protocol to the single precise clock of the specification. As in most verifications of data link protocols it is essential to realize at what moment which part of the message is in transit between the sender and the receiver. In our case this comes down to establishing when there is a "0" in transit that is about to be accepted by the receiver.

Theorem 5.11 *The relation determined by the following formula over the state variables of Impl and P is a weak timed forward simulation from Impl to P:*

```
SIM  ≜  if $error$ then $P.chaos$ else
         if $R.list{=}\epsilon$ then $P.list{=}S.list \wedge (S.list{=}\epsilon \vee P.x{=}0)$ else
          if $R.x \approx S.x + 2\mathsf{Q}(\mathsf{last}(R.list) - 1)$
           then  $\wedge\ P.list{=}R.list\hat{\ }S.list$
                 $\wedge\ (1-\mathsf{T})P.x \leq 4\mathsf{Q}\mathsf{length}(R.list) - 2\mathsf{Q}(1+\mathsf{last}(R.list))$
                         $+\mathsf{min}(R.x, S.x + 2\mathsf{Q}(\mathsf{last}(R.list) - 1))$
           else  $\wedge\ P.list{=}R.list\hat{\ }\langle 0 \rangle\hat{\ }S.list$
                 $\wedge\ (1-\mathsf{T})P.x \leq 4\mathsf{Q}\mathsf{length}(R.list) - 2\mathsf{Q}(1+\mathsf{last}(R.list))$
                         $+\mathsf{min}(R.x, S.x + 2\mathsf{Q}(\mathsf{last}(R.list) + 1))$.
```

6 Proof-checking

The proof-checking activities consisted mainly of the following parts:

- formalize the protocol verification in type theory;
- check the simulation using the lemmas;
- check all lemmas.

The first part was easily realized using the same approach as in [7]. Some new constructions were formalized. For example, the simulation was indeed a relation, and not a function as in [7]. This lead to the introduction of an existential quantifier in the simulation relation.

We also had the notion of *dead* and *living* states to cope with. These were straightforward to implement using inductive definitions. The real-time aspects were modeled with $\mathbb{Q}$, and some functions on $\mathbb{Q}$ had to be defined. This formed no problem either.

The proof-checking itself posed more problems. It appeared that the manipulation with data, e.g. solving an inequality with the two unknowns Q and T, took more time than expected. Another reason for this fact was that these calculations were not spelled out in the proof.

This had implications for the speed of the proof-checking process. Since many lemmas used the parameters and the $\approx$ construct, the checking of the simulation would be very time consuming. We investigated to automate the handling of these inequalities, but were unable to find a satisfiable solution.

After we checked some lemmas we therefore concentrated on the simulation. Here we found an error in the inequality part. To repair this error, the simulation relation changed radically. This meant we could start all over again. At this moment, we have checked the simulation against the lemmas, but did not yet check all lemmas.

We checked lemmas 5.1, 5.2, 5.7 and 5.8. The others are left for further work.

In the remainders of this section we will explain how the correctness proof was formalized in Coq and (partially) computer-checked. To this end, we first give a short introduction to Coq itself. After that, we will explain the translation of the specification. Finally we show some proof-checking examples. In the following we will show Coq input and output in `typewriter style`.

6.1 The Coq system

Coq is an interactive computer program based on the *Calculus of Inductive Constructions*, [5]. It uses the *Curry-Howard isomorphism* to represent natural deduction proofs as λ-terms.

In addition, we can define inductive types. If we define e.g. the natural numbers as follows :

```
Inductive Definition nat :  Set = O : nat | S : nat -> nat.
```

then we have defined the type *nat* as the smallest set X such that X is closed under two constructors, $O : X$ and $S : X \to X$. This gives us the *no confusion* and *no junk* properties normally associated with the natural numbers: all members of *nat* are different and there are no other elements besides those built from O and S.

This mechanism gives the system at least the expressive power of the primitive recursive functions. It is also very useful in the proof: when we have to do an induction over a number of possible actions in the timed I/O-automaton, we can use this inductive mechanism very elegantly.

