The Bounded Retransmission Protocol Must Be on Time!

P.R. D'Argenio[a]*, J.-P. Katoen[b], T.C. Ruys[a], and J. Tretmans[a]

[a] Faculty of Computing Science. University of Twente.
P.O.Box 217. 7500 AE Enschede. The Netherlands.
{dargenio,ruys,tretmans}@cs.utwente.nl

[b] Lehrstuhl für Informatik VII. University of Erlangen.
Martensstrasse 3. 91058 Erlangen. Germany.
katoen@informatik.uni-erlangen.de

Abstract. This paper concerns the transfer of files via a lossy communication channel. It formally specifies this file transfer service in a property-oriented way and investigates—using two different techniques—whether a given bounded retransmission protocol conforms to this service. This protocol is based on the well-known alternating bit protocol but allows for a bounded number of retransmissions of a chunk, i.e., part of a file, only. So, eventual delivery is not guaranteed and the protocol may abort the file transfer. We investigate to what extent real-time aspects are important to guarantee the protocol's correctness and use SPIN and UPPAAL model checking for our purpose.

1 Introduction

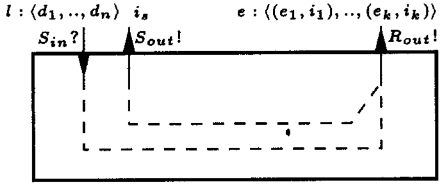
This paper concerns a file transfer service (FTS) and a given bounded retransmission protocol (BRP), a protocol used in one of Philips' products. It addresses the correctness of the BRP with respect to the FTS. The BRP is based on the well-known alternating bit protocol but is restricted to a bounded number of retransmissions of a chunk, i.e., part of a file. So, eventual delivery is not guaranteed and the protocol may abort the file transfer. Timers are involved in order to detect the loss of chunks and the abortion of transmission. The protocol verification is carried out by model checking. This technique facilitates the automatic verification of properties, usually stated in some dialect of modal logic, with respect to a protocol specified as a finite-state system. The tools used in this paper are SPIN [13] for untimed and UPPAAL [4] for timed systems.

The FTS is specified in a property-oriented way by providing relations between inputs and outputs of the service. This is done without using modal operators. We validate the consistency of this logical service specification against the process algebraic "external behavior" specification of [9]. The BRP is modeled as a network of timed automata that communicate via handshaking (like in CCS). This results in a compact and intuitively appealing protocol specification. Using

* Supported by the NWO/SION project 612-33-006.

UPPAAL we verify the correctness of the protocol by proving that it satisfies a number of properties, specified as logical formulas. We indicate the importance of real-time aspects for the correctness of the BRP. This complements the untimed BRP verifications of [9, 10, 11, 14] that focussed on the data aspects of the BRP. To investigate and compare the relevance of the modeling assumptions made by others we check, using SPIN, the correctness of our protocol description when omitting the timing aspects. Due to the recent improvements of UPPAAL this paper contains substantially more complete verifications than reported earlier by us [6]. In particular, we could obtain tight constraints on the timing aspects of the BRP. The full report of this work appeared in [7].

2 Service specification


$l : \langle d_1, .., d_n \rangle$
i_s
$e : \langle (e_1, i_1), .., (e_k, i_k) \rangle$
$S_{in}?$
$S_{out}!$
$R_{out}!$

Schematic view of the FTS.

