Proof-Checking a Data Link Protocol*

L. Helmink[1], M.P.A. Sellink[2], F.W. Vaandrager[3]

[1] Philips Research Laboratories, helmink@prl.philips.nl
[2] Utrecht University, alex@phil.ruu.nl
[3] CWI and University of Amsterdam, fritsv@cwi.nl

Abstract. A data link protocol developed and used by Philips Electronics is modeled and verified using I/O automata theory. Correctness is computer-checked with the Coq proof development system.

Key words: Communication Protocols, I/O Automata, Proof-Checking, Protocol Verification, Type Theory.

1 Introduction

The data-link layer of a telecommunication protocol is verified and proof-checked. The protocol has been designed to communicate messages of arbitrary length over unreliable channels. The messages are transmitted in small packets or *frames*. The protocol does not rely on fairness of data transmission channels, i.e., repeated transmission of a frame does not guarantee its eventual arrival. For this reason, the number of retransmission attempts is limited and the protocol is called Bounded Retransmission Protocol.

Reliable communication protocols are vital to the telecommunication industry. They are also of increasing importance to the electronics business because more and more products consist of communicating subsystems and because many products integrate technology from the fields of computers, telecommunication devices, and consumer electronics. The pressure for reliability of the protocols involved poses an important challenge to verification techniques.

Design, implementation and testing of communication protocols is a complicated and error-prone activity. For many protocol-based products, erroneous protocol behavior is met by error-recovery procedures or by issuing a new software release. For some products however, error situations are not acceptable and software maintenance is impossible. Correctness of protocols is usually examined by careful testing of implementations.

Thorough testing increases confidence but testing is only semi-decidable: it may reveal the presence of errors but not the absence of errors. Protocol verification is required to obtain a higher degree of confidence. The protocol is modeled

* The work of the first author is partially supported by ESPRIT Basic Research Action 6453: 'Types for Proofs and Programs'. The work of the third author took place in the context of the ESPRIT Basic Research Action 7166: 'CONCUR2'.

in a mathematical structure and correctness is guaranteed by showing that the protocol satisfies the required behavior under all circumstances. Verification is not restricted to implementations but can also be applied to designs that have not yet been implemented. It should be stressed however that although verification excludes *design* errors, it cannot replace testing of implementations.

A hand-written protocol verification may itself contain certain errors that can be eliminated by computer tools. Verification errors can be classified into two categories: wrong assumptions and wrong deductions, corresponding to errors in the protocol model and to errors in its correctness proof, respectively. Errors of the first type are the responsibility of the modeler. Errors of the second type can be eliminated using computer tools for proof development or proof-checking. There is an additional advantage to the use of computer tools in protocol verification. Protocol verification is a labour-intensive and a non-trivial activity: much effort of skilled experts is required. With the current state-of-the-art, it is cost-effective only for those (parts of) protocols that are truly critical. Computer tools will enable more efficient verification of protocols.

In this paper we describe a verification and the associated proof-checking of a simplified and stylized version of a Philips telecommunication protocol. First, the protocol is proven correct using the input/output automaton model of Lynch and Tuttle [19], a formalism based on extended finite state machines. Next, the verification is proof-checked in type theory with the Coq system [9]. The objective of this work is twofold. The first objective is to prove correctness of the protocol with the highest possible level of confidence. The second objective of this work is to bring to light all technical issues that are involved in obtaining this result.

A starting point for the work described here was an algebraic specification of the protocol in PSF [22], a language based on process algebra. This specification was developed and validated using PSF simulation tools. The PSF description was translated into I/O automata theory and a suitable correctness criterion was defined. The protocol was verified by proving that it satisfies the correctness criterion. This specification and verification were then translated into type theory and checked with the Coq proof development system.

This paper is divided into the following parts: Section 2 gives an informal description of the protocol. Next, Section 3 explains the verification of the protocol. Section 4 discusses the proof-checking with the Coq system. Section 5 concludes with a discussion of the results.

2 Protocol Outline

Like most data link protocols, the Bounded Retransmission Protocol can be regarded as an extended version of the Alternating Bit Protocol. The protocol uses a stop-and-wait approach known as 'positive acknowledgement with retransmission' [27]: after transmission of a frame the sender waits for an acknowledgement before sending a new frame. The protocol procedures are similar to the LAPB link control procedures of the X.25 protocol [6] (for X.25 acknowledged mode and window size = 1, viz. one outstanding unacknowledged frame). Incoming frames are checked for errors. Correctly received frames are acknowledged while erroneous frames are simply discarded. If the acknowledgement fails to appear, the sender times out and retransmits the frame. An alternating bit is used to detect duplication of a frame. Real-time aspects are limited to the use of time-outs to detect loss of frames and loss of acknowledgements. Three service primitives are offered by the protocol: a request and confirm service at the sender side, and an indication service at the receiver side.

– $REQ(s)$
  The request service to transmit a finite list s of data. Each datum will be transferred in a separate message frame.

– $CONF(c)$ $\quad (c \in \{\mathsf{C\_OK}, \mathsf{C\_NOT\_OK}, \mathsf{C\_DONT\_KNOW}\})$
  The confirmation service that informs the sender about the result of a request.
  - $c = \mathsf{C\_OK}$: the request has been dispatched successfully.
  - $c = \mathsf{C\_NOT\_OK}$: the request has not been dispatched completely.
  - $c = \mathsf{C\_DONT\_KNOW}$: the request may or may not have been handled completely. This situation only occurs when the last frame is sent but not acknowledged.

– $IND(d, i)$ $\quad$ (d a datum and $i \in$ {I_FIRST, I_INCOMPLETE, I_OK})
  The indication service to pass a new frame to the receiver application.
  - $i =$ I_FIRST : the packet is the first one of a message; more data to follow.
  - $i =$ I_INCOMPLETE : the packet is an intermediate one; more data to follow.
  - $i =$ I_OK : the packet is the last one of a series, completing the transmission of a message.

– *IND_NOT_OK*
  The indication service to report loss of contact to the sender. Only part of a message has been received.

The protocol control procedures will be described by means of a sender S, a receiver R, and two communication channels K and L (Figure 1). We will assume that K and L are lossy channels: message frames are either lost or they arrive without corruption in the order in which they are sent. Messages can be communicated over ports *REQ, CONF, F, G, A, B, IND*. A data frame consists of a datum preceded by a header with three information bits named *first*, *last* and *toggle*: (*first*, *last*, *toggle*, *datum*). Bits *first* and *last* indicate if a packet is the first or last frame of a series, respectively. For a single-frame message both are set. *toggle* plays the role of alternating bit to distinguish between subsequent data frames. Acknowledgement frames consist of these three information bits only: (*first*, *last*, *toggle*).

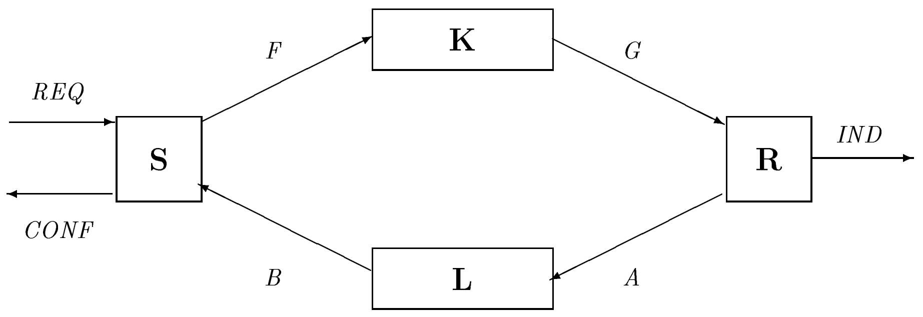

Fig. 1. Bounded Retransmission Protocol.

First consider a faultless transmission where no frames are lost. Suppose the sender S receives a request to transmit data $d_1 \ldots d_n$: $REQ(d_1 \ldots d_n)$. Here we will assume n>2; the cases n=1 or n=2 are similar. A frame $(\mathsf{true}, \mathsf{false}, \mathit{toggle}, d_1)$ is sent on port F. Channel K passes on the frame to receiver R over port G. R then issues an $\mathit{IND}(d_1, \mathsf{I\_FIRST})$ to port IND, and sends an acknowledgement frame $(\mathsf{true}, \mathsf{false}, \mathit{toggle})$ on port A, which is passed on by channel L to port B. The acknowledgement frame consists of the header of the data frame. Upon receipt of the acknowledgement, the sender transmits the second datum: $(\mathsf{false}, \mathsf{false}, \neg\mathit{toggle}, d_2)$, where *toggle* has flipped. The receiver issues $\mathit{IND}(d_2, \mathsf{I\_INCOMPLETE})$ and acknowledges the frame: $(\mathsf{false}, \mathsf{false}, \neg\mathit{toggle})$. This procedure is repeated until the last frame is sent with first=false, last=true, and datum=d_n. The receiver sends $\mathit{IND}(d_n, \mathsf{I\_OK})$ to report completion of the message and acknowledges receipt. The sender then informs the application of the successful dispatch of the transmission request with $\mathit{CONF}(\mathsf{C\_OK})$.

Now consider a transmission where data or acknowledgement frames are lost. First we take the sender's point of view. Upon sending a frame the sender S starts a timer t_1 and waits until either the frame is acknowledged or the timer goes off. If the acknowledgement is received, the timer is switched off and the next frame is sent. The timer is attuned to exceed the round trip time for sending a data frame and receipt of its acknowledgement. If the timer goes off no acknowledgement can come anymore and the frame is retransmitted.

The number of retransmission attempts is bounded by a parameter max, and if this maximum number of retransmissions has been reached, the sender gives up. The confirmation service is invoked in one of two ways: if the data frame in question is not the last frame of a series, then $\mathit{CONF}(\mathsf{C\_NOT\_OK})$ confirms failure of message transfer. For the last data frame, a $\mathit{CONF}(\mathsf{C\_DONT\_KNOW})$ is called: there is no way the sender can tell if the last frame was lost and never arrived, or if its acknowledgement was lost.

Finally consider the loss of frames from the receiver's point of view. Suppose a lost data frame is not the first one, i.e. the receiver is expecting a data frame follow-up. Upon receipt of a data frame, the receiver starts a timer t_2 and goes to a waiting state. When a data frame arrives it is acknowledged and timer t_2 is switched off. If the data frame has a flipped *toggle* then it is new and it is also indicated to the upper layers. When no data frame arrives, timer t_2 goes off eventually and service *IND_NOT_OK* is called. Timer t_2 will only go off if the sender has aborted the transmission, and therefore $t_2 > \mathsf{max}{*}t_1$.

3 Verification

3.1 I/O Automata Theory

In this section we give a brief account of those parts of I/O automata theory that we need for the purposes of the paper. For a more extensive introduction to the I/O automata model we refer to [19, 20].

I/O automata An *action signature* S is a triple $(\mathit{in}(S), \mathit{out}(S), \mathit{int}(S))$ of three disjoint sets of respectively *input actions*, *output actions* and *internal actions*. The derived sets of *external actions*, *locally controlled actions* and *actions* of S are defined respectively by

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

We say that S is *finite* if $\mathit{acts}(S)$ is a finite set.

An *I/O automaton* A consists of the following five components:

- an action signature $\mathit{sig}(A)$ (we will write $\mathit{in}(A)$ for $\mathit{in}(\mathit{sig}(A))$, $\mathit{out}(A)$ for $\mathit{out}(\mathit{sig}(A))$, etc.),
- a set $\mathit{states}(A)$ of *states*,
- a nonempty set $\mathit{start}(A) \subseteq \mathit{states}(A)$ of *start states*,
- a set $\mathit{steps}(A) \subseteq \mathit{states}(A) \times \mathit{acts}(A) \times \mathit{states}(A)$ of *transitions*, with the property that for every state s and input action a in $\mathit{in}(A)$ there is a transition (s, a, s') in $\mathit{steps}(A)$,
- an equivalence relation $\mathit{part}(A)$ on $\mathit{local}(A)$, having at most countably many equivalence classes.

We let s, s', u, u',.. range over states, and a,.. over actions. We write $s \xrightarrow{a}_A s'$, or just $s \xrightarrow{a} s'$ if A is clear from the context, as a shorthand for $(s, a, s') \in \mathit{steps}(A)$.

An action a is said to be *enabled* in a state s, if $s \xrightarrow{a} s'$ for some s'. Since every input action is enabled in every state, I/O automata are said to be *input enabled*. The intuition behind the input-enabling condition is that input actions are under control of the environment, and that the system that is modeled by an I/O automaton cannot prevent the environment from doing these actions. The partition $\mathit{part}(A)$ describes, what intuitively are the 'components' of the system, and will be used to define fairness.

Composition Intuitively, the composition of a collection of I/O automata is their Cartesian product, with the added requirement that automata synchronize the performance of shared actions. This synchronization models communication between system components: if a is an output action of A and an input action of B, then the simultaneous performance of a models communication from A to B. Since we do not want synchronization between output action of different I/O automata, or synchronizations involving internal actions, we require that the I/O automata are *compatible* in the sense that they do not share these actions.

Formally, we say that action signatures $S_1, \ldots, S_n$ are *compatible* if, for all $i, j \in \{1, \ldots, n\}$ satisfying $i \neq j$, $out(S_i) \cap out(S_j) = \emptyset$ and $int(S_i) \cap acts(S_j) = \emptyset$. We say that a number of I/O automata are *compatible* if their action signatures are compatible. The *composition* $S = \prod_{i=1}^{n} S_i$ of a finite collection of compatible action signatures $S_1, \ldots, S_n$ is defined to be the action signature with

- $in(S) = \bigcup_{i=1}^{n} in(S_i) - \bigcup_{i=1}^{n} out(S_i)$,
- $out(S) = \bigcup_{i=1}^{n} out(S_i)$,
- $int(S) = \bigcup_{i=1}^{n} int(S_i)$.