The user communicates with Coq by typing in commands. For an example, consider the following λ-term:

$$\lambda x{:}A \to B\,.\,\lambda y{:}A\,.\,x\,y$$

This term corresponds to the following natural deduction rule:

$$\frac{A \to B \quad A}{B}\,\text{Modus Ponens}$$

We will show a proof-session in Coq that uses the above deduction rule.

```
Coq < Goal ((A -> B) /\ A) -> B.

Coq < Show.
1 subgoal
  ((A->B)/\A)->B
Coq < Intro H.
```

```
1 subgoal
  B
  ============================
    H : (A->B)/\A

Coq < Elim H.
1 subgoal
  (A->B)->A->B
  ============================
    H : (A->B)/\A

Coq < Intro H0.
1 subgoal
  A->B
  ============================
    H0 : A->B
    H : (A->B)/\A

Coq < Intro H1.
1 subgoal
  B
  ============================
    H1 : A
    H0 : A->B
    H : (A->B)/\A

Coq < Apply H0.
1 subgoal
  A
  ============================
    H1 : A
    H0 : A->B
    H : (A->B)/\A

Coq < Assumption.
Goal proved!
```

We used the following commands:

- Goal x. This tells Coq we want to proof x.
- Show. This command shows the current subgoals.
- Intro x. This corresponds with $\rightarrow$I.
- Elim x. This eliminates an inductively defined type. In this case it means that if we want to prove C from $A \wedge B$, then it is enough to prove $A \rightarrow (B \rightarrow C)$.

- Apply x. This corresponds to the Modus Ponens rule.
- Assumption. This tells Coq that the current subgoal is given in the context, i.e. the terms below the double bar.

We see the proof is constructed bottom up instead of top down: we start with the goal and then we interactively build the proof tree. For more information about Coq see [5].

6.2 I/O-automata in Coq

In the proof we constantly use very specific aspects of I/O-automata theory. For instance we often do a case distinction on all possible steps from a given state. Therefore it is useful to create a framework within Coq that makes this reasoning easy. In [7] this is done already, and in general we use the same framework.

We make a distinction between the abstract data types and the transition systems. For this particular case study, the manipulation of data is a crucial aspect. In many other case studies with Coq the manipulation of data forms a big problem. In our case this was not different. In the hand-made proof we generally use the theory of real numbers. However this datatype is not specifiable in a finite way. Therefore we chose to model $\mathbb{R}$ with $\mathbb{Q}$, the set of rational numbers.

We believe this to be a sound choice, since the hand-made proof only used the axioms of an ordered field. Therefore if the proof in $\mathbb{Q}$ can be computer checked with the axioms of an ordered field, we can use the same strategy to check the proof in $\mathbb{R}$. However, we must now pay attention to the rules we used. It would have been better if we had not chosen any model, but only stated the axioms for an ordered field. The recognition that we only used properties of an ordered field came after some time, and at that point the effort to redo the proof-checking in this setting was not cost-effective anymore.

6.3 Datatypes

In general all our datatypes are inhabitants of the predefined type *Set*. We have the following data types to translate:

- The sort BOOL of booleans;
- the sort LIST of lists of booleans;
- the sort REAL of real numbers.

6.3.1 Booleans

In general we can opt for two strategies: the first is to inductively define the booleans with constants $true$ and $false$ as constructors. This has the advantage that we can do induction on the possible values of a boolean. A disadvantage of this method is that we must do extra work to derive a contradiction when we have $(true = false)$ in the context. However this is a small price to pay. The second approach is to use the predefined type *Prop* that represents propositions. This has the disadvantage that we can not do induction in a straightforward way. Therefore we chose for the first approach.

```
Inductive Definition bool : Set = true : bool | false : bool.
```

6.3.2 Lists of booleans

Again we used a inductive type: a list is build from two constructors, $NIL : LIST$ and $CONS : BOOL \times LIST \rightarrow LIST$. Again the advantage of this approach is that we can do induction very easy.

```
Inductive Definition LIST : Set = NIL : LIST |
                                  CONS : bool -> LIST -> LIST.
```

6.3.3 Reals

As said before, the reals are modelled as rationals in this case study. This proved to be sufficient. A real is modelled as a cartesian product of two natural numbers. The natural numbers are predefined (inductively!) in Coq.

```
Definition REAL = (nat * nat).
```

6.3.4 Functions

We also have to define the functions on our data types. In general this can be done in two ways. In Coq it is possible to state axioms like $0 + x = x$. With this axiom we can rewrite terms using that axiom. The danger of this approach is that we can easily end up in a situation where every type is inhabited by assuming unsound axioms. Obviously in a situation like this inhabitance has become a trivial notion from which you never can conclude the correctness of a protocol.