Signatures of the input and output:

$S_{in} : l = \langle d_1, \ldots, d_n \rangle$ for $n > 0$
$S_{out} : i_s \in \{\text{I\_OK}, \text{I\_NOK}, \text{I\_DK}\}$
$R_{out} : \langle (e_1, i_1), \ldots, (e_k, i_k) \rangle$ for $0 \leqslant k \leqslant n$,
$i_j \in \{\text{I\_FST}, \text{I\_INC}, \text{I\_OK}, \text{I\_NOK}\}$
for $0 < j \leqslant k$

The FTS receives a large file from a sending client via port S_{in}. The file is modeled as a list of small data chunks: $l = \langle d_1, \ldots, d_n \rangle$ with $n > 0$. The FTS delivers the data to the receiving client as a list of chunks $e_1, \ldots, e_k$. Chunks may get lost, but are neither garbled nor received out of order. If a chunk is lost, transfer is aborted. So the receiving client receives a (possibly empty, possibly all) prefix of the list l. Both the sender and the receiver client obtain indications about the success of the transfer. The sending client receives an indication i_s via S_{out}, while the receiving client receives an indication i_j with each chunk that is delivered via R_{out}. Multiple file can be transfered with the FTS and we assume that all outputs regarding previous lists have been completed when a next list is input via S_{in}.

Table 1 specifies the FTS in a logical way, i.e., by stating properties that should be satisfied by the service. These properties are relations between inputs at S_{in} and outputs at S_{out} and R_{out}. A distinction is made between the case in which the receiving client receives at least one chunk ($k > 0$) and the case that it receives none ($k = 0$). A protocol (like the BRP) conforms to the FTS if it satisfies all listed properties.

For $k > 0$ we have the following requirements. Normally each correctly received chunk e_j equals d_j, the chunk sent via S_{in}. However, if the notification i_j indicates that an error occurred, no restriction is imposed on the accompanying chunk, cf. **(1.1)**. **(1.2)** through **(1.4)** address the constraints concerning the received indications via R_{out}, i.e., i_j. If the number n of chunks in l exceeds one then $i_1 = \text{I\_FST}$, indicating that e_1 is the first chunk of the file and more will follow, cf. **(1.2)**. The indications of all chunks, apart from the first and last chunk, should equal I_INC, cf. **(1.3)**. The requirement concerning the last chunk

Table 1. Formal specification of the FTS.

	$k > 0$		
(1.1)	$\forall 0 < j \leqslant k : i_j \neq \mathsf{I\_NOK} \Rightarrow e_j = d_j$	**(1.4.3)**	$i_k = \mathsf{I\_NOK} \Rightarrow k > 1$
(1.2)	$n > 1 \Rightarrow i_1 = \mathsf{I\_FST}$	**(1.5)**	$i_s = \mathsf{I\_OK} \Rightarrow i_k = \mathsf{I\_OK}$
(1.3)	$\forall 1 < j < k : i_j = \mathsf{I\_INC}$	**(1.6)**	$i_s = \mathsf{I\_NOK} \Rightarrow i_k = \mathsf{I\_NOK}$
(1.4.1)	$i_k = \mathsf{I\_OK} \vee i_k = \mathsf{I\_NOK}$	**(1.7)**	$i_s = \mathsf{I\_DK} \Rightarrow k = n$
(1.4.2)	$i_k = \mathsf{I\_OK} \Rightarrow k = n$		
	$k = 0$		
(2.1)	$i_s = \mathsf{I\_DK} \Leftrightarrow n = 1$	**(2.2)**	$i_s = \mathsf{I\_NOK} \Leftrightarrow n > 1$

(e_k, i_k) consists of three parts. After this indication, the FTS is ready to transmit a subsequent list. Indication i_k can be either I_OK or I_NOK, cf. **(1.4.1)**. An I_OK indicates that all chunks have been received correctly, so $k = n$, see **(1.4.2)**. The receiving client does not need to be notified in case an error occurs before delivery of the first chunk, cf. **(1.4.3)**. The sending client is informed after transfer of the whole file, or when the transmission is aborted. This indication can be I_OK, I_NOK, or I_DK. After an I_OK or an I_NOK indication, the sender can be sure, that the receiver has the corresponding indication. This corresponds to **(1.5)** and **(1.6)**. A "don't know" indication I_DK may occur after delivery of the last-but-one chunk d_{n-1}. This situation arises, because no realistic implementation can ensure whether the last chunk got lost. The reason is that information about a successful delivery has to be transported back somehow over the same unreliable medium. In case the last acknowledgment fails to come, there is no way to know whether the last chunk d_n has been delivered or not. In this case the number of indications received by the receiving client must equal n, see **(1.7)** (Either this last chunk is received correctly or not, and in both cases an indication (+ chunk) is present at R_{out}.)