The *composition* $A = \|_{i=1}^{n} A_i$ of a finite collection of compatible I/O automata $A_1, \ldots, A_n$ is the I/O automaton defined as follows:

- $sig(A) = \prod_{i=1}^{n} sig(A_i)$,
- $states(A) = states(A_1) \times \cdots \times states(A_n)$,
- $start(A) = start(A_1) \times \cdots \times start(A_n)$,
- $steps(A)$ is the set of triples $(\mathbf{s}, a, \mathbf{s}')$ in $states(A) \times acts(A) \times states(A)$ such that, for all $1 \leq i \leq n$, if $a \in acts(A_i)$ then $\mathbf{s}[i] \xrightarrow{a}_{A_i} \mathbf{s}'[i]$ else $\mathbf{s}[i] = \mathbf{s}'[i]$,
- $part(A) = \bigcup_{i=1}^{n} part(A_i)$.

Notice that A is an I/O automaton indeed: $start(A)$ is nonempty because all the sets $start(A_i)$ are nonempty, A is input enabled because all the automata A_i are input enabled, and $part(A)$ is a partition of $local(A)$. We will sometimes write $A_1 \| \cdots \| A_n$ for $\|_{i=1}^{n} A_i$.

Hiding If S is an action signature and $I \subseteq out(S)$, then the action signature $\mathsf{HIDE}\ I\ \mathsf{IN}\ S$ is defined as the triple $(in(S), out(S) - I, int(S) \cup I)$. If A is an I/O automaton and $I \subseteq out(A)$, then $\mathsf{HIDE}\ I\ \mathsf{IN}\ A$ is the I/O automaton obtained from A by replacing $sig(A)$ by $\mathsf{HIDE}\ I\ \mathsf{IN}\ sig(A)$, and leaving all the other components unchanged.

Traces and fair traces Let A be an I/O automaton. An *execution fragment* of A is a finite or infinite alternating sequence $s_0 a_1 s_1 a_2 s_2 \cdots$ 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, $s_i \xrightarrow{a_{i+1}} s_{i+1}$. An *execution* of A is an execution fragment that begins with a start state. A state s of A is *reachable* if it is the final state of some finite execution of A.

Suppose $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ is an execution fragment of A. Then the *trace* of α is the subsequence of $a_1 a_2 \cdots$ consisting of the external actions of A. With *traces*$^*(A)$ we denote the set of traces of finite executions of A. For s, s' states of A and β a finite sequence of external actions of A, we define $s \overset{\beta}{\Longrightarrow}_A s'$ iff A has a finite execution fragment with first state s, last state s' and trace β.

A *fair execution* of an I/O automaton A is defined to be an execution α of A such that the following conditions hold for each class C of *part*(A):

1. If α is finite, then no action of C is enabled in the final state of α.
2. If α is infinite, then either α contains infinitely many occurrences of actions from C, or α contains infinitely many occurrences of states in which no action from C is enabled.

This says that a fair execution gives fair turns to each class of *part*(A), and therefore to each component of the system being modeled. A state of A is said to be *quiescent* if only input actions are enabled in this state. Intuitively, in a quiescent state the system is waiting for an input from the environment. A finite execution is fair if and only if its final state is quiescent. We denote the set of traces of fair executions of A by *fairtraces*(A). Also, we write *qtraces*(A) for the set of traces of finite fair executions of A.

Safety, deadlock freeness and implementation Let A and B be I/O automata with the same input and output actions, respectively. Then we say that

– A is *safe* with respect to B iff *traces*$^*(A) \subseteq$ *traces*$^*(B)$,
– A is *deadlock free* with respect to B iff *qtraces*$(A) \subseteq$ *qtraces*(B),
– A *implements* B iff *fairtraces*$(A) \subseteq$ *fairtraces*(B).

If A is safe with respect to B then all finite behaviors of A are allowed by B. Thus A may still have behaviors that are not allowed by B, but these are all infinite, and so it cannot be concluded from a finite observation that A violates the requirements imposed by B.

Because in I/O automata input actions are always enabled, they will typically not have deadlocks in the sense of states without any outgoing transitions. Instead we define deadlock freeness as a relation between I/O automata. If A is deadlock free with respect to B, this means that whenever it is possible to reach a quiescent state of A via some trace, we can also reach a quiescent state of B with the same trace. Thus A can only become inactive when this is allowed by B.

In I/O automata theory, inclusion of fair traces is commonly used as implementation relation. Intuitively, one may think of B as defining a set of constraints, which A must obey. Note that A does not need exhibit *all* of the behaviors in *fairtraces*(B); merely a subset is sufficient. However, by requiring that A and B have the same input actions, and since input actions must always be enabled, trivial implementations are excluded. Here it is important to note that the concept of fairness used within the I/O automata model is *feasible* in the sense of [1]: each finite execution of an I/O automaton can be extended to a fair execution (for instance, by giving turns in a roundrobin way to all classes that are continuously enabled). As a consequence it also follows that A implements

B implies that A is safe with respect to B. In general, A implements B does not imply that A is deadlock free with respect to B. This is because a quiescent execution of A may be matched by a *divergent* fair execution of B, i.e., an infinite execution in which after some point only internal actions occur. However, it is easy to see that the implication does hold if B is *divergence free* in the sense that it has no divergent fair executions.

Refinements In the literature, a whole menagerie of so-called simulation techniques has been proposed to prove that the set of (finite, quiescent, fair,...) traces of one automaton is included in that of another. We refer to [21] for an overview and for further references. In this paper we only need a very simple type of simulation, which is called *weak refinement*.

Let A and B be I/O automata with the same input and output actions, respectively. A *weak refinement* from A to B is a function r from $states(A)$ to $states(B)$ such that:

1. If $s \in start(A)$ then $r(s) \in start(B)$.
2. If s is a reachable state of A and $s \xrightarrow{a}_A s'$, then $r(s) \overset{\beta}{\Longrightarrow}_B r(s')$ where β equals a if $a \in ext(A)$, and is empty otherwise.

Lemma 1. *If there exists a weak refinement from A to B then A is safe with respect to B.*

The converse implication does not hold, i.e. there exist I/O automata A and B such that A is safe with respect to B, but no weak refinement from A to B can be given. In those cases one has to use other, more general simulations. Also, if there exists a weak refinement from A to B then it is not in general the case that A is deadlock free with respect to B, or that A implements B. However, in the protocol that we analyze in this paper we will establish a weak refinement that maps quiescent executions to quiescent executions, and fair executions to fair executions, and these additional properties immediately imply absence of deadlock and the implementation relation.

The precondition/effect style In the I/O automata approach, the automata that model the basic building blocks of a system are usually specified in the so-called *precondition/effect* style. In this section we will briefly describe the syntax of this language.

We start from a typed signature Σ together with a Σ-algebra $\mathcal{A}$ which gives meaning to the function and constant symbols in Σ. To describe properties, we use a first-order language over signature Σ and a set V of (typed) variables, with equality and inequality predicates, and the usual logical connectives. If ξ is a valuation of variables in their domains, and b is a formula, then we write $\mathcal{A}, \xi \models b$ if b holds in $\mathcal{A}$ under valuation ξ.

An *I/O automaton generator* G consists of five components:

– a finite action signature $sig(G)$,

– a finite set *vars*(G) of (typed) *state variables*,
– a satisfiable formula *init*(G), in which variables from *vars*(G) may occur free,
– for each action $a \in acts(G)$, a *transition type*, i.e., an expression of the form

```
$a(y_1, \ldots, y_n)$
    **Precondition**:
        $b$
    **Effect**:
        $x_1 := e_1$
            $\vdots$
        $x_m := e_m$
```

where the y_i are (typed) variables, b is a formula in which variables from $vars(G) \cup \{y_1, \ldots, y_n\}$ may occur free, and which is `true` if $a \in in(G)$, $vars(G) = \{x_1, \ldots, x_m\}$, and the e_j are expressions with the same type as x_j, in which the variables $vars(G) \cup \{y_1, \ldots, y_n\}$ may occur,
– an equivalence relation *part*(A) on *local*(G).

Each I/O automaton generator G denotes an I/O automaton A in the obvious way: states of A are interpretations of the variables of *vars*(G) in their domains; start states of A are those states that satisfy formula *init*(G); for each (input, output or internal) action $a \in acts(G)$ with a transition type as above, and for each choice of values $v_1, \ldots, v_n$ taken from the domains of $y_1, \ldots, y_n$, respectively, A contains an (input, output or internal) action $a(v_1, \ldots, v_n)$; A has a transition

$$s \xrightarrow{a(v_1,\ldots,v_n)} s'$$

iff there exists a valuation ξ such that

– for all $x \in vars(G)$, $\xi(x) = s(x)$,
– for $1 \leq i \leq n$, $\xi(y_i) = v_i$,
– $\mathcal{A}, \xi \models b$, and
– for $1 \leq j \leq m$, e_j evaluates to $s'(x_j)$ under ξ;

part(G) trivially induces a partition on *local*(A).

Let $a \in acts(G)$ be an action with a transition type as above. We define the formulas *enabled*(a) and *quiescent*(G) by

$$enabled(a) \triangleq \exists y_1, \ldots, y_n \,.\, b$$
$$quiescent(G) \triangleq \bigwedge_{a \in acts(G)} \neg enabled(a)$$

It follows that a state s of the automaton associated to G is quiescent iff it satisfies formula *quiescent*(G).

The reader will observe that the translation from I/O automata generators to I/O automata is quite straightforward. In fact, Lynch and Tuttle [19, 20] do not even bother to distinguish between these two levels of description. For the formalization of I/O automata theory in Coq the distinction between the semantic and syntactic levels is of course important, which is why we have discussed it here. The definition of I/O automata generators has been inspired by similar definitions in the work of Jonsson (see, for instance, [15]). In the sequel we will, like Lynch and Tuttle, often refer to I/O automata when we actually mean I/O automata generators.

3.2 Protocol Specification

In this section, we present the formal specification of the Bounded Retransmission Protocol. Following a brief description of the many-sorted algebra that we use, we will first give I/O automata for each of the components of the protocol, and then define the full protocol as the composition of these I/O automata. At the end of this section we will moreover present the definition of an I/O automaton that gives the intended external behavior of the protocol. Since the BRP protocol has been explained already in considerable detail in Section 2, we will not repeat that explanation here, and confine ourselves in this section to the formal definitions, together with a brief discussion of some of the notation and certain modeling assumptions.

Data types We start the specification of the protocol with a description of the various data types that play a role. We assume a typed signature Σ and a Σ-algebra $\mathcal{A}$ which consist of the following components:

– a type **Bool** of booleans with constant symbols true and false, and a standard repertoire of function symbols ($\wedge$, $\vee$, $\neg$, $\rightarrow$), all with the standard interpretation over the booleans. Also, we require, for all types **S** in Σ, an equality, inequality, and if-then-else function symbol, with the usual interpretation:

$$\begin{aligned} .=. &: \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ .\neq. &: \mathbf{S}\times\mathbf{S} \rightarrow \mathbf{Bool} \\ \mathsf{if}\ .\ \mathsf{then}\ .\ \mathsf{else}\ . &: \mathbf{Bool}\times\mathbf{S}\times\mathbf{S} \rightarrow \mathbf{S} \end{aligned}$$

  Note the (harmless) overloading of the constants and function symbols of type **Bool** with the propositional connectives used in formulas. We will frequently view boolean valued expressions as formulas, i.e., we use b as an abbreviation of b=true.
– a type **Nat** of natural numbers, with constant symbol 0, successor function symbol succ, and function symbol $\leq$: $\mathbf{Nat}\times\mathbf{Nat} \rightarrow \mathbf{Bool}$, all with the usual interpretation. We also need a constant symbol max, which denotes the maximum number of retransmissions within the protocol.
– a type **Data** of data elements that the protocol has to transmit. We find it convenient to assume the presence of a constant symbol $\perp$ of type **Data**, which denotes the *undefined* data element.

- a type **List** of finite lists over the domain of **Data**, with a constant symbol ϵ, denoting the empty list, and a function symbol $\mathsf{add} : \mathbf{Data} \times \mathbf{List} \rightarrow \mathbf{List}$, denoting the operation of prefixing a list with a data element. Besides these constructors, there are function symbols $\mathsf{hd} : \mathbf{List} \rightarrow \mathbf{Data}$, $\mathsf{tl} : \mathbf{List} \rightarrow \mathbf{List}$, and $\mathsf{one} : \mathbf{List} \rightarrow \mathbf{Bool}$. hd takes the first element of a list, tl returns the remainder of a list after removal of the first element, and one returns true iff the argument list has length one. These operations are fully characterized by the axioms (where s is a variable of type **List**, and d, e are variables of type **Data**):

$$\begin{array}{ll} \mathsf{hd}(\epsilon) = \bot & \mathsf{one}(\epsilon) = \mathsf{false} \\ \mathsf{hd}(\mathsf{add}(d, s)) = d & \mathsf{one}(\mathsf{add}(d, \epsilon)) = \mathsf{true} \\ \mathsf{tl}(\epsilon) = \epsilon & \mathsf{one}(\mathsf{add}(d, \mathsf{add}(e, s))) = \mathsf{false} \\ \mathsf{tl}(\mathsf{add}(d, s)) = s & \end{array}$$

- a type **Conf** of *confirmation messages*, with constant symbols C_NOT_OK, C_OK and C_DONT_KNOW.
- a type **Ind** of *indication messages*, with constant symbols I_OK, I_FIRST and I_INCOMPLETE.
- a type **Spc** of *program counter values of the sender*, with constant symbols SF, WA, SC, ET2 and WT2.
- a type **Rpc** of *program counter values of the receiver*, with constant symbols WF, SI, SA, RTS and NOK.