For an example, if we want to code the integers as built from three constructors, O, S and P (for predecessor). Now we want that $S(P(O))$ and O denote the same integer.

```
Inductive Definition Z = O : Z | S : Z->Z | P : Z->Z.
```

If we then add the following proof term h to our context:

```
h : <Z>O=P(S(O))
```

Then we can find a predicate Q that has the following reduction behavior:

```
Q O        ==> (A:Prop)A->A
Q (P(S O)) ==> (A:Prop)A
```

and hence we can inhabit any proposition F as follows:

```
h Q [A:Prop][x:A]x F : F
```

So in order to use the inductive approach safely, without introducing contradictions, one needs to be sure that the introduced axioms are sound. In our case the used axioms of the datatypes are standard and generally accepted as sound.

The other option is to code all functions directly in lambda-calculus, using the naturals and ordering thereon that are given with Coq. This approach is much safer and is nearly always preferable, since all rewrites are $\beta\iota$-rewrites and therefore directly recognized by the system. The problem with this approach is that it can take a long time to formulate a certain function in λ-calculus, and the notation becomes less readable. We coded the functions on naturals, booleans and lists directly in lambda-calculus to take as much advantage of the increased speed of computation. In summary, we followed the following guidelines in our choices:
1. Represent all datatypes inductively with proper constructors;
2. Represent equations over the rationals with axioms;
3. Represent functions on the other data-types in lambda-calculus for maximum speed.

In this way we could do structural induction where we needed it and use the speed of $\beta\iota$-rewrites when we needed to compute a lot.

The protocol mentions three parameters:

- T, the tolerance for which the system functions correctly;
- Q, the size of the bit-time.
- MAX, the maximal length of a message.

In Coq these parameters were simply defined with the *Parameter* command that introduces the constants.

Many times we needed to solve inequalities over these parameters. To solve these we needed the axioms over the real numbers. If the protocol did not have these parameters, we could have represented all functions as λ-terms conveniently.

Now we will give some example definitions of the functions. For example, plus on the naturals:

```
Definition plus = [n,m:nat](<nat>Match n with m [p:nat]S).
```

The even function on naturals:

```
Definition even = [N:nat](<bool>Match N with
                  true
                  [n1:nat][b:bool](negb b)).
```

Negation on booleans:

```
Definition negb = [x:bool](<bool>Match x with false true).
```

Concatenation and reversal on lists:

```
Definition concat = [L1:LIST][L2:LIST](<LIST>Match L1 with
                    L2
                    [b:bool][l:LIST][rl:LIST]((CONS b rl))).
```

```
Definition reverse = [L1:LIST](<LIST>Match L1 with
                     NIL
                     [b:bool][l:LIST][rc:LIST](concat rc (CONS b NIL))).
```

In this way we have coded all necessary functions in type theory.

6.4 The transition systems

In [7] we see that the transition systems are defined using cartesian products over inductively defined sets. In general we followed the same approach.

6.4.1 Actions

The actions of a timed automaton S are defined as an inductively defined set. For an example, consider the set *act_S_R*:

```
Inductive Definition act_S_R =
   IN   :  LIST -> act_S_R  |
   OUT  :  LIST -> act_S_R  |
   TIME :  REAL -> act_S_R  |
   UP   :  act_S_R  |
   DOWN :  act_S_R  .
```

In a similar way we define *act_P*, the set of actions of the full protocol.

The state space is defined with cartesian products over the state variables. E.g., the parallel behavior is defined using a cartesian product:

```
Definition states_system = (states_S * states_R).
```

We have a function $ev : act\_S\_R \rightarrow act\_P$ that renames actions of $(S \parallel R)$ into the corresponding actions of P in such a way that internal actions are renamed into τ.

To reason easily about the state space we defined several projections on specific actions of the cartesian products. For instance,
`(p_t_S (p_S (S * R)))` gives us the *transmitting* variable of the *Sender* in $S \times R$:

```
Definition p_t_S = [s:states_S]
   <bool, bool * LIST * REAL * bool>Fst(s).

Definition p_S = [s:states_system]
   <states_S, states_R>Fst(s).
```

$Fst(s)$ is predefined as the first projection on a cartesian product.