For $k = 0$ the sender should receive an indication I_DK if and only if the file to be sent consists of a single chunk. This corresponds to the fact that a "don't know" indication may occur after the delivery of the last-but-one chunk only. For $k = 0$ the sender is given an indication I_NOK if and only if n exceeds one.

3 Protocol specification

Informal description. The protocol consists of a sender S equipped with a timer T_1, and a receiver R equipped with a timer T_2 which exchange data via two unreliable (lossy) channels, K and L.

Sender S reads a file to be transmitted and sets the retry counter to 0. Then it starts sending the elements of the file one by one over K to R. Timer T_1 is set and a frame is sent into channel K. This frame consists of three bits and a datum (= chunk). The first bit indicates whether the datum is the first element of the file. The second bit indicates whether the datum is the last item of the file. The third bit is the so-called alternating bit, that is used to guarantee that data is not duplicated. After having sent the frame, the sender waits for an acknowledgment from the receiver, or for a timeout. In case an acknowledgment arrives, the timer T_1 is reset and (depending on whether this was the last element of the file) the sending client is informed of correct transmission, or the next element of the file is sent. If timer T_1 times out, the frame is resent (after the counter for the number of retries is incremented and the timer is set again), or the transmission of the file is broken off. The latter occurs if the retry counter exceeds its maximum value MAX.

Receiver R waits for a first frame to arrive. This frame is delivered at the receiving client, timer T_2 is started and an acknowledgment is sent over L to S. Then the receiver simply waits for more frames to arrive. The receiver remembers whether the previous frame was the last element of the file and the expected value of the alternating bit. Each frame is acknowledged, but it is handed over to the receiving client only if the alternating bit indicates that it is new. In this case timer T_2 is reset. Note that (only) if the previous frame was last of the file, then a fresh frame will be the first of the subsequent file and a repeated frame will still be the last of the old file. This goes on until T_2 times out. This happens if for a long time no new frame is received, indicating that transmission of the file has been given up. The receiving client is informed, provided the last element of the file has not just been delivered. Note that if transmission of the next file starts before timer T_2 expires, the alternating bit scheme is simply continued. This scheme is only interrupted after a failure.

Timer T_1 times out if an acknowledgment does not arrive "in time" at the sender. It is set when a frame is sent and reset after this frame has been acknowledged. (Assume that premature timeouts are not possible, i.e., a message must not come *after* the timer expires.)

Timer T_2 is (re)set by the receiver at the arrival of each new frame. It times out if the transmission of a file has been interrupted by the sender. So its delay must exceed MAX times the delay of T_1.[2] Assume that the sender does not start reading and transmitting the next file before the receiver has properly reacted to the failure. This is necessary, because the receiver has not yet switched its alternating bit, so a new frame would be interpreted as a repetition.

This completes the informal description of the BRP (as adopted from [9]). It is important to note that two significant assumptions are made in the above description, referred to as **(A1)** and **(A2)** below.

(A1) Premature timeouts are not possible

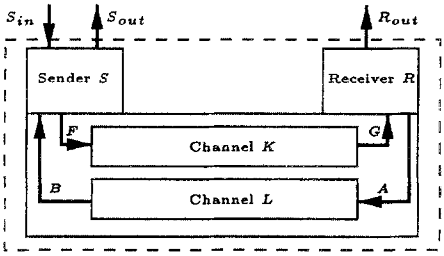
Let's suppose that the maximum delay in the channel K (and L) is TD and that timer T_1 expires if an acknowledgment has not been received within T1 time units since the first transmission of a chunk. Then this assumption requires that $\mathsf{T1} > 2 \cdot \mathsf{TD} + \delta$ where δ denotes the processing time in the receiver R. **(A1)** thus requires knowledge about the processing speed of the receiver.