The intended meaning of all these constants will be explained further on in this section. We assume that the interpretation of **Conf**, **Ind**, **Spc** and **Rpc** is free, in the sense that, for each of these types, different constants symbols are mapped to different elements in their domain ("no confusion"), and each element in the domain is denoted by some constant symbol ("no junk").

Notation In the presentation below, we use the following conventions:

- We omit the precondition of an input action (since this equals true by definition).
- In the effect part of transition types we omit assignments of the form $x := x$.
- We write if c then $[z_1 := f_1, \ldots, z_k := f_k]$ as an abbreviation for

$$\begin{array}{c} z_1 := \mathsf{if}\ c\ \mathsf{then}\ f_1\ \mathsf{else}\ z_1 \\ \vdots \\ z_k := \mathsf{if}\ c\ \mathsf{then}\ f_k\ \mathsf{else}\ z_k \end{array}$$

- We never mention the partition of the local action types because in all I/O automata generators that we consider it is trivial in the sense that there is just only block which contains all the locally controlled actions. (Note that the composition of the I/O automata denoted by these generators will not have a trivial partition!)

- We write $pc \in \{\mathsf{SF}, \mathsf{WA}, \mathsf{SC}\}$ for $pc{=}\mathsf{SF} \vee pc{=}\mathsf{WA} \vee pc{=}\mathsf{SC}$, etc.
- To improve readability we sometimes use Lamport's list notation for conjunction.

The sender We will now present the I/O automaton S, which models the sender of the protocol. An important state variable of S is *pc*, which gives the current value of the program counter of the sender. This variable, which is of type **Spc**, may have five different values:

- SF: Send a Frame at port F,
- WA: Wait for an Acknowledgement to arrive at port B,
- SC: Send a Confirmation message to the upper layer,
- ET2: Enable Timer 2, and
- WT2: Wait for Timeout of Timer 2.

We have modeled the arrival of a request (*REQ*) as an input action, since it is clearly under control of the environment. However, once we have taken this decision the I/O automata model forces us to specify, for all possible states, what happens if an *REQ* action occurs. In our modeling, the sender discards an incoming request if it is busy handling the previous request, something which is recorded by the boolean state variable *busy*.

T3 is a time out action that occurs when S wants to send a frame into channel K but does not succeed because other agents (not specified here) are using the channel. After the occurrence of a *T3* action, S will send a confirmation message C_DONT_KNOW or C_NOT_OK.

When S sends a frame into channel K by doing F, it simultaneously starts a timer by setting boolean state variable *timer1_on* to true. This timer will timeout if an acknowledgement for the frame does not arrive in time. Since we cannot explicitly model real-time aspects in the I/O automata model, we deal with this timing behavior in a different way. Under the assumptions that (1) the transmission of a frame through channels K and L takes a bounded time, and (2) R will always acknowledge an incoming frame in a bounded time, and (3) the timer is set properly, a timeout will occur iff a frame gets lost in channel K or in channel L. Thus one could say that the loss of a message in the channel "causes" a timeout action. In our specification we have made these causal links visible by introducing output actions *E1K* and *E1L* for channels K and L, respectively, which occur when a message gets lost, and corresponding input actions *E1K* and *E1L* of sender S, whose occurrence sets a boolean state variable *timer1_enabled*. By taking *timer1_enabled* to be part of the precondition of the timeout action *T1*, this gives us the desired causal links.

If something goes wrong during the handling of a request, and S sends a C_DONT_KNOW or C_NOT_OK confirmation message, then before dealing with a new request, S will wait long enough to make sure that the receiver R is prepared to receive new frames. Also here, since we cannot deal with real-time directly within our model, we describe the causal links that result from these real-time constraints. After sending a C_DONT_KNOW or C_NOT_OK confirmation message, the sender does an output action *E2*, which corresponds to starting a new timer (that is not specified here). Since it depends on the state of R when this timer will timeout, *E2* is made into an input action of R. At the appropriate moment R will generate the timeout action *T2* for the timer started by S, so that S can proceed and handle the next request.

We now give the code for I/O automaton S.

```
Input:    REQ, B, E1K, E1L, T2
Output:   CONF, F, E2
Internal: T1, T3
State Variables: pc:                   Spc    Initialization: ∧ pc=SF
                 busy,first,toggle:    Bool                   ∧ ¬busy
                 list:                 List                   ∧ first
                 timer1_on:            Bool                   ∧ ¬timer1_on
                 timer1_enabled:       Bool                   ∧ ¬timer1_enabled
                 rn:                   Nat                    ∧ rn=0

REQ(s : List)
  Effect:
    if ¬busy ∧ s≠ε then  [list := s
                          busy := true]

F(f : Bool, l : Bool, t : Bool, d : Data)                  T3
  Precondition:                                              Precondition:
    pc=SF ∧ busy ∧                                             pc=SF ∧ busy
    f=first ∧ l=one(list) ∧ t=toggle ∧ d=hd(list)            Effect:
  Effect:                                                      pc := SC
    pc := WA
    timer1_on := true
    rn := succ(rn)
E1K                                                 E1L
  Effect:                                             Effect:
    timer1_enabled := true                              timer1_enabled := true
B(f : Bool, l : Bool, t : Bool)          T1
  Effect:                                  Precondition:
    pc := if one(list) then SC else SF       timer1_on ∧ timer1_enabled
    first := one(list)                     Effect:
    toggle := ¬toggle                        pc := if rn≤max then SF else SC
    timer1_on := false                       timer1_on := false
    list := tl(list)                         timer1_enabled := false
    if ¬(one(list)) then  [rn := 0]

CONF(c : Conf)
  Precondition:
    pc=SC ∧ c=if list=ε then C_OK else
                   if one(list) ∧ rn≠0 then C_DONT_KNOW else C_NOT_OK
  Effect:
    pc := if list=ε then SF else ET2
    busy := false
    list := ε
    rn := 0
```

```
E2                                          T2
  Precondition:                               Effect:
    $pc$=ET2                                    $pc$ := SF
  Effect:
    $pc$ := WT2
    $first$ := true
    $toggle$ := $\neg toggle$
```

Channel *K* I/O automaton K models in a straightforward way the behavior of a faulty message buffer with input channel F, output channel G, and capacity one. Messages that arrive when the buffer is full are discarded. In Lemma 2 we will show that actually such a situation never occurs during a run of the protocol.

Input: F
Output: G, $E1K$
State Variables: *full,first,last,toggle*: **Bool**   **Initialization**: $\neg full$
datum: **Data**

```
$F(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool}, d : \mathbf{Data})$
  Effect:
    if $\neg full$ then [$full$ := true
                         $first$ := $f$
                         $last$ := $l$
                         $toggle$ := $t$
                         $datum$ := $d$]

$G(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool}, d : \mathbf{Data})$          E1K
  Precondition:                                                 Precondition:
    $full \wedge f=first \wedge l=last \wedge t=toggle \wedge d=datum$          $full$
  Effect:                                                       Effect:
    $full$ := false                                               $full$ := false
```

Channel *L* I/O automaton L is the same as K, except that L handles frames that consist of 3 instead of 4 fields, and its actions have different names.

Input: A
Output: B, $E1L$
State Variables: *full,first,last,toggle*: **Bool**   **Initialization**: $\neg full$

```
$A(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool})$
  Effect:
    if $\neg full$ then [$full$ := true
                         $first$ := $f$
                         $last$ := $l$
                         $toggle$ := $t$]

$B(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool})$                          E1L
  Precondition:                                                 Precondition:
    $full \wedge f=first \wedge l=last \wedge t=toggle$                       $full$
  Effect:                                                       Effect:
    $full$ := false                                               $full$ := false
```

The receiver The most important state variable of I/O automaton R is pc, which gives the value of the program counter of the receiver. This variable, which is of type **Rpc**, can have five possible values:

- WF: Wait for a Frame to arrive at port G,
- SI: Send an Indication message to the upper layer,
- SA: Send an Acknowledgement message at port A,
- RTS: Return control bits of received frame To Sender via port A, and
- NOK: send an indication message NOT_OK to the upper layer.

The subtle part in the definition of R is again the part concerned with timing. The receiver has a timer of its own, which is started at the moment an acknowledgement message is sent by setting a boolean variable $timer2\_on$ to true. The timer will time out if after some time still no new frame has arrived at port G and it is clear that the sender has interrupted the transmission a list. When a timeout occurs, the receiver sets $ctoggle$ to false to indicate that it will not reject the next frame on basis of its toggle bit, and it generates an indication NOT_OK in case some messages have not yet been received. If R has set the timer and S generates an $E2$ action, then a transmission has been interrupted and a timeout action may occur. For convenience we identify in our model $E2$ with the timeout action. However, if an $E2$ action occurs and the receiver's timer has not been set, then this action should not be interpreted as a timeout, but just as a signal that an action $T2$ can be generated at the sender side.

We now present the code for I/O automaton R.

```
Input:  $G, E2$
Output: $A, IND, IND\_NOT\_OK, T2$
State Variables: $pc$:                       Rpc    Initialization: $\wedge\ pc$=WF
                 $first, toggle, ctoggle$:   Bool                   $\wedge\ first$
                 $ffirst, flast, ftoggle$:   Bool                   $\wedge\ \neg ctoggle$
                 $fdatum$:                   Data                   $\wedge\ \neg timer2\_on$
                 $timer2\_on$:               Bool                   $\wedge\ \neg timer2\_enabled$
                 $timer2\_enabled$:          Bool

$G(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool}, d : \mathbf{Data})$
  Effect:
    if $pc$=WF then  [$pc$ := if $ctoggle \rightarrow t=toggle$ then SI else RTS
                     $ffirst := f$
                     $flast := l$
                     $ftoggle := t$
                     $fdatum := d$
                     if $ctoggle \rightarrow t=toggle$ then  [$timer2\_on$ := false] ]

$IND(d : \mathbf{Data}, i : \mathbf{Ind})$
  Precondition:
    $pc$=SI $\wedge\ d=fdatum$
    $\wedge\ i$=if $flast$ then I_OK else (if $ffirst$ then I_FIRST else I_INCOMPLETE)
  Effect:
    $pc$ := SA
    $first := flast$
    $ctoggle :=$ true
    $toggle := \neg ftoggle$
```

```
$A(f : \mathbf{Bool}, l : \mathbf{Bool}, t : \mathbf{Bool})$
  **Precondition**:
    $pc \in \{\mathsf{SA}, \mathsf{RTS}\} \wedge f{=}ffirst \wedge l{=}flast \wedge t{=}ftoggle$
  **Effect**:
    if $pc{=}\mathsf{SA}$ then [$timer2\_on :=$ true]
    $pc :=$ WF
```

```
$IND\_NOT\_OK$
  **Precondition**:
    $pc{=}$NOK
  **Effect**:
    $pc :=$ WF
    $first :=$ true
    $timer2\_on :=$ true
```

```
$E2$
  **Effect**:
    $timer2\_enabled :=$ true
    if $timer2\_on$ then [$ctoggle :=$ false
                          if $\neg first$ then [$pc :=$ NOK
                                                $timer2\_on :=$ false] ]
```

```
$T2$
  **Precondition**:
    $timer2\_enabled \wedge pc{=}$WF
  **Effect**:
    $timer2\_enabled :=$ false
```

The full protocol I/O automaton *BRP* is defined as the parallel composition of I/O automata S, K, L and R, with all communication between these components hidden:

$$BRP \triangleq \mathsf{HIDE}\ I\ \mathsf{IN}\ (S \| K \| L \| R)$$

where $I \triangleq \{F(f,l,t,d), G(f,l,t,d), A(f,l,t), B(f,l,t), E1K, E1L, E2, T2$
$\mid f, l, t$ in domain $\mathbf{Bool}, d$ in domain $\mathbf{Data}\}$.

The correctness criterion We specify the collection of allowed behaviors of the Bounded Retransmission Protocol via an I/O automaton P, which has the same input and output actions as *BRP*, but no internal actions. If a $REQ(s)$ action occurs in the initial state, then the regular behavior of P is to output the elements of s one by one, tagging the first datum with an indication I_FIRST, intermediate data with I_INCOMPLETE, and the last datum with I_OK. After sending the last datum P generates a confirmation message C_OK to indicate that the request has been carried out successfully, and return to its initial state. Requests that arrive at a time when the previous request has not yet been processed are ignored. While a request is being processed, something may go wrong at any point and, instead of the C_OK message a C_DONT_KNOW or a C_NOT_OK confirmation message may be sent. The C_DONT_KNOW message will only occur, however, if at most one data element has not been delivered, and the C_NOT_OK will only occur if at least one data element has not been delivered. If a C_NOT_OK or C_DONT_KNOW message is sent somewhere in the middle of the processing of a request, i.e., after the first but before the last data element has been delivered, P generates a NOT_OK message. After such a message P returns to its initial state, except if it has just received a new request, which will then be processed.