6.4.2 Transitions

The transitions are defined inductively as a predicate over the actions and state space. We will give an example.

```
Inductive Definition step :
act_S_R -> states_system -> states_system -> Prop =
step_IN_1   : (m:LIST)(x:REAL)(e:bool)(lr:LIST)(xr:REAL)
            (le (length m) max) ->
            (<bool>(head m)=true) ->
            ((<bool>(odd (length m))=true) \/
             (<LIST>(last_two m)=(CONS false (CONS false NIL)))) ->

   (step (IN m)
     (st_system (st_S false false NIL x e) (st_R NIL xr))
     (st_system (st_S false false m (realc O (S O)) e) (st_R NIL xr)))
|
...
```

This shows an action IN. This action corresponds to the case where $\neg transmitting \wedge \neg wire\_high \wedge list = e$ is valid. In a similar way all other actions are defined. Since we have an inductive definition, we can proof goals with induction over *step*. This gives us a proof obligation for all possible steps, which is exactly what we wanted. The major disadvantage of the above approach is that by extracting the *if* and *case* constructions, we have a lot of goals to consider. Another possibility would have been to do the following:

```
Inductive Definition step :
act_S_R -> states_system -> states_system -> Prop =
step_IN  : (m:LIST)(s,s':states_system)
           (le (length m) max) ->
           (<bool>(head m)=true) ->
           ((<bool>(odd (length m))=true) \/
            (<LIST>(last_two m)=(CONS false (CONS false NIL))))->
           (If ((<bool>(p_t_S (p_S s))=false) /\
                (<bool>(p_w_S (p_S s)))=false) /\
                (<bool>(empty (p_l_S (p_S s)))=true) /\
                (<LIST>(p_l_S (p_S s'))=m) /\
                (<REAL>(p_x_S (p_S s'))=(realc O (S O)))
     (* then *) (step (IN m) s s')) |
...
```

This is an equivalent approach that introduces less but more complex subgoals. It is an open question whether the added complexity weighs more heavily than the more subgoals. It probably depends on the way the proof is structured which approach is best. We chose to do the first approach, that is filling in as much state variables as possible.

6.5 Simulation

To code the weak simulation property of the timed I/O-automata, we used a predicate over the state-space.

```
(
(s:states_system)(p:states_P)
(start s)->(start' p)->(simrel s p)
) /\ (* Start states should be related *)
(
(s,s':states_system)(p:states_P)(a:act_S_R)
(step a s s')->(reach s)->(living_sys s)->(simrel s p)->
(* related through step or... *)
  (
   <states_P>Ex([p':states_P]((step' (ev a) p p'))/\(simrel s' p'))
  ) \/
  (
   (living_sys s')->False (* ...dead... *)
  )
) /\
((s:states_system)(p:states_P)
 (accepting_sys s)->(simrel s p)->(accepting_P p)
 (* accepting states related *)
).
```

`(simrel s p)` is the simulation relation as defined in theorem 5.11. `Ex` is the existential quantifier as is predefined in Coq. If we must prove

```
<states_P>Ex([p':states_P]F(p'))
```

then we explicitly must give a witness `w` and after that a proof of `F(w)`. If we compare this with [7], we see that there the simulation relation was a function, and the existential quantifier could be dropped.

In a similar way we defined all other needed predicates as reachability, living states, start states, accepting states etc. In general we could code all aspects of the timed I/O-automata model we were interested in. For another example, look at the definition of living states:

```
Inductive Definition living_sys : states_system -> Prop =
lives_INIT : (s:states_system)(reach s)->(accepting_sys s)->
             (living_sys s) |
lives_step : (a:act_S_R)(s1,s2:states_system)
             (step a s1 s2)->(reach s1)->(living_sys s2)->
             (living_sys s1).
```

A living state is a reachable accepting state, or a reachable state from which we can reach a living state in one step. Compare this with the definition of reachability:

```
Inductive Definition reach : states_system -> Prop =
reach_INIT : (s:states_system)(start s)->(reach s) |
reach_STEP : (a:act_S_R)(s1,s2:states_system)
             (step a s1 s2) -> (reach s1) -> (reach s2).
```

We see the same inductive definition using the step predicate. To prove whether a certain state s is reachable or living, we first prove the initial formula and after that do an induction over all possible steps.