(A2) In case of abort, S waits before starting a new file
until R reacted properly to abort

[2] Later on we will show that this lowerbound is not sufficient.

Since there is no mechanism in the BRP that notifies the expiration of timer T_2 (in R) to the sender S this is a rather strong and unnatural assumption. It is unclear how S "knows" that R has properly reacted to the failure, especially in case S and R are geographically distributed processes. We, therefore, consider **(A2)** as an unrealistic assumption. In the next section we ignore this assumption and adapt the protocol slightly such that this assumption appears as a property of the protocol (rather than as an assumption!).

Formal specification. The BRP consists of a sender S and a receiver R communicating through channels K and L, see the figure below. S sends chunk d_i via F to channel K accompanied with an alternating bit ab, an indication b whether d_i is the first chunk of a file (i.e., $i = 1$), and an indication b' whether d_i is the last chunk of a file (i.e., $i = n$). K transfers this information to R via G. Acknowledgments ack are sent via A and B using L.



Schematic view of the BRP.

The signatures of A, B, F, and G are:

$$
\begin{aligned}
F, G &: (b, b', ab, d_i)\\
&\quad \text{with } ab \in \{0, 1\},\\
&\quad\quad b, b' \in \{\text{true}, \text{false}\}\\
&\quad\quad \text{and } 0 < i \leqslant n\\
A, B &: ack
\end{aligned}
$$

Our starting-point for modeling and verifying the BRP is a specification of the BRP in terms of a network of timed automata. A timed automaton [1] is a classical finite-state automaton equipped with *clock variables* and *state invariants.* The state of a timed automaton is determined by the system variables and clock variables. The value of a system variable is changed explicitly by an assignment that is carried out at a transition; the value of clock variables increases implicitly as time advances. A state invariant constrains the amount of time the system may idle in a state. Clock values may be tested (i.e., compared with naturals) and reset. In the sequel we will use u through z to denote clock variables.

A network of timed automata consists of a number of processes (modeled as timed automata) that communicate with each other in a CCS-like manner.

Transitions consist of an (optional) guard and zero or more actions. Depending on the validity of the guard a transition is either *enabled* or *disabled.* In a state the process selects non-deterministically between all enabled transitions, it performs the (possibly empty) set of actions associated with the selected transition and goes to the next state. When there are no enabled transitions the process remains in the same state (if allowed by the state invariant) and time passes implicitly. If neither idling is allowed nor an enabled transition can be taken, the system halts. Evaluation of a guard, taking a transition and executing its associated actions constitutes a single *atomic* event. *Guards* are boolean expressions and may contain system and clock variables. For convenience, guards

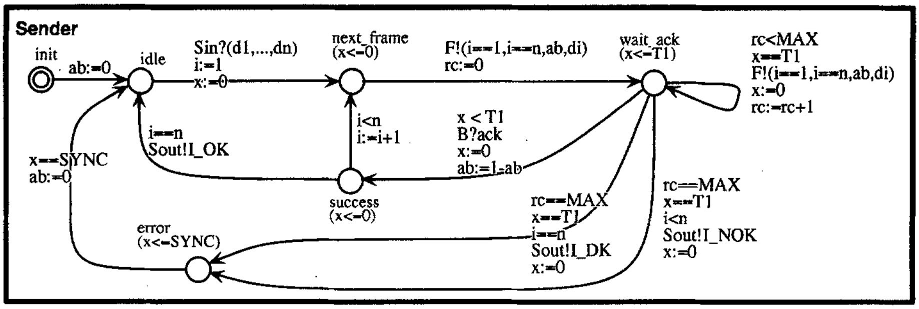


Fig. 1. Timed automaton for sender S.

that are equal to true are omitted. Possible *actions* are assignments to system variables and resetting of clock variables.