Below we present the code of I/O automaton P. In the next section we will establish that BRP is an implementation of P.

```
Input:   $REQ$
Output:  $IND$, $IND\_NOT\_OK$, $CONF$
State Variables: $busy, first, error$:   Bool        Initialization: $\wedge \neg busy$
                 $list$:                 List                        $\wedge\ first$
                                                                     $\wedge \neg error$

$REQ(s : \mathbf{List})$
    Effect:
        if $\neg busy \wedge s \neq \epsilon$ then  [$busy$ := true
                                  $list := s$]

$IND(d : \mathbf{Data}, i : \mathbf{Ind})$
    Precondition:
        $busy \wedge \neg error \wedge list \neq \epsilon \wedge d = \mathsf{hd}(list)$
        $\wedge\ i =$ if one($list$) then I_OK else (if $first$ then I_FIRST else I_INCOMPLETE)
    Effect:
        $first$ := one($list$)
        $list$ := tl($list$)

$CONF(c : \mathbf{Conf})$                                         $IND\_NOT\_OK$
    Precondition:                                               Precondition:
        $busy \wedge \neg error$                                     $error$
        $\wedge\ (c = \mathsf{C\_OK} \rightarrow list = \epsilon)$                       Effect:
        $\wedge\ (c = \mathsf{C\_DONT\_KNOW} \rightarrow (list = \epsilon \vee \mathsf{one}(list)))$      $first$ := true
        $\wedge\ (c = \mathsf{C\_NOT\_OK} \rightarrow list \neq \epsilon)$                     $error$ := false
    Effect:
        $busy$ := false
        $error := \neg first$
        $list := \epsilon$
```

3.3 Protocol Correctness Proof

Invariants In order to establish a weak refinement from BRP to P we must first gain insight into what are the reachable states of BRP. To this end, we present a number of *invariants* of the protocol, i.e., properties that are valid for all reachable states. Most of these invariants are proved by a routine induction on the length of the executions to the reachable states. The full, handwritten proofs of the invariants together occupy about 16 pages of ASCII text. We used numbering of assertions, as advocated by Lamport [16], although, due to the fact that the proofs went rarely more than 4 levels deep, we found it easier to use explicit names, like 3.1.1.1, instead of the implicit ones, like $\langle 4 \rangle 1$. As an illustration we have included the full proof of the invariant *INVR* (Lemma 3). In order to distinguish between the state variables of different components of *BRP*, we prefix each state variable by the name of the component it originates from.

Lemma 2. *The following property INV1 is an invariant of BRP.*

$$
\begin{array}{lll}
\wedge & S.pc \in \{\mathsf{SF}, \mathsf{SC}, \mathsf{ET2}\} & \rightarrow & R.pc = \mathsf{WF} \\
\wedge & S.timer1\_enabled & \rightarrow & S.pc = \mathsf{WA} \wedge R.pc = \mathsf{WF} \wedge \neg K.full \wedge \neg L.full \\
\wedge & K.full & \rightarrow & S.pc = \mathsf{WA} \wedge R.pc = \mathsf{WF} \wedge \neg L.full \\
\wedge & R.pc \in \{\mathsf{SI}, \mathsf{SA}, \mathsf{RTS}\} & \rightarrow & S.pc = \mathsf{WA} \\
\wedge & R.timer2\_enabled & \rightarrow & S.pc = \mathsf{WT2} \wedge R.pc \in \{\mathsf{WF}, \mathsf{NOK}\} \wedge \\
 & & & \neg K.full \wedge \neg L.full \\
\wedge & L.full & \rightarrow & S.pc = \mathsf{WA} \wedge R.pc = \mathsf{WF} \wedge \neg K.full
\end{array}
$$

Invariant *INV1* relates the control variables of the different components of the protocol. The invariant already allows us to make several important observations on the behavior of the protocol. The third clause implies that sender S will never send a frame into channel K when the channel is busy delivering another frame. Similarly, receiver R will never send a frame into channel L when L already contains a frame. Thus the protocol does not need communication channels with a buffering capacity of more than one. Clause three and six together give that there will never be a message in both K and L at the same time. Thus, an implementation of the protocol may use a single bidirectional medium to implement both channels. If channel L delivers a frame to the sender S, then S is in fact waiting for this frame to arrive. Similarly, if channel K delivers a frame to receiver R, then the receiver is waiting for this frame. It follows rather directly from invariant *INV1* that in each reachable state of the protocol at most one of the four components enables a locally controlled action. This means that the protocol operates in a is fully sequential way.

Invariants *INVR* of Lemma 3 gives some relationships between the state variables of R.

Lemma 3. *The following property INVR is an invariant of BRP.*

$$
\begin{array}{llll}
\wedge & R.pc = \mathsf{NOK} & \rightarrow & \neg R.ctoggle \\
\wedge & R.pc = \mathsf{SI} & \rightarrow & R.ctoggle \rightarrow R.ftoggle = R.toggle \\
\wedge & R.pc \in \{\mathsf{RTS}, \mathsf{SA}\} & \rightarrow & R.ctoggle \wedge R.ftoggle \neq R.toggle
\end{array}
$$

Proof. Let s' be a reachable state of *BRP*. By induction on the length n of the shortest execution of *BRP* that ends in s', we prove $s' \models INVR$. If $n = 0$, then s' is a start state. Hence $s' \models R.pc = \mathsf{WF}$, which implies $s' \models INVR$.

For the induction step, suppose that s' is reachable via an execution with length $n + 1$. Then there exists a state s that is reachable via an execution of length n and $s \xrightarrow{a} s'$, for some action a. By induction hypothesis, $s \models INVR$. We prove $s' \models INVR$ by a routine case distinction on a. In the proof we will use several times that, by Lemma 2, $s \models INV1$.

1. Assume a is an action in which R does not participate. Then $s' \models \mathit{INVR}$ trivially follows from $s \models \mathit{INVR}$ and the observation that a does not change any of the state variables mentioned in INVR.
2. Assume $a = G(f,l,t,d)$
   2.1) $s \models K.\mathit{full}$ (by 2 and precondition G)
   2.2) $s \models R.\mathit{pc}{=}\mathsf{WF}$ (by 2.1 and $\mathit{INV1}$)
   2.3) Assume $s \models R.\mathit{ctoggle} \rightarrow t{=}R.\mathit{toggle}$
   2.3.1) $s' \models R.\mathit{ctoggle} \rightarrow t{=}R.\mathit{toggle}$ (by 2 and 2.3 since G does not change $R.\mathit{ctoggle}$ and $R.\mathit{toggle}$)
   2.3.2) $s' \models R.\mathit{pc}{=}\mathsf{SI} \wedge R.\mathit{ftoggle}{=}t$ (by $2, 2.2, 2.3$ and effect G)
   2.3.3) $s' \models \mathit{INVR}$ (by 2.3.1 and 2.3.2)
   2.4) Assume $s \not\models R.\mathit{ctoggle} \rightarrow t{=}R.\mathit{toggle}$
   2.4.1) $s' \models \neg(R.\mathit{ctoggle} \rightarrow t{=}R.\mathit{toggle})$ (by 2 and 2.4 since G does not change $R.\mathit{ctoggle}$ and $R.\mathit{toggle}$)
   2.4.2) $s' \models R.\mathit{pc}{=}\mathsf{RTS} \wedge R.\mathit{ftoggle}{=}t$ (by $2, 2.2, 2.4$ and effect G)
   2.4.3) $s' \models \mathit{INVR}$ (by 2.4.1 and 2.4.2)
   2.5) $s' \models \mathit{INVR}$ (by 2.3 and 2.4)
3. Assume $a = \mathit{IND}(d,i)$
   3.1) $s' \models R.\mathit{pc}{=}\mathsf{SA} \wedge R.\mathit{ctoggle} \wedge R.\mathit{ftoggle}{\neq}R.\mathit{toggle}$ (by 3 and effect IND)
   3.2) $s' \models \mathit{INVR}$ (by 3.1)
4. Assume $a = A(f,l,t)$
   4.1) $s' \models R.\mathit{pc}{=}\mathsf{WF}$ (by 4 and effect A)
   4.2) $s' \models \mathit{INVR}$ (by 4.1)
5. Assume $a = \mathit{E2}$
   5.1) $s \models S.\mathit{pc}{=}\mathsf{ET2}$ (by 5 and precondition $\mathit{E2}$)
   5.2) $s \models R.\mathit{pc}{=}\mathsf{WF}$ (by 5.1 and $\mathit{INV1}$)
   5.3) Assume $s \models R.\mathit{timer2\_on} \wedge \neg R.\mathit{first}$
   5.3.1) $s' \models R.\mathit{pc}{=}\mathsf{NOK} \wedge \neg R.\mathit{ctoggle}$ (by $5, 5.3$ and effect $\mathit{E2}$)
   5.3.2) $s' \models \mathit{INVR}$ (by 5.3.1)
   5.4) Assume $s \not\models R.\mathit{timer2\_on} \wedge \neg R.\mathit{first}$
   5.4.1) $s' \models R.\mathit{pc}{=}\mathsf{WF}$ (by $5, 5.2, 5.4$ and effect $\mathit{E2}$)
   5.4.2) $s' \models \mathit{INVR}$ (by 5.4.1)
   5.5) $s' \models \mathit{INVR}$ (by 5.3 and 5.4)
6. Assume $a = \mathit{T2}$
   6.1) $s \models R.\mathit{pc}{=}\mathsf{WF}$ (by 6 and precondition $\mathit{T2}$)
   6.2) $s' \models R.\mathit{pc}{=}\mathsf{WF}$ (by $6, 6.1$ and effect $\mathit{T2}$)
   6.3) $s' \models \mathit{INVR}$ (by 6.2)
7. Assume $a = \mathit{IND\_NOT\_OK}$
   7.1) $s' \models R.\mathit{pc}{=}\mathsf{WF}$ (by 7 and effect $\mathit{IND\_NOT\_OK}$)
   7.2) $s' \models \mathit{INVR}$ (by 7.1)
8. $s' \models \mathit{INVR}$ (by 1-7)

The next invariant INVL implies that when an acknowledgement message arrives at the sender, the three bits of this acknowledgement are determined by the state of the sender, and hence provide no information. The only information conveyed by an acknowledgement is the fact of its arrival itself, the rest is redundant.

Lemma 4. *The following property INVL is an invariant of BRP.*

$$\begin{array}{lll} L.full & \rightarrow & L.first{=}R.ffirst{=}S.first \wedge L.last{=}R.flast{=}\mathsf{one}(S.list) \wedge \\ & & R.ctoggle \wedge L.toggle{=}\neg R.toggle{=}S.toggle \end{array}$$

The following invariant is not used in the proof of the refinement, but is interesting because it implies that, when a frame arrives at the receiver, the first field of this frame is determined by the state of the receiver and the other fields of the frame. Hence the first bit of the frame conveys no information and is redundant.

Lemma 5. *The following property INVK′ is an invariant of BRP.*

$$K.full \quad \rightarrow \quad K.first{=}\mathsf{if}\ (R.ctoggle \rightarrow K.toggle{=}R.toggle)\ \mathsf{then}\ R.first\ \mathsf{else}\ R.ffirst$$

Safety We have now prepared the ground for the first main results of this paper: the existence of a weak refinement from *BRP* to *P*. Since states of *BRP* and *P* are fully determined by the values of their state variables, we can define a weak refinement from *BRP* to *P* by expressing the values of the state variables of *P* in terms of those of *BRP*. The weak refinement function, which is given in Theorem 6, turns out to be surprisingly simple: *P.list* is either *S.list* or $\mathsf{tl}(S.list)$, *P.busy* is just *S.busy*, *P.first* is just *R.first*, and *P.error* holds iff the receiver's program counter equals NOK or will necessarily do so after the next locally controlled action.

Theorem 6. *The function REF defined by the following formula is a weak refinement from BRP to P.*

$$\begin{array}{lll} \wedge\ P.list & = & \mathsf{if}\ S.pc{\in}\{\mathsf{ET2},\mathsf{WT2}\} \vee (R.ctoggle \rightarrow S.toggle{=}R.toggle)\ \mathsf{then}\ S.list \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{else}\ \mathsf{tl}(S.list) \\ \wedge\ P.busy & = & S.busy \\ \wedge\ P.first & = & R.first \\ \wedge\ P.error & = & R.pc{=}\mathsf{NOK} \vee (S.pc{=}\mathsf{ET2} \wedge R.timer2\_on \wedge \neg R.first) \end{array}$$

Proof. 5 pages densely filled with ASCII.

Corollary 7. *BRP is safe with respect to P.*

Deadlock freeness

Theorem 8. *For each reachable and quiescent state s of BRP, REF(s) is a quiescent state of P.*

Corollary 9. *BRP is deadlock free with respect to P.*

Implementation We now come to the main result of this section, which says that the Bounded Retransmission Protocol correctly implementation specification P. Given that we have already shown that BRP is safe and deadlock free, the essential fact that remains to be established is that BRP is divergence free, i.e., will always eventually produce some allowed output after a given input. As is usual with liveness properties, we show this by presenting a weight function that maps states onto a well-founded domain (the natural numbers in our case) and demonstrating that after an input all actions, except possible further inputs and the required outputs, decrease the weight.

For each state s of BRP, define $weight(s)$ as the result of evaluating the following expression in s:

$$\begin{aligned}
&(6\mathsf{max}+5)\cdot \mathsf{length}(S.list) + 6\cdot(\mathsf{max}+1 \mathbin{\dot{-}} S.rn) + \iota(S.timer1\_on)\\
&+ 4\cdot\iota(S.pc{=}\mathsf{ET2}) + 2\cdot\iota(S.pc{=}\mathsf{WT2}) + \iota(S.pc\in\{\mathsf{SF},\mathsf{WA}\})\\
&+ 4\cdot\iota(K.full)\\
&+ 3\cdot\iota(R.pc{=}\mathsf{SI}) + 2\cdot\iota(R.pc\in\{\mathsf{SA},\mathsf{RTS}\}) + \iota(R.pc{=}\mathsf{NOK})\\
&+ \iota(L.full)
\end{aligned}$$

Besides some standard arithmetic operations, we have used here a function $\mathsf{length} : \mathbf{List} \to \mathbf{Nat}$, which gives the length of a list, and a function $\iota : \mathbf{Bool} \to \mathbf{Nat}$ defined by

$$\iota(\mathsf{false}) = 0 \qquad \iota(\mathsf{true}) = 1$$

Lemma 10. *Suppose s, s' are reachable states of BRP and $s \xrightarrow{a} s'$ for some action a, with $a \notin in(BRP)$ and a not of the form $CONF(c)$, for some c. Then $weight(s) > weight(s')$.*

Theorem 11. *BRP implements P.*

Proof. Assume that $\beta \in fairtraces(BRP)$. We must prove $\beta \in fairtraces(P)$.