6.6 Invariants

Several invariants were necessary to proof the desired results. To code these, a similar approach as above was followed. Again we will give an example.

```
Lemma 5.1.
Assumes
(St:states_system)
(reach St)->
(<bool>(p_w_S (p_S St))=true)->(<bool>(p_t_S (p_S St))=true).
```

In general the invariants were more easy to prove than the simulation property. However some invariants consisted of large and complex propositions. Especially those concerning the real-time clocks were time consuming to check. It appears that much time can be saved if the invariants are as simple as possible. It takes probably less time to check a lot of small invariants like Lemma 5.1, which was checked whithin 30 minutes (human development + wall clock cpu time), than to check a very large one like lemma 5.7, which took more than a week to develop. This is mainly due to the addition of real numbers, parameters and inequalities: much time was spent dealing with those.

6.7 Session examples

We will show some sample session examples to illustrate the proof-checking process.

```
  (simrel s p)->
   (step a s s')->
    (reach s)->
     (living_sys s)->
      ((<states_P>Ex([p':states_P](step' (ev a) p p')/\(simrel s' p')))
       \/((living_sys s')->False))
  ============================
    H2 : (simrel s p)
    H1 : (living_sys s)
    H0 : (reach s)
    H : (step a s s')
    a : act_S_R
    p : states_P
    s' : states_system
    s : states_system
```

Here we see one of the crucial moments in checking the simulation: we proceed by doing induction over all possible steps with the command

```
Elim H; Intros.
```

After this we have to proof the above Goal for every possible action a. In our case there are 34 possible steps to consider. We will show one of them.

```
subgoal 1 is:
  (<states_P
   >Ex([p':states_P]
        (step' (ev (IN m)) p p')
        /\(simrel
             (st_system (st_S false false m (realc O (S O)) e)
                (st_R NIL xr))
             p')

   ))
  \/((living_sys
        (st_system (st_S false false m (realc O (S O)) e) (st_R NIL xr)))
     ->False)

  ============================
    H9 : (living_sys
            (st_system (st_S false false NIL x e) (st_R NIL xr)))
    H8 : (reach (st_system (st_S false false NIL x e) (st_R NIL xr)))
    H7 : (step (IN m)
            (st_system (st_S false false NIL x e) (st_R NIL xr))
            (st_system (st_S false false m (realc O (S O)) e)
               (st_R NIL xr)))
    H6 : (simrel (st_system (st_S false false NIL x e) (st_R NIL xr)) p)
    H5 : (<bool>(odd (length m))=true)
         \/(<LIST>(last_two m)=(CONS false (CONS false NIL)))
    H4 : <bool>(head m)=true
    H3 : (le (length m) max)
    xr : REAL
    lr : LIST
    e : bool
    x : REAL
    max : nat
    m : LIST
    p : states_P
```

Here we see that Coq has filled in the values of the state variables. H6 equals the induction hypothesis. The proof proceeds with choosing the left disjunct, giving the right state for p', doing induction over e and using induction. This is a simple subgoal that takes only 25 lines of tactics. More involved subgoals can take 275 lines of tactics to solve. On one line we generally put more than one tactic.

Another example shows an inequality.

```
Goal
 (X:REAL)
 (ler T
  (realc (S O) (S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S(S O)))))))))))))))))))->
 (ler X (multr (divr epT emT) (multr (realc (S (S O)) (S O)) Q)))->
 (ler (multr Q (realc (S (S (S O))) (S O))) X)->
 (ler X (multr Q (realc (S (S (S (S (S O))))) (S O))))->
 False.
```

This expresses the folllowing fact:

$$\begin{array}{l} \forall X, T : REAL. \\ \neg((T \leq 1/17)\wedge \\ (X \leq ((1+T)/(1-T)) * 2Q)\wedge \\ (3Q \leq X)\wedge \\ (X \leq 4Q)). \end{array}$$

To proof this we needed about 100 lines of tactics mainly consisting of rewriting subterms with the axioms on the real-numbers. These kinds of subgoals appeared very frequently. We think that a tool like mathematica would be more convenient to check such goals. Therefore we checked some of them and left the rest for further work.

6.8 Coq tactics

The command language of Coq consists apart from tactics dealing with goals themselves also of a few tactic functions to compose atomic tactics to composed tactics called *tacticals*.

The semi-colon ; in t1 ; t2 applies the tactic t2 to all subgoals generated by tactic t1. We solved lemma 5.1 with one tactical using this construct.

Another tactical is the *t1 Orelse t2* tactical that tries tactic t1, and if it fails, applies tactic t2. In this way we can compose very complicated tacticals to solve a lot of different subgoals with one tactical.

However we also could use additional tacticals. For example it would be very useful if we could write parametrized tacticals. Then we can make the tacticals more general, and better reusable.