We adopt the following notational conventions. States are represented by labeled circles, the initial state as double-lined labeled circle. State invariants are denoted in brackets. Transitions are denoted by directed, labeled arrows. A list of guards denotes the conjunction of its elements.

Channels K and L are simply modeled as first-in first-out queues of unbounded capacity with possible loss of messages. We assume that the maximum latency of both channels is TD time units.

The sender S (see Figure 1) has three system variables: $ab \in \{0, 1\}$ indicating the alternating bit that accompanies the next chunk to be sent, i, $0 \leqslant i \leqslant n$, indicating the subscript of the chunk currently being processed by S, and rc, $0 \leqslant rc \leqslant \mathsf{MAX}$, indicating the number of attempts undertaken by S to retransmit d_i. Clock variable x is used to model timer T_1 and to make certain transitions urgent (see below). The reader can check that the automaton in Figure 1 behaves as it was explained above.

Two remarks are in order. First, notice that transitions leaving state s, say, with state invariant $x \leqslant 0$ are executed without any delay with respect to the previous performed action, since clock x equals 0 if s is entered. Such transitions are called *urgent*. Urgent transitions forbid S to stay in state s arbitrarily long and avoid that receiver R times out without abortion of the transmission by sender S. Urgent transitions will turn out to be necessary to achieve the correctness of the protocol. They model a maximum delay on processing speed, cf. assumption (**A1**). Secondly, after a failure (i.e., S is in state *error*) an additional delay of SYNC time units is incorporated. This delay is introduced in order to ensure that S does not start transmitting a new file before the receiver has properly reacted to the failure. This timer will make it possible to satisfy assumption (**A2**). In case of failure the alternating bit scheme is restarted.

The reader can check that the receiver specified in Figure 2 represents the behaviour explained in the previous paragraph. System variable *exp_ab* $\in \{0, 1\}$ in receiver R models the expected alternating bit. Clock z is used to model timer T_2 that determines transmission abortions of sender S, while clock w is used to make some transitions urgent. Notice that if timer z expires (i.e., z = TR), it may be the case that the communication has been lost. Thus, in case R did not just receive the last chunk of a file an indication I_NOK (accompanied with an

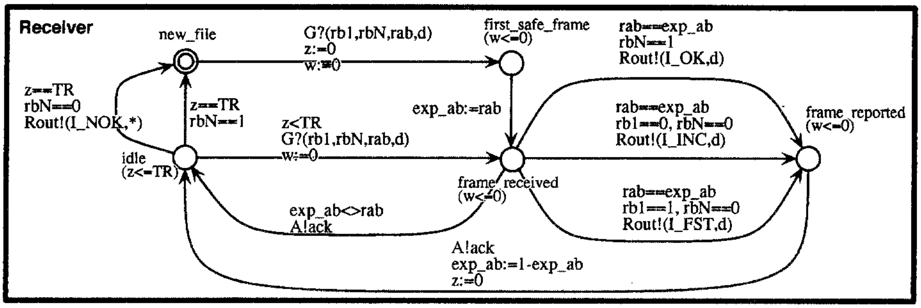


Fig. 2. Timed automaton for receiver *R*.

arbitrary chunk "*") is sent via R_{out} indicating a failure, and in case *R* just received the last chunk, no failure is reported.

Most of the transitions in *R* are made urgent in order to be able to fulfill assumption **(A1)**.

4 UPPAAL

UPPAAL [4] is a tool suite for symbolic model checking of real-time systems. Systems in UPPAAL are described as networks of timed automata [1] like described in Section 3. UPPAAL reduces the verification problem to solving a (simple) set of constraints on clock variables. Experimental results indicate that these techniques have a good performance (both in space and time) compared to other verification techniques for timed automata [4]. The UPPAAL verifier can be used to determine the satisfaction of a given property with respect to a network of timed automata. If a property is not satisfied, a diagnostic trace can be generated that indicates how the property is violated. UPPAAL also provides a simulator that allows a graphical visualization of possible dynamic behaviours of a system description (i.e., a symbolic trace). This last tool becomes powerful when combined with the diagnostic information provided by the verification tool.