Let $\alpha = s_0 a_1 s_1 a_2 s_2 \cdots$ be a fair execution of BRP with trace β.

If α is finite then α is quiescent and it follows by Corollary 9 that P has a quiescent execution with trace β. Since each quiescent execution is also fair, this implies $\beta \in fairtraces(P)$. So we may assume w.l.o.g. that α is infinite.

Using the fact that REF is a weak refinement (Theorem 6) we can easily construct an execution α' of P with trace β. It remains to prove that α' is fair. For this we distinguish between two cases.

1. β contains infinitely many $CONF$ actions. Since $part(P)$ contains only one class, and execution α' contains infinitely many occurrences of actions from that class, α' is fair.
2. β contains only finitely many $CONF$ actions. Call an input action a_i in α *discarded* if $s_{i-1} = s_i$. Then between any pair of non-discarded inputs in α there must be a $CONF$ action, because a non-discarded input always changes $S.busy$ from false to true, and $CONF$ is the only action that can set $S.busy$ to false again. Thus there is a point N in α after which

there are no more *CONF* actions and moreover all inputs are discarded. By Lemma 10 it follows that, for all $i > N$, if a_i is a locally controlled action then $weight(s_{i-1}) > weight(s_i)$ else $weight(s_{i-1}) = weight(s_i)$. Thus there must be a point $M \geq N$ after which α only contains discarded inputs. Moreover, all states from that point on are equal and quiescent, otherwise α would not be fair. By Lemma 8, $S.busy$=false for all quiescent states of *BRP*. It follows that α has an infinite suffix $s_M\ REQ(\epsilon)\ s_M\ REQ(\epsilon)\ s_M \cdots$. But this means that the corresponding execution α' of P has an infinite suffix $s_q\ REQ(\epsilon)\ s_q\ REQ(\epsilon)\ s_q \cdots$. Moreover, by Lemma 8 state s_q is quiescent. This implies that α' is fair.

4 Proof-Checking

In this Section, we report on the proof-checking of the protocol verification of Section 3. We have checked the proofs of all the invariants (Lemmas 2, 3, 4 and 5, as well as some other invariants that we needed as lemmas but that are not discussed here), the proof that *REF* is a weak refinement (Theorem 6), and the proof that *REF* preserves quiescence (Theorem 8). We did not proof-check "meta-results" such as Lemma 1 and Corollaries 7 and 9. Also, we have not checked the "liveness" result of Theorem 11. Proof-checking these results as well would have required a considerable effort with, at least in the case of the Bounded Retransmission Protocol, only a small payoff. Still we think that the formalization and mechanical checking of these type of results will be an important topic of future research.

4.1 Coq Proof Development System

Coq is a proof assistant for higher-order logic. It is based on the *Calculus of Inductive Constructions* [24], which is a polymorphic type theory allowing dependent types and inductive types. Constructing a proof in Coq is an interactive process. The user specifies the proof strategy (e.g. which deduction rule should be applied) and Coq does all the calculations.

Notation Coq is based on type theory, which means that (apart from some built in 'pretty printing' rules) all applications are denoted in pre-fix. In this paper we adapt the ASCII input and output of Coq in order to improve the readability. We write

$\lambda x{:}A\,.\,b$	for	`[x:A]b`
$\forall x{:}A\,.\,B$	for	`(x:A)B` when `x` is a free variable in `B`
$A_1 \to A_2$	for	`A1->A2`
$x = y$	for	`eq A x y` (or `<A>x=y`)
$A \wedge B$	for	`and A B` (or `A/\B`)
$A \vee B$	for	`or A B` (or `A\/B`)
$A \times B$	for	`prod A B` (or `A*B`)
$\sim A$	for	`A->False` (or `~A`)
$x \neq y$	for	$\sim(x = y)$

Note that we omit the type information in $x = y$. The reader can easily deduce this type from the context.

The Tactics theorem prover The Coq system makes use of the *Curry-Howard isomorphism*, which states that λ-terms can be used to encode natural deduction proofs. For instance the well-known S-combinator

$$\lambda x{:}A \to (B \to C) \,.\, \lambda y{:}A \to B \,.\, \lambda z{:}A \,.\, x\ z\ (y\ z)$$

encodes under the Curry-Howard isomorphism the following natural deduction proof. (Cancelled hypotheses are placed between square brackets.)

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{[\,A \to (B \to C)\,]^3 \quad [\,A\,]^1}{B \to C} \qquad \cfrac{[\,A \to B\,]^2 \quad [\,A\,]^1}{B}}{C}}{A \to C}\,1}{(A \to B) \to (A \to C)}\,2}{(A \to (B \to C)) \to ((A \to B) \to (A \to C))}\,3$$

In order to give the reader a flavor of a proof session in Coq we give the list of commands (*tactics*) needed to construct the proofterm above. At the right we expose how the proofterm is built step by step. Note that proofterm is constructed 'top-down'. The terms $Hyp_1, \ldots, Hyp_5$ represent the subgoals that are generated during the proof session. (We omit the types in the proofterm in order to save space.)

Goal $(A \to (B \to C)) \to ((A \to B) \to (A \to C))$.	proofterm: Hyp_1
Intros $x\ y\ z$.	proofterm: $\lambda xyz.\, Hyp_2$
Apply x.	proofterm: $\lambda xyz.\, x\ Hyp_3\ Hyp_4$
Assumption.	proofterm: $\lambda xyz.\, x\ z\ Hyp_4$
Apply y.	proofterm: $\lambda xyz.\, x\ z\ (y\ Hyp_5)$
Assumption.	proofterm: $\lambda xyz.\, x\ z\ (y\ z)$

Tactics can be composed to so called *tacticals*. The tactical tac_0 ; tac_1 first applies tac_0 on the current goal and then applies tac_1 on all the subgoals generated by tac_0. More generally, the tactical tac_0 ; $[tac_1|\cdots|tac_N]$ first applies tac_0 and then applies tac_i on the i-th subgoal generated by tac_0 $(i = 1, \ldots, N)$. (When tac_0 does not generate N subgoals, this tactical fails.) The following tactical generates the same S-combinator.

$$\textit{Intros}\ x\ y\ z\ ;\ \textit{Apply}\ x\ ;\ [\textit{Assumption} \mid \textit{Apply}\ y\ ;\ \textit{Assumption}].$$

Details about the use of Coq can be found in the Coq manual [9].

One of the most important features of Coq is the so called *program abstraction*. From a proof of $\forall x{:}A.\ \exists y{:}B.\, P(x, y)$ one can extract a function (program) $f : A \longrightarrow B$ such that $\forall x : A.\, P(x, f(x))$. We do not need this facility for our purposes.

Inductive types In our encodings we extensively use inductive types. For details about this notion we refer to [24]. In this paper we restrict ourselves to some examples. When we define $nat : Set$, $O : nat$ and $S : nat \to nat$ then $\underbrace{S(\cdots(S}_{n}\ O))$ is of type *nat* for all $n \geq 0$ but there might still be other terms of type *nat.* In Coq we have the alternative possibility

$$\begin{array}{lll} nat & := & \mathsf{Ind}(X{:}Set)\{X \mid X \to X\} \\ O & := & \mathsf{Constr}(1, nat) \\ S & := & \mathsf{Constr}(2, nat). \end{array}$$

which is the result of the Coq command:

Inductive Definition nat : Set = O : nat | S : nat → nat.

This should be read as '*nat* is the smallest set X closed under two constructors, one of type X and one of type $X \to X$'. When we choose for the second option then *nat* contains no other terms then those constructed from O and S. In other words: for arbitrary $P : nat \to *$ and $x : nat$ we are able to construct a term of type $P\,x$ from terms $\wp_o : P\,O$ and $\wp_s : \forall y : nat\,.\, P\,y \to P\,(S\,y)$. This term is written as $<\!P\!>\mathsf{Match}\ x\ \mathsf{with}\ \wp_o\ \wp_s$ in the system. The reduction behavior of this term is determined by the construction of x from O and S.

$$\begin{array}{l} <\!P\!>\mathsf{Match}\ O\ \mathsf{with}\ \wp_o\ \wp_s\ \ \twoheadrightarrow \wp_o \\ <\!P\!>\mathsf{Match}\ S\,y\ \mathsf{with}\ \wp_o\ \wp_s \twoheadrightarrow \wp_s\ y\ (<\!P\!>\mathsf{Match}\ y\ \mathsf{with}\ \wp_o\ \wp_s) \end{array}$$

Note that this reductions are well typed, i.e. reduction of a term does not change its type. When $* \equiv Prop$ (which is a predefined notion of Coq, representing the type of all propositions) then P is a predicate over *nat* and $\wp_o$ and $\wp_s$ are just the usual proofs for the zero-case and the successor-case. When $* \equiv Set$ (another predefined notion, representing the type of all sets) and P is a constant function on *nat*, say $P \equiv \lambda n : nat\,.\,A$ for some $A : Set$, then $\wp_o : A$ and $\wp_s : nat \to A \to A$ and $\lambda x : nat\,.\ <\!P\!>\mathsf{Match}\ x\ \mathsf{with}\ \wp_o\ \wp_s$ represents the function from *nat* to A that is defined by primitive recursion from $\wp_o$ and $\wp_s$. In other words: $\lambda a : A\,.\,\lambda g : nat \to A \to A\,.\,\lambda x : nat\,.\ <\!P\!>\mathsf{Match}\ x\ \mathsf{with}\ a\ g$ is a recursor. With this mechanism one can define any primitive recursive function (and even more because one can use higher order recursion).

We conclude this subsection with the illustration of how one can use inductive types to encode logical conjunction and disjunction. Define

$$\begin{array}{lll} and & := & \lambda A, B : Prop.\mathsf{Ind}(X{:}Prop)\{A \to B \to X\} \\ or & := & \lambda A, B : Prop.\mathsf{Ind}(X{:}Prop)\{A \to X \mid B \to X\} \\ conj & := & \lambda A, B : Prop.\mathsf{Constr}(1, (and\ A\ B)) \\ or\_introl & := & \lambda A, B : Prop.\mathsf{Constr}(1, (or\ A\ B)) \\ or\_intror & := & \lambda A, B : Prop.\mathsf{Constr}(2, (or\ A\ B)) \end{array}$$

then $(and\ A\ B)$ contains no other terms (proofs) then those constructed from $(conj\ A\ B)$ and $(or\ A\ B)$ contains no other terms then those constructed from $(or\_introl\ A\ B)$ or $(or\_intror\ A\ B)$. This exactly reflects the intuitionistic meanings of conjunction and disjunction. The only way to prove $A \wedge B$ is proving both A and B, and the only way to prove $A \vee B$ is proving A or B.

4.2 Protocol Specification

The hand-written proof is written in many sorted predicate logic. For each sort there is an equality relation. We use the built-in encoding of polymorphic Leibniz equality

$$eq := \lambda A\!:\!Set\,.\,\lambda a\!:\!A\,.\,\mathsf{Ind}(X\!:\!A \to Prop)\{X\ a\}$$

to represent these equalities. Furthermore we use the standard encodings for conjunction and disjunction, briefly explained in the previous subsection. The types *Prop* and *Set*, also mentioned in the previous subsection, are predefined notions (constants) of Coq, comparable with the star ($*$) in systems of the Barendregt-cube [2]. The logical implication and the functional implication are both identified with the arrow of type theory. (As a consequence our proof is intuitionistically valid.)

There are at least two ways to encode the functional behavior of a function $F : A \longrightarrow B$. For instance the sum $+ : \mathbf{Nat} \longrightarrow \mathbf{Nat} \longrightarrow \mathbf{Nat}$ can be defined by

$$\begin{array}{lll} sum & : nat \to nat \to nat & \\ s_1 & : \forall x\!:\!nat\,. & sum\ O\ x \quad = x \\ s_2 & : \forall x, y\!:\!nat\,. & sum\ (S\,y)\ x = S\,(sum\ y\ x) \end{array}$$

In this case *sum* is just a variable without any computational power. Computing the value of ($sum\ n\ m$) can be done by the Coq command '*Rewrite* s_1.' or '*Rewrite* s_2.' depending on the value of n. The alternative is to define *sum* as an abbreviation.

$$sum := \lambda x, y\!:\!nat\,.\ <\!\lambda z\!:\!nat\,.\,nat\!>\!\mathsf{Match}\ x\ \mathsf{with}\ y\ \lambda z\!:\!nat\,.\,S \tag{1}$$

The advantage of the second approach is that one does not have to give any command for computing the value of ($sum\ n\ m$). Computation in this case is just normalization, and done automatically by the system. Note that Coq can not reduce ($sum\ n\ O$) to n when n is a variable. In such a case one can do a case analysis on n. We try to use the second approach as much as possible.

In Coq it is allowed to omit the λ-abstraction in $<\!P\!>\mathsf{Match}\ x\ \mathsf{with}\ldots$ when $P\,x$ does not actually depend on x. So $<nat>\mathsf{Match}\ x\ \mathsf{with}\ldots$ can replace $<\lambda z : nat\,.\,nat>\mathsf{Match}\ x\ \mathsf{with}\ldots$ in (1). In the sequel we will omit such λ-abstractions.