Another point is the ability to look in your context with a tactic. For instance, if the term t is part of our context then apply tactic x with parameter t. In this way we could guide the proof much more easily.

Another point is the ability to save system states during a proof session. In Coq you can only save a state when you have finished a proof. However loading old proofs can take a long time. It would be convenient if the user could save his current state and when he wants can enter back.

7 Conclusions

On the account of finding an error in the proof one could say that the experiment was succesful: the main goal of proof-checking is to find errors in proofs. However we did not completely proof-check the new proof. This is mainly due to the great complexity of the protocol and the need to do many data manipulations, for which Coq seems not particularily suited. One can choose to check these algebraic theorems outside the theorem prover using an algebraic tool, e.g. maple. Another solution is to add preproven libraries and theories to the distribution, which has already been done in the latest version for Coq, but were not available at the time we used Coq.

Proof-checking with Coq We can distinguish two aspects here. On aspect deals with the ability to formulate all aspects of timed I/O-automata. This formed no problem using the rich and expressive language of Coq. The second aspect deals with the support for the proof-checking itself. In this respect we encountered limitations of the implementation of Coq we used. Especially when we had to deal with heavy data manipulations we felt the need for preproven libraries and theories.

Related work In [7] a very similar experiment was shown. Their conclusions confirm the view that data manipulations are troublesome and that the command language is not very strong. However their data structures did not include an uncountable set and inequalities with two unknowns.

In [6] LP (the Larch Prover) is used to check an algorithm stated in TLA (the Temporal Logic of Actions). We see here that the distinction between 1. the action logic 2. the temporal logic give reason to use different encodings for these. We see similar activities in our case: The action logic is directly coded in λ-calculus while many lemma's on the Real numbers are stated and used by axioms.

In [1] we see a verification of *Pbs*. Here the much use of inductional reasoning struck us as similar to our case. It also illustrates that more experience is needed to reach a situation where formal verification becomes cost effective.

In [9] the system *Isabelle* is used which is very similar to Coq. We also have λ-calculus and higher order logics to work with. We also see here that a theorem prover is not always especially suited to proof-check a realistic protocol.

Acknowledgements

Thanks to the following people : first and above all Alex Sellink who taught me type theory and Coq with an astonishing amount of patience and enthusiasm. Without him the article would not have been written at all. Secondly Jan Friso Groote who guided the project. Then Frits Vaandrager and Doeko Bosscher for joint work on [4], the starting point of the present paper. Then Jan Springintveld, Jaco van de Pol and Marc Bezem who helped in all sorts of ways. Thanks also to Philips Electronics for several discussions on the protocol and for giving access to confidential documentation.

References

[1] M. Agaard and M. Leeser. Verifying a logic synthesis tool in nuprl: a case study in software verification. In Bochmann and Probst [3], pages 69–81.

[2] R. Alur and D.L. Dill. The theory of timed automata. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Real-Time: Theory in Practice,* Mook, The Netherlands, June 1991, volume 600, pages 45–73, 1992.

[3] G. v. Bochmann and D.K. Probst, editors. *Proceedings of the 4th International Workshop on Computer Aided Verification,* Montreal, Canada, volume 663, 1992.

[4] D.J.B. Bosscher, I. Polak, and F.W. Vaandrager. Verification of an audio control protocol. In *Proceedings of the Third International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems,* Lübeck, Germany, September 1994. To appear.

[5] G. Dowek, A. Felty, H. Herbelin, G.P. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner. The Coq proof assistant user's guide. Version 5.8. Technical report, INRIA – Rocquencourt, May 1993.

[6] U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In Bochmann and Probst [3], pages 44–55.

[7] L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof-checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Proceedings Workshop Esprit BRA "Types for Proofs and Programs",* Nijmegen, The Netherlands, May 1993, number 806, 1994. Full version available as Report CS-R9420, CWI, Amsterdam, March 1994.

[8] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations – part II: Timing-based systems. Report CS-R9314, Amsterdam, March 1993.

[9] D. Mery and A. Mokkedem. Crocos: an integrated environment for interactive verification of sdl specifications. In Bochmann and Probst [3], pages 343–356.

[10] F.W. Vaandrager and N.A. Lynch. Action transducers and timed automata. In W.R. Cleaveland, editor, *Proceedings CONCUR 92,* Stony Brook, NY, USA, volume 630, pages 436–455, 1992.