In the current version (i.e., β-release 1.91), UPPAAL is able to check only reachability properties. Properties are terms of the form $\forall\Box\phi$ or $\exists\Diamond\phi$ where ϕ is a propositional formula with atoms being either a state of a component, i.e., one of the states of any of the timed automata, or a simple linear constraint on clocks or integer variables. The use of data in UPPAAL 1.91 is restricted to clocks and integers (rather than system variables of arbitrary type) and value passing at synchronization is not supported (but can be mimicked using shared variables).

Protocol model in UPPAAL. The UPPAAL models of sender *S* and receiver *R* are a straightforward adaptation of the specifications given in Section 3; see Figure 3. (Nomenclature is similar to Section 3 except for minor changes; e.g., *Sout_I_OK?* instead of *Sout?I_OK*). Channels *K* and *L* are reduced from unbounded queues to one-place buffers. Below we will derive a constraint under which this simplification is justified. In addition, the following considerations have been taken into account.

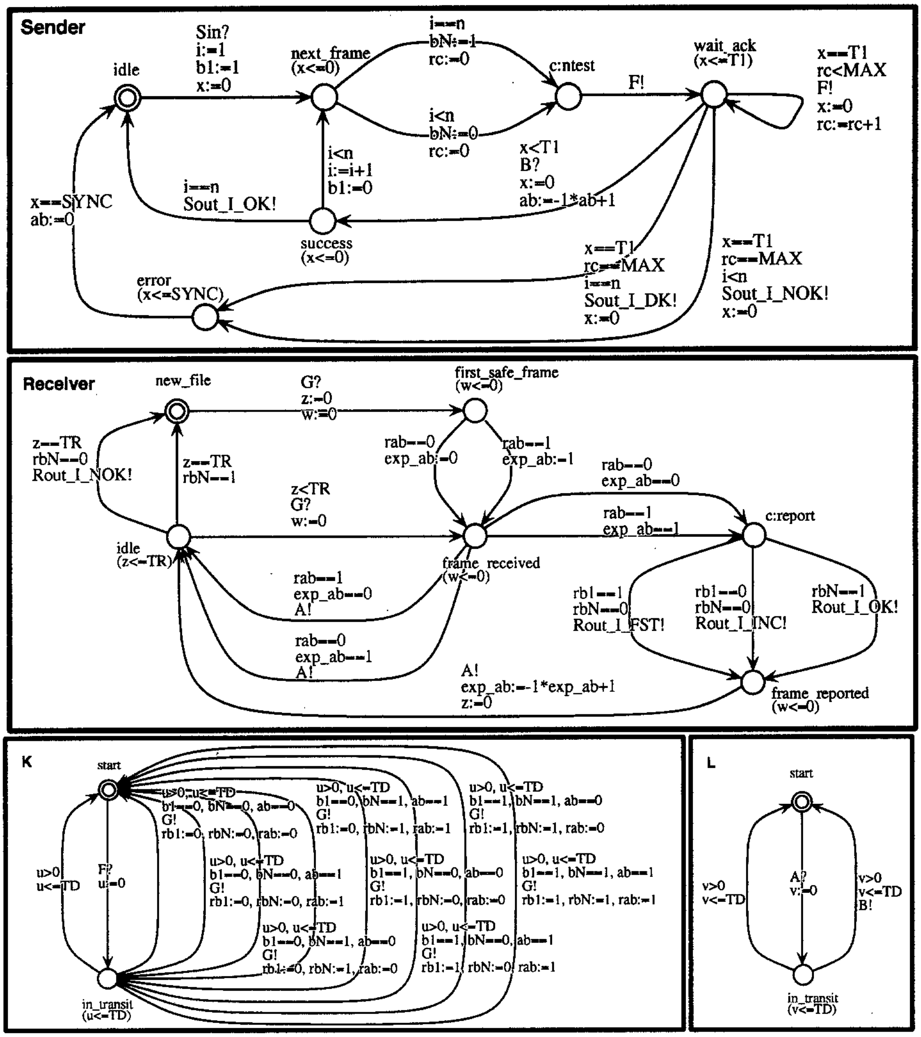


Fig. 3. The protocol in UPPAAL.