The main result in the hand-written proof is that there exists a *weak refinement* from automaton *BRP* to automaton P. We modified our encodings several times in order to get a better formulation in Coq of this weak refinement property. In the approach that we have chosen eventually, we can represent the function $REF : states(BRP) \longrightarrow states(P)$ by a λ-term like in (1).

Data types The specification of the Bounded Retransmission Protocol makes use of several data types. We represent these types by inhabitants of *Set*.

– the sort **Bool** is represented by the inductive type

$$\begin{aligned} \mathit{bool} &:= \mathsf{Ind}(X\!:\!\mathit{Set})\{X \mid X\} \\ \mathit{true} &:= \mathsf{Constr}(1, \mathit{bool}) \\ \mathit{false} &:= \mathsf{Constr}(2, \mathit{bool}) \end{aligned}$$

All functions on booleans can be represented by λ-terms. We will write

$$\begin{array}{lll l} \neg\, x & \text{for} & <\!\mathit{bool}\!>\mathsf{Match}\ x\ \mathsf{with}\ \mathit{false}\ \mathit{true} & \text{(negation)} \\ x \equiv y & \text{for} & <\!\mathit{bool}\!>\mathsf{Match}\ x\ \mathsf{with}\ y\ (\neg\, y) & \text{(equality)} \\ x \not\equiv y & \text{for} & \neg\,(x \equiv y) & \text{(inequality)} \\ x \sqcap y & \text{for} & <\!\mathit{bool}\!>\mathsf{Match}\ x\ \mathsf{with}\ y\ \mathit{false} & \text{(conjunction)} \\ x \sqcup y & \text{for} & <\!\mathit{bool}\!>\mathsf{Match}\ x\ \mathsf{with}\ \mathit{true}\ y & \text{(disjunction)} \end{array}$$

– The sort **Data** is represented by the variable $\mathit{data} : \mathit{Set}$. Furthermore we defined a variable $\mathit{Undefined} : \mathit{data}$ which represents the element $\perp \in$ **Data**.
– The sort **List** is defined as the inductive type with constructors NIL and ADD, representing ϵ and add respectively. In formula:

$$\begin{aligned} \mathit{LIST} &:= \mathsf{Ind}(X\!:\!\mathit{Set})\{X \mid \mathit{data} \to X \to X\} \\ \mathit{NIL} &:= \mathsf{Constr}(1, \mathit{LIST}) \\ \mathit{ADD} &:= C\mathsf{onstr}(2, \mathit{LIST}) \end{aligned}$$

Functions like hd, tl, ϵ and one can all be represented by λ-terms. For instance

```
one := λL:LIST . <bool>Match L with
           false
           λd:data . λy:LIST . λb:bool . <bool>Match y with
              true
              λd:data . λy:LIST . λb:bool . false
```

The equalities from Subsection 3.2 are satisfied. All the right-hand-sides are just the normal forms of the left-hand-sides.
– The finite sets **Spc**, **Rpc**, **Conf** and **Ind** are encoded as inductive types in the same style as the booleans. Some versions of Coq can not distinguish inductive types that have the same structure. Choosing different names for such types just introduces different names for the same expression. In particular, finite sets that have the same cardinality are not distinguished. For instance *Spc* and *Rpc* are two abbreviations for the same type $\mathsf{Ind}(X : \mathit{Set})\{X \mid X \mid X \mid X \mid X\}$. We can prove $\mathit{SF} = \mathit{WF}$ by reflexivity because *SF* and *WF* are both the first element of a set of five elements. A typing error $\pi_{\mathit{Rpc}}(\pi_R(x)) = \mathit{WA}$ in one of the invariants was overlooked for a long time.

Similar to the binary operators $\equiv$ and $\not\equiv$ on *bool*, we have defined λ-terms representing $\equiv$ and $\not\equiv$ on *Spc, Rpc, Conf* and *Ind*.

Note: $\mathit{SF} = \mathit{WF} : \mathit{Prop}$ and $\mathit{SF} \equiv \mathit{WF} : \mathit{bool}$.

The actions We define finite sets *act_BRP* and *act_P* representing the sets of actions. We can not use the same name for actions of different automata. Hence we add a prime ($'$) by those action of automaton P that already occurred in automaton *BRP*. Constructors of *act_BRP* are $REQ : LIST \to act\_BRP$, $F : bool \to bool \to bool \to data \to act\_BRP$, etc. Elements of *act_P* are $REQ' : LIST \to act\_P$, etc. We add an extra element τ to the inductive set *act_P*. Next we define a term *ev* (evaluate) which maps actions of *BRP* to the corresponding actions of P. ($REQ \overset{ev}{\longmapsto} REQ'$, etc.) Internal actions of BRP are mapped to τ.

$$
\begin{aligned}
ev := \lambda a\!:\!act\_BRP\,.\ {<}act\_P{>}\mathsf{Match}\ a\ \mathsf{with}\ & REQ' \\
& \lambda B_1, B_2, B_3\!:\!bool\,.\,\lambda d\!:\!data\,.\,\tau \\
& \tau \\
& \lambda B_1, B_2, B_3\!:\!bool\,.\,\tau \\
& \tau \\
& \tau \\
& \tau \\
& CONF' \\
& \tau \\
& \tau \\
& \lambda B_1, B_2, B_3\!:\!bool\,.\,\lambda d\!:\!data\,.\,\tau \\
& \lambda B_1, B_2, B_3\!:\!bool\,.\,\tau \\
& IND' \\
& INDn'.
\end{aligned}
$$

Note that we simply postulate which actions are internal in automaton *BRP*. One could think of encoding whole the theory about input- and output actions. Given the status of the actions in the components, the status of the actions in the product automaton could then be computed. However, this part of the hand-written proof is not the kind of reasoning where automatic verification pays off.

The state spaces The following step is the definition of types *states_BRP* and *states_P* representing the state spaces of the two automata *BRP* and P. The state space of *BRP* is encoded as a cartesian product of cartesian products. $states\_S := Spc \times bool \times \cdots \times bool \times LIST \times nat$. Analogously we define *states_K, states_L* and *states_R*. Now $states\_BRP := states\_S \times states\_K \times states\_L \times states\_R$. Finally $states\_P := LIST \times bool \times bool \times bool$.

We use the standard inductive type *prod* [9] with constructor *pair* for the encoding of the cartesian product. When A and B are sets and $(a, b) \in A \times B$ then (a, b) is represented in Coq by $(pair\ A\ B\ a\ b)$. Hence $(a, b, c) \in A \times B \times C$ is represented by $(pair\ A\ (B \times C)\ a\ (pair\ B\ C\ b\ c))$ which is already a bit less friendly. An element of *states_BRP* would cover the whole page. However, we can represent a function $F : A \longrightarrow B \longrightarrow C \longrightarrow (A \times B \times C)$ such that $F(a)(b)$ maps c to (a, b, c) by a λ-term in the style of (1).

$$F := \lambda x\!:\!A\,.\,\lambda y\!:\!B\,.\,\lambda z\!:\!C\,.\,pair\ A\ (B \times C)\ x\ (pair\ B\ C\ y\ z)$$

Then (a, b, c) can be represented by $(F\ a\ b\ c)$. This way we define functions *st_S, st_K, st_L, st_R, st_BRP* and *st_P* mapping the components of the state spaces to the corresponding elements in the cartesian products. Furthermore we define projection functions $\pi_{toggle} : states\_S \to bool$, $\pi_R : states\_BRP \to states\_R$, etc. For instance

$$\begin{array}{ll} \pi_{toggle}\ (st\_S\ B_1\ B_2\ B_3\ B_4\ B_5\ L\ n) & \twoheadrightarrow B_3 \\ \pi_R\ (st\_BRP\ S\ K\ L\ R) & \twoheadrightarrow R \end{array}$$

The weak refinement The mapping *REF* can now be represented by the λ-term below.

$$\begin{array}{l} ref := \lambda x : states\_BRP\,.\,(st\_P \\ \quad <LIST>\mathsf{Match}\ \begin{array}[t]{l} \pi_{Spc}\ (\pi_S\ x) \equiv ET2\ \sqcup \\ \pi_{Spc}\ (\pi_S\ x) \equiv WT2\ \sqcup \\ \neg\, \pi_{ctoggle}\ (\pi_R\ x)\quad \sqcup \\ \pi_{toggle}\ (\pi_S\ x) \equiv \pi_{toggle}\ (\pi_R\ x) \end{array} \\ \\ \qquad\qquad \mathsf{with}\ \ \pi_{list}\ (\pi_S\ x)\quad (tl\ (\pi_{list}\ (\pi_S\ x))) \\ \\ \quad \pi_{busy}\ (\pi_S\ x) \\ \\ \quad \pi_{first}\ (\pi_R\ x) \\ \\ \quad \pi_{Rpc}\ (\pi_R\ x) \equiv NOK \\ \quad \sqcup \\ \quad \pi_{Spc}\ (\pi_S\ x) \equiv ET2\quad \sqcap \\ \quad \pi_{timer2\_on}\ (\pi_R\ x)\quad \sqcap \\ \quad \neg\ (\pi_{first}\ (\pi_R\ x))). \end{array}$$

The step relation The next step is the definition of types $step : act\_BRP \to states\_BRP \to states\_BRP \to Prop$ and $step' : act\_P \to states\_P \to states\_P \to Prop$ representing the notion of 'step'. The intended meaning of $step\ a\ s_1\ s_2$ is $s_1 \xrightarrow{a} s_2$. We use an inductive type again.

$$\begin{array}{l} step := \mathsf{Ind}(X : act\_BRP \to states\_BRP \to states\_BRP \to Prop)\{ \\ \qquad \forall \sigma : LIST\,.\,\forall s : Spc\,.\,\forall B_1, B_2, B_3, B_4 : bool\,. \\ \qquad \forall L : LIST\,.\,\forall n : nat\,. \\ \qquad \forall s_K : states\_K\,.\,\forall s_L : states\_L\,.\,\forall s_R : states\_R\,. \\ \qquad false = (empty\ \sigma) \to \\ \qquad (X(REQ\ \sigma) \\ \qquad\ (st\_BRP\ (st\_S\ s\ false\ B_1\ B_2\ B_3\ B_4\ L\ n)\ s_K\ s_L\ s_R) \\ \qquad\ (st\_BRP\ (st\_S\ s\ true\ B_1\ B_2\ B_3\ B_4\ \sigma\ n)\ s_K\ s_L\ s_R)) \\ \quad | \\ \quad \vdots \\ \quad |\quad \cdots\} \end{array}$$

This enables us to do a case analysis on H when we have a proof $H : (\mathit{step}\ a\ s_1\ s_2)$ in our context and we want to prove $\phi(s_1, s_2)$ for some $s_1, s_2 : \mathit{states\_BRP}$. The first constructor of *step* leads to the following subgoal:

$$\begin{array}{l}
\forall \sigma : \mathit{LIST} \,.\, \forall s : \mathit{Spc} \,.\, \forall B_1, B_2, B_3, B_4 : \mathit{bool} \,.\\
\forall L : \mathit{LIST} \,.\, \forall n : \mathit{nat} \,.\\
\forall s_K : \mathit{states\_K} \,.\, \forall s_L : \mathit{states\_L} \,.\, \forall s_R : \mathit{states\_R} \,.\\
(\mathit{st\_S}\ s\ \mathit{false}\ B_1\ B_2\ B_3\ B_4\ L\ n)\ s_K\ s_L\ s_R),\\
\quad (\mathit{st\_BRP}\ (\mathit{st\_S}\ s\ \mathit{true}\ B_1\ B_2\ B_3\ B_4\ \sigma\ n)\ s_K\ s_L\ s_R))
\end{array}$$

This is the result of matching s_1 and s_2 with the terms of type *states_BRP* on which X is applied in the first–constructor–case of *step* (describing the behavior of the request action).

In our approach we encode directly how the actions affect the product automaton *BRP*. This way we avoid the problem of encoding how the composition of the automaton *BRP* out of its components S, K, L and R is organized. The fact that local actions that have the same name are synchronized in the product automaton is difficult to express.

Some actions are split in more than one case. For instance the action B is split into *B_1* with extra precondition one(*list*)=true and *B_2* with extra precondition one(*list*)=false. This way we obtain 24 constructors for *step*.

Reachability Reachability is encoded as an inductive type, having two constructors. The first constructor encodes the reachability of the initial state. The second constructor encodes the preservation of *reach* under *step*.

$$\begin{array}{rl}
\mathit{reach} := \mathsf{Ind}(X : \mathit{states\_BRP} \to \mathit{Prop})\{ & \\
& \forall s : \mathit{states\_BRP} \,.\, (\mathit{start}\ s) \to (X\ s)\\
| & \forall a : \mathit{act\_BRP} \,.\, \forall s_1, s_2 : \mathit{states\_BRP} \,.\\
& \quad (\mathit{step}\ a\ s_1\ s_2) \to (X\ s_1) \to (X\ s_2)\}
\end{array}$$

where *start* is the predicate on *states_BRP* that holds only for the initial state, also defined inductively:

$$\begin{array}{l}
\mathit{start} := \mathsf{Ind}(X : \mathit{states\_BRP} \to \mathit{Prop})\{\\
\qquad \forall B_1, B_2, B_3, B_4, B_5, B_6, B_7, B_8, B_9, B_{10}, B_{11} : \mathit{bool} \,.\\
\qquad \forall L : \mathit{LIST} \,.\, \forall d_1, d_2 : \mathit{data} \,.\\
\qquad (X\ (\mathit{st\_BRP}\\
\qquad (\mathit{st\_S}\ \mathit{SF}\ \mathit{false}\ \mathit{true}\ B_1\ \mathit{false}\ \mathit{false}\ L\ O)\\
\qquad (\mathit{st\_K}\ B_2\ B_3\ B_4\ \mathit{false}\ d_1)\\
\qquad (\mathit{st\_L}\ B_5\ B_6\ B_7\ \mathit{false})\\
\qquad (\mathit{st\_R}\ \mathit{WF}\ B_8\ B_9\ B_{10}\ d_2\ \mathit{true}\ B_{11}\ \mathit{false}\ \mathit{false}\ \mathit{false})))\}.
\end{array}$$

Assume that we want to prove $\phi(s_0)$ for some s_0:*states_BRP* and that we have a proof R:$(\mathit{reach}\ s_0)$. Eliminating the inductive type *reach* returns two subgoals:

$$\begin{array}{ll}
(i) & : \forall s : \mathit{states\_BRP} \,.\, (\mathit{start}\ s) \to \phi(s)\\
(ii) & : \forall a : \mathit{act\_BRP} \,.\, \forall s_1, s_2 : \mathit{states\_BRP} \,.\\
& \quad (\mathit{step}\ a\ s_1\ s_2) \to (\mathit{reach}\ s_1) \to \phi(s_1) \to \phi(s_2)
\end{array}$$

The first goal can be proved via '*Intros s H ; Elim H*' and the second goal can be proved via '*Intros a* s_1 s_2 *H* ; *Elim H*' which leads to proving $(\mathit{reach}\ s_1) \rightarrow \phi(s_1) \rightarrow \phi(s_2)$ by a case analysis on the proofterm H of type $(\mathit{step}\ a\ s_1\ s_2)$.

The weak refinement property Assume that we have in *BRP* a transition $s_1 \xrightarrow{a} s_2$ for some external *BRP*-action a, then we must have a transition $\mathit{REF}(s_1) \xrightarrow{a} \mathit{REF}(s_2)$ in automaton P. We are able to express this as

$$(\mathit{step}\ a\ s_1\ s_2) \rightarrow (\mathit{step}'\ (\mathit{ev}\ a)\ (\mathit{ref}\ s_1)\ (\mathit{ref}\ s_2)) \tag{2}$$

When a is an internal action then (2) evaluates to

$$(\mathit{step}\ a\ s_1\ s_2) \rightarrow (\mathit{step}'\ \tau\ (\mathit{ref}\ s_1)\ (\mathit{ref}\ s_2))$$

which we can not prove for there are no constructors of the form $\mathit{step}'\ \tau \ldots$ in the definition of step'. When we add a constructor of type $\forall s{:}\mathit{states\_P}\,.\,(\mathit{step}'\ \tau\ s\ s)$ then we can prove $(\mathit{step}\ a\ s_1\ s_2) \rightarrow (\mathit{step}'\ \tau(\mathit{ref}\ s_1)\ (\mathit{ref}\ s_2))$ iff we can prove $(\mathit{ref}\ s_1) = (\mathit{ref}\ s_2)$. This is exactly what we required so (2) also encodes the weak refinement property when a is internal.

Of course (2) does not have to hold for states that can not be reached. Hence we can add an extra precondition. Furthermore we quantify over the states and the action:

$$\begin{aligned}&\forall a{:}\mathit{act\_BRP}\,.\,\forall s_1, s_2{:}\mathit{states\_BRP}\,.\,(\mathit{reach}\ s_1) \rightarrow\\ &(\mathit{step}\ a\ s_1\ s_2) \rightarrow (\mathit{step}'\ (\mathit{ev}\ a)\ (\mathit{ref}\ s_1)\ (\mathit{ref}\ s_2))\end{aligned} \tag{3}$$

A weak refinement mapping also has to map initial states to initial states. This is encoded as

$$\forall s{:}\mathit{states\_BRP}\,.\,(\mathit{start}\ s) \rightarrow (\mathit{start}'\ (\mathit{ref}\ s)) \tag{4}$$

The invariants For proving (3) we have to use the invariants. These invariants are proven valid in the reachable states only. Their formulation is rather straightforward. Below we give the encoded version of invariant *INVR* of Lemma 3. Note that the expression is prefixed by the precondition $(\mathit{reach}\ x)$.

$$\begin{aligned}\mathit{invr} := {}& \forall x{:}\mathit{states\_BRP}\,.\,(\mathit{reach}\ x) \rightarrow\\[1em]
&\quad \pi_{Rpc}(\pi_R\ x) = \mathit{NOK} \rightarrow\\
&\quad \pi_{ctoggle}(\pi_R\ x) = \mathit{false}\\
&\quad \wedge\\
&\quad \pi_{Rpc}(\pi_R\ x) = \mathit{SI} \rightarrow\\
&\quad \pi_{ctoggle}(\pi_R\ x) = \mathit{true} \rightarrow\\
&\quad \pi_{ftoggle}(\pi_R\ x) = \pi_{toggle}(\pi_R\ x)\\
&\quad \wedge\\
&\quad (\pi_{Rpc}(\pi_R\ x) = \mathit{RTS} \vee \pi_{Rpc}(\pi_R\ x) = \mathit{SA}) \rightarrow\\
&\quad (\pi_{ctoggle}(\pi_R\ x) = \mathit{true} \wedge \pi_{ftoggle}(\pi_R\ x) = \neg\,(\pi_{toggle}(\pi_R\ x)))\end{aligned}$$

4.3 Correctness Proof

Goals Proofs of the invariants and the refinement are essentially by induction over the transitions and split in the corresponding 25 cases (one initial state and 24 transition steps have to be considered). As is to be expected, transitions that do not affect variables that occur in an invariant prove in Coq simply by assumption with the induction hypothesis. Other cases resolve into further subgoals.

In this application of I/O automata, most predicates are equality assertions over state variables and the proofs involve much propositional reasoning. This is best illustrated by means of an example subgoal: Figure 2 shows a Coq goal that occurs when proving invariant *INVR* (Lemma 3). After elimination of reachable states (Section 4.2), Coq has filled in the variables and terms in proper places in the states before and after the transition, in the precondition, and in the invariant. The assertion to prove is on top, below that are the assumptions. This case corresponds to action G in case ($ctoggle \rightarrow t{=}toggle$). The latter condition is expressed by assumption H. Other preconditions of this transition arise as equalities over state variables that have been filled in automatically in the states before and after this transition. H_0 and H_1 assume reachability of these states. H_2 contains the induction hypothesis for the invariant property. The goal to prove is that the property holds for states after a G step.

The goal in Figure 2 decomposes in a number of subgoals. Figure 3 focuses on a particular subgoal. The proposition occurs in the rightmost conjunction of the invariant, viz. $R.pc{\in}\{\mathsf{RTS},\mathsf{SA}\} \rightarrow R.ftoggle{\neq}R.toggle$. H_5 assumes the precondition of this implication. The proof uses assumption H. The induction hypothesis (H_2 in Figure 2) has been decomposed into its constituent conjuncts. Applications of projection functions in the goal and in the assumptions have been reduced to retrieve the appropriate terms.

Many of the goals and subgoals that occur while proving the invariants in this exercise consist of a logical combination of equality statements. The same observation holds for those assumptions in the context that have not yet been eliminated and can be of relevance to the unfinished proof. In nearly all these cases the equality statements are over elements from finite sets. This holds for preconditions of transition steps as well as for predicates in the invariants.

The induction mechanism is often used in this exercise. Induction serves two purposes in the definition of a set: it states that the given elements are the only inhabitants of the set (*no junk* property) and it states that all elements are different (*no confusion* property). Inductively defined finite sets play an important role in the Coq checking of this verification, both to do analysis by cases as well as to distinguish between elements. Analysis by cases is provided directly in Coq via elimination of a variable over the elements of inductive set. Inequality of different elements of an inductively defined finite set is not directly available in Coq but must be derived with the *Match* mechanism. Because the verification described here uses this type of reasoning extensively, it will be

$(RTS = NOK \rightarrow$
$\pi_{ctoggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8) = false)$
$\wedge$
$(RTS = SI \rightarrow$
$\pi_{ctoggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8) = true \rightarrow$
$\pi_{ftoggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8) =$
$\pi_{toggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8))$
$\wedge$
$((RTS = RTS \vee RTS = SA) \rightarrow$
$\pi_{ctoggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8) = true$
$\wedge$
$\pi_{ftoggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8) =$
$\neg\,\pi_{toggle}(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8))$

============================

H_2 : $(WF = NOK \rightarrow$
$\pi_{ctoggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8) = false)$
$\wedge$
$(WF = SI \rightarrow$
$\pi_{ctoggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8) = true \rightarrow$
$\pi_{ftoggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8) =$
$\pi_{toggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8))$
$\wedge$
$((WF = RTS \vee WF = SA) \rightarrow$
$\pi_{ctoggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8) = true$
$\wedge$
$\pi_{ftoggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8) =$
$\neg\,\pi_{toggle}(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8))$
H_1 : $reach\ \ (st\_BRP\ s_S\ (st\_K\ f\ l\ t\ false\ d)\ s_L$
$(st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8))$
H_0 : $reach\ \ (st\_BRP\ s_S\ (st\_K\ f\ l\ t\ true\ d)\ s_L$
$(st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8))$
H : $\sim(B_6 = true \rightarrow B_5 = t)$
s_L : $states\_L$
s_S : $states\_S$
$d\ d_1$: $data$
$B_1\ B_2\ B_3\ B_4\ B_5\ B_6\ B_7\ B_8$: $bool$
$f\ l\ t$: $bool$
S : $step\ a\ s_1\ s_2$
$s_1\ s_2$: $states\_BRP$
a : act_BRP
R : $reach\ x$
x : $states\_BRP$

Fig. 2. Characteristic Coq subgoal for this application. The assertion to prove is on top, the assumptions are below. The goal forms part of the obligation to prove that transition G preserves invariant $INVR$ (Lemma 3). H_2 assumes the invariant property holds for states that enable this transition step. The assertion to prove is that the property holds for states after the transition.

$t = \neg\, B_5$

$$\text{==============================}$$

$$\begin{array}{ll}
H_5 & : RTS = RTS \vee RTS = SA \\
H_4 & : (WF = RTS \vee WF = SA) \rightarrow (B_6 = true \wedge B_3 = \neg\, B_5) \\
H_3 & : WF = SI \rightarrow B_6 = true \rightarrow B_3 = B_5 \\
H_2 & : WF = NOK \rightarrow B_6 = false \\
H_1 & : reach\ \ (st\_BRP\ s_S\ (st\_K\ f\ l\ t\ false\ d)\ s_L \\
 & \qquad\qquad\qquad\quad (st\_R\ RTS\ f\ l\ t\ d\ B_4\ B_5\ B_6\ B_7\ B_8)) \\
H_0 & : reach\ \ (st\_BRP\ s_S\ (st\_K\ f\ l\ t\ true\ d)\ s_L \\
 & \qquad\qquad\qquad\quad (st\_R\ WF\ B_1\ B_2\ B_3\ d_1\ B_4\ B_5\ B_6\ B_7\ B_8)) \\
H & : \sim(B_6 = true \rightarrow B_5 = t) \\
s_L & : states\_L \\
s_S & : states\_S \\
d\ d_1 & : data \\
\multicolumn{2}{l}{B_1\ B_2\ B_3\ B_4\ B_5\ B_6\ B_7\ B_8 : bool} \\
f\ l\ t & : bool \\
S & : step\ a\ s_1\ s_2 \\
s_1\ s_2 & : states\_BRP \\
a & : act\_BRP \\
R & : reach\ x \\
x & : states\_BRP
\end{array}$$

Fig. 3. A subgoal of the goal in Figure 2. The proof uses assumption H.

illustrated by means of a small example. We inductively define a finite set S and a predicate that discriminates its elements:

$$\textit{Inductive Definition } S : Set = a : S \mid b : S \mid c : S.$$

$$\begin{array}{ll}
\textit{Definition } neq\_S = \lambda x, y\!:\!S\ .\ <\!Prop\!>\text{Match } x \text{ with} & \\
\qquad\qquad\qquad\qquad\qquad\qquad (<\!Prop\!>\text{Match } y \text{ with } False\ \ True\ \ True) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (<\!Prop\!>\text{Match } y \text{ with } True\ \ False\ \ True) & \\
\qquad\qquad\qquad\qquad\qquad\qquad (<\!Prop\!>\text{Match } y \text{ with } True\ \ True\ \ False) &
\end{array}$$

I.e., $(neq\_S\ x\ y)$ reduces to $False$ if $x{=}y{=}a$ or $x{=}y{=}b$ or $x{=}y{=}c$ and it reduces to $True$ otherwise. It serves to prove the desired inequalities $\sim(a{=}b)$, $\sim(a{=}c)$, etc. Instead of deriving and naming all n^2 lemmas for an n-ary set, we prove the following generalized lemma to derive contradictions:

$$\forall x, y\!:\!S\,.\,(x = y) \rightarrow (neq\_S\ x\ y) \rightarrow \forall P\!:\!Prop\,.\,P$$

Such a lemma is derived for all inductive sets. Suppose the lemma is named *absurd_S*. The latter is then used extensively to resolve goals with an inconsistent equality assumption in the context, say $a{=}b$. The following Coq tactical solves such goals immediately:

$$Apply\ (absurd\_S\ a\ b)\ ;\ [Assumption \mid Simpl\ ;\ Exact\ I] \qquad (5)$$

For invariants that are proved by induction over transition steps, sometimes a majority of the subgoals prove by contradiction because they assume $a{=}b$ for different a and b from an inductive set.

Tacticals Both tactics and tacticals have been used in the proof-checking. Coq tacticals are composed of tactics and they can be used to apply at once a combination of rules. They can also be used to accomplish a limited form of proof search. Such tacticals have been written for five of the invariants in this application. One generic tactical was developed to decompose and investigate several lemmas. After case distinction over 25 cases (initial state and 24 transition steps), the tactical attempts to decompose these cases by elimination of logical connectives until only simple goals are left, where the assertion to prove is an equality assertion. For our invariants, typically some 50-100 simple goals are left then. Many of these are solved automatically by assumption, reflexivity or by means of an inconsistent equality statement in the context. For the invariants above, only a handful of non-trivial goals then remain to be solved by the user.

To achieve a form of search, the tacticals are mainly composed of combinations of the ";" and "*Orelse*" tacticals explained below.

$tactical_1 \;\; ; tactical_2 \;\; ; tactical_3$

This applies $tactical_2$ to the subgoals generated by $tactical_1$ and $tactical_3$ to those that are generated by $tactical_2$.

$tactical_1$ *Orelse* $tactical_2$ *Orelse* $tactical_3$

This tries to apply $tactical_1$. If that fails, $tactical_2$ is applied. If that fails, $tactical_3$ is applied. Coq tactical building blocks are fairly elementary. A definition mechanism or parameterization is not provided. This could be convenient for this application, since it would allow often recurring tacticals like (5) to be written very compactly.

The current Coq tactical language has no variables and pattern matching. As a consequence, tacticals must be tailored to the overall structure of goals if they are used for proof search. Because of this, writing a tactical proof often is as much effort as writing the corresponding tactic proof. Currently, the advantage of such tacticals is mainly that it is easier to adapt them than to adapt tactic proofs: tactical proofs are less affected when invariants or automata are modified.

5 Discussion

The main objectives of this work have been fulfilled: the protocol has been verified and the verification (at least the safety part of it) has been proof-checked. Although the Bounded Retransmission Protocol is small, it is by no means trivial and the efforts involved are considerable. While the PSF specification and simulation activity have been carried out in only two man-weeks, the manual verification took roughly two man-months (including write-up) and the proof-checking took more than three man-months. Part of the latter effort is due to a learning effect. Analysis of the Bounded Retransmission Protocol is not completed: the original protocol has an additional *disconnect* service that allows the sender and the receiver to disrupt an ongoing communication. This service has been neglected here and will be verified later. Apart from this, the protocol as described and verified in this paper contains most of the characteristics of the real protocol. It should be noted however that the model simplifies the real-time aspects of the protocol by the way timers are encoded. We could have modeled these real-time aspects more realistically by using a real-time extension of the I/O automata model (see [5]), but then the verification would have been much more involved.

Importance of the verification The verification has answered a number of questions about the protocol. Foremost, it proves that the data link protocol is free of design errors. An important result of the work is that it has corrected several inconsistencies, ambiguities, and omissions in the semi-formal original specification of the protocol. For instance, the exercise has pinned down the behavior of the *toggle* bit between subsequent messages and has formalized many assumptions that were previously left implicit. In addition, the correctness criterion given in the I/O automaton model formalizes the protocol service requirements, i.e., the required external behavior of the protocol.

The automaton specification also serves as a precise functional description for protocol implementations. In this description, all kind of important questions for implementors have been answered, like : “Can I send an empty message?”, “How to respond if a request comes before the previous request is completed?”, “What is the start value of the *toggle* bit for subsequent messages?”. These issues are important if protocol implementations have to be developed by different programmers at different locations, as is the case with this protocol.

Other protocol properties are confirmed by the automaton model. For instance, invariant $INVK'$ (Lemma 5) proves that the use of the bit named *first* in data frames is redundant, because the receiver can always predict its value. This is consistent with the situation in the X.25 LAPB protocol [6] that has no comparable field and that uses a *more_data* bit only, which corresponds to the (inverted) bit named *last* in the Bounded Retransmission Protocol. Further, the automaton model confirms that the *first*, *last* and *toggle* bits from the header of acknowledgements are irrelevant for correctness.

Proof-checking with Coq The experiences with the Coq system are positive. The Coq system 5.8 is robust and reliable and is well-documented. Most shortcomings are related to the ASCII interface: it is easy to lose the overall picture when dealing with large contexts and large proofs.

The Coq proof-checking confirms that the verification is correct. It was not first-time right though and the proof-checking has corrected a number of draft versions. Both the verification and the specification have been revised several times. Other corrections relate to various errors and inaccuracies in versions of the manuscript proof. Preliminary versions of six invariants required modification. One invariant proved false and required weakening. In four cases the original invariants were probably valid but the proof-checking revealed that they needed strengthening (induction loading) to admit a proof. In several cases small modifications to the automaton were necessary to admit a missing proof. Much of the checking was done while parts of the proof were still under development and certain errors must therefore be ascribed to the iterative approach that characterizes the development of automata proofs. Usually the manuscript proof was followed, unless obvious simplifications were seen. For one invariant the use of tacticals simplified a handwritten proof by abstaining from the application of two other invariants that were used in the manuscript proof.

If this application is characteristic of I/O automata proofs — and this seems to be the case — then I/O automata verifications could benefit from proof search procedures. Many (sub-)proofs are truly elementary. It must be stressed that this quality does not come for free. In I/O automata verifications the crucial and most difficult part is *finding* the proper automata, the weak refinement relation and the invariants. This is an iterative process that can benefit from proof search support. Proof search can be used in two ways: it can speed up the checking of manuscript proofs but it can also speed up their development. The Coq system is currently designed as a proof-checker and not as a theorem prover. Accordingly, the system was used in this exercise to check versions of the manuscript proof and the system was not explicitly exploited in the development of the proof. The tacticals written for this application indicate that it is feasible to reduce conjectures of invariants to a few non-trivial or impossible subgoals for the user. Most proof obligations in this application require very specific and elementary reasoning. It seems that additional tactical building blocks can be of great help for future I/O automata verifications. Such tacticals can facilitate the proof-checking but they may also be used in the development of the proofs.

Modeling I/O automata in type theory Modeling the Bounded Retransmission Protocol automata, the invariants and the weak refinement proof in type theory (Coq's Inductive Calculus of Constructions) posed no problem. The translation into type theory that has been used skips much of the generic notions of I/O automata introduced in Section 3.1, like action signatures and explicit sets of states and transitions, but instead directly encodes these notions for this particular application. An important question is if this encoding is satisfactory or how it can be improved upon. An advantage of the current mapping to type theory is that it closely follows the application and directly supports the checking of the invariants and the refinement proof. While this encoding thus facilitates the operational checking, it also amalgamates the automata theory and the application which makes it difficult to reuse much of the Coq text for other applications.

An interesting option is to use a more general encoding of automata theory, together with a compact application description similar to the specification in Section 3.2. This can lead to an approach that is more flexible because it allows reuse of the theory part for different applications. Also, the simpler application description is less error-prone. The current encoding is tailored towards the proving of invariants and weak refinement relations. Absence of deadlock has to be defined specifically for this application and cannot be reconstructed easily from the transition steps. In an approach that explicitly models the meta theory of I/O automata, such properties can be defined independent of the particular application. A disadvantage of that approach is the extra theory level that enforces more elaborate and indirect proofs. Automatic translation of a combined meta theory encoding together with a particular application description into one application-specific encoding seems desirable, in order to obtain the advantages of the latter. The translation can be within Coq or part of a preprocessor. One may even want to use different translations for different purposes. Some of these options are currently investigated by the authors.

Related work Recently, there has been a growing interest in proof-checking protocol correctness proofs, see for instance [4, 8]. Since it is impossible to give here a complete overview of all the work in this area, we will only mention some papers that are directly related to our work, either by the choice of the concurrency formalism or by the choice of the proof-checking system.

Nipkow [23] verified two implementations of a memory system and a mutual exclusion algorithm using the theorem prover Isabelle [25]. The verifications were done both in a setting of algebraic data types (using data refinement) and in the I/O automaton model (using simulation relations). Loewenstein and Dill [17] verified a multiprocessor cache protocol using simulation relations and HOL, the Higher-Order Logic of [11]. This case study is similar in spirit to the one of Nipkow but more involved. Engberg, Grønning and Lamport [10] report on a tool that translates proofs in Lamport's Temporal Logic of Actions to input for LP, the Larch Prover of [13]. A few simple examples were verified using the tool, including a spanning tree algorithm. In these examples the mechanically checkable proofs written in the translator were only two to three times longer than careful hand proofs. Søgaard-Andersen et.al. [26] formalized a simple I/O automata verification of a communication protocol using the LP verification system. They report that, after all the basic machinery of the I/O automata model has been formalized, as well as the basic data types employed by the protocol, the use of LP even leads to a reduction in the size of the proofs. However, their example is quite simple (there is no need to establish state invariants) and it remains to generalize these results to larger examples. Bezem and Groote [3] have used Coq to check a verification of the alternating bit protocol in process algebra. Their proofs are essentially based on rewriting. Recently, Groote and Van de Pol [12] have also verified the Bounded Retransmission Protocol in process algebra using Coq. Whether one prefers process algebra or the I/O automata model appears to be a matter of taste, and in order to evaluate the relative merits of both approaches we will have to consider more and bigger examples. Martin Hofmann [14] in Edinburgh has checked a verification of the Alternating Bit Protocol with LEGO [18]. His verification is based on a functional approach and uses stream transformers.

All of the above researchers arrive at approximately the same conclusion: mechanically checking of protocol verifications is feasible and highly promising, but the current proof-checkers are not optimal: we need an improved user interface (along the lines of [10]) and better proof search procedures.

Acknowledgements

The authors wish to thank Thijs Winter for specifying and simulating the protocol. Doeko Bosscher investigated the state space of the protocol using the Concurrency Workbench [7], and in this way corrected an error in an early version of the specification. Final gratitude goes to Jan Friso Groote, Rob Jansen and Hans Oerlemans for several discussions on the subject.

References

1. K. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
2. H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, pages 117–309. Oxford University Press, 1992.
3. M. Bezem and J. Groote. A formal verification of the alternating bit protocol in the calculus of constructions. Logic Group Preprint Series 88, Dept. of Philosophy, Utrecht University, Mar. 1993.
4. G. v. Bochmann and D. Probst, editors. *Proceedings of the 4th International Conference on Computer Aided Verification*, volume 663 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.
5. D. Bosscher, I. Polak, and F. Vaandrager. Verification of an audio control protocol. Report CS-R94XX, CWI, Amsterdam, 1994. In preparation.
6. CCITT Fascicle VIII.3. *CCITT Recommendation X.25. Interface between DTE and DCE for Terminals Operating in the Packet Mode on Public Data Networks*, 1988.
7. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 1(15):36–72, 1993.
8. C. Courcoubetis, editor. *Proceedings of the 5th International Conference on Computer Aided Verification,* Elounda, Greece, June/July 1993, volume 697 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
9. G. Dowek, A. Felty, H. Herbelin, G. 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.
10. U. Engberg, P. Grønning, and L. Lamport. Mechanical verification of concurrent systems with TLA. In Bochmann and Probst [4].
11. M. Gordon. HOL: a proof generating system for higher-order logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1988.
12. J. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. Logic Group Preprint Series 100, Dept. of Philosophy, Utrecht University, Oct. 1993.

13. J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.
14. M. Hofmann. *Extensional Concepts in Intensional Type Theory.* PhD thesis, University of Edinburgh, 1994. Forthcoming.
15. B. Jonsson. *Compositional Verification of Distributed Systems.* PhD thesis, Department of Computer Systems, Uppsala University, 1987. DoCS 87/09.
16. L. Lamport. How to write a proof. Research Report 94, Digital Equipment Corporation, Systems Research Center, Feb. 1993.
17. P. Loewenstein and D. Dill. Verification of a multiprocessor cache protocol using simulation relations and higher-order logic (summary). In E. Clarke and R. Kurshan, editors, *Proceedings of the 2nd International Conference on Computer-Aided Verification,* New Brunswick, NJ, USA June 1990, volume 531 of *Lecture Notes in Computer Science*, pages 302–311. Springer-Verlag, 1991.
18. Z. Luo, R. Pollack, and P. Taylor. How to use LEGO. Technical Report LFCS-TN-27, University of Edinburgh, Edinburgh, Scotland, Oct. 1989.
19. N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the* 6^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151, Aug. 1987. A full version is available as MIT Technical Report MIT/LCS/TR-387.
20. N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, Sept. 1989.
21. N. Lynch and F. Vaandrager. Forward and backward simulations – part I: Untimed systems. Report CS-R9313, CWI, Amsterdam, Mar. 1993.
22. S. Mauw and G. Veltink, editors. *Algebraic Specification of Communication Protocols.* Cambridge Tracts in Theoretical Computer Science 36. Cambridge University Press, 1993.
23. T. Nipkow. Formal verification of data type refinement — theory and practice. In J. de Bakker, W. d. Roever, and G. Rozenberg, editors, *Proceedings REX Workshop on Stepwise Refinement of Distributed Systems: Models, Formalism, Correctness,* Mook, The Netherlands, May/June 1989, volume 430 of *Lecture Notes in Computer Science*, pages 561–591. Springer-Verlag, 1990.
24. C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J. Groote, editors, *Proceedings of the* 1^{st} *International Conference on Typed Lambda Calculi and Applications, TCLA'93,* Utrecht, The Netherlands, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.
25. L. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*. Academic Press, 1989.
26. J. Søgaard-Andersen, S. Garland, J. Guttag, N. Lynch, and A. Pogosyants. Computer-assisted simulation proofs. In Courcoubetis [8], pages 305–319.
27. A. Tanenbaum. *Computer networks.* Prentice-Hall International, Englewood Cliffs, 1981.

This article was processed using the LaTeX macro package with LLNCS style