Guards in UPPAAL (i.e., constraints labeling the transitions) are conjunctions of atomic constraints which have the form $x \sim n$ where x is a variable (a clock or an integer), n a non-negative integer, and $\sim \in \{<, \leqslant, =, \geqslant, >\}$[3]. Thus, conditions like $rab \neq exp\_ab$ are not possible. For this reason some transitions are splitted (compare, for example, the acknowledgement transitions outgoing from state *frame_received* in the receiver R of Figure 2 and 3).

We have said that UPPAAL is not a data-oriented tool. If we had included data in our model, we would have had an explosion of the amount of states and transitions. This induces two main problems. Firstly, the original simple speci-

[3] Notice that in Figure 3, values like n, TD or MAX have not yet been instantiated, but they must get a concrete value for each UPPAAL verification run. (See the last paragraph of this section.)

fication would become too cumbersome and quite difficult to understand. Secondly, although UPPAAL uses quite efficient compositional techniques to attack the problem of state space explosion (or, more accurate, region space explosion), it is anyway sensible to the amount of locations, clocks, and variables. Therefore, we decided to remove the chunks to be transmitted keeping only the control data, i.e., the indication of the first and last chunk, and the alternating bit.

In UPPAAL assignments to clock x should be of the form $x := 0$, while assignments to integer variable i must have the form $i := n_1 * i + n_2$. Notice that for the latter assignments the variable on the right-hand side of the assignment should be the same as the variable on the left-hand side. UPPAAL does also not include mechanisms for value passing. We modeled value passing by means of assignments. Due to the above mentioned restriction on integer assignments, however, we had to explode some transitions. For example, for channel K a transition had to be introduced for each combination of values for $b1$, bN, and ab that can be received via G; this resulted in 8 transitions, see Figure 3.

We use the so-called *committed locations* [3]. Committed locations are states which introduce the notion of atomicity. On the one hand, a committed location forbids interference in the activity that is taking place around such a location, i.e., the execution of the ingoing and outgoing actions of a committed location cannot be interleaved with actions of other timed automata. On the other hand, actions outgoing from a committed location are executed urgently, that is, no time elapses between its execution and the execution of the previous action. We made locations *R.report* and *S.ntest* committed (indicated with a *c:* prefix) since they originate from splitting transitions of the original specification (compare with Figures 1 and 2).

Deducing time constraints. In this section we derive a constraint under which the modeling of channels K and L as one-place buffers is justified. In addition, we present timing conditions under which assumptions (**A1**) and (**A2**) are fulfilled.

To justify the simplified modeling of K and L we slightly changed the channels by adding location *BAD*. This location can only be reached when in location *in_transit* a new message is out in the channel. Location *BAD* can thus only be reached if the channel capacity is insufficient, that is, when S and R are sending messages to K and L, respectively, too fast. We could check that a *BAD* state is never reached, i.e., properties

$$\forall\Box\ \neg K.BAD \qquad \text{and} \qquad \forall\Box\ \neg L.BAD$$

are satisfied, only under the condition that $\mathsf{T1} > 2 \cdot \mathsf{TD}$. Moreover, under this condition, the following property

$$\forall\Box\ \neg(K.\mathit{in\_transit}\ \wedge\ L.\mathit{in\_transit}) \qquad (1)$$

could be verified. This means that under the condition that $\mathsf{T1} > 2 \cdot \mathsf{TD}$, it is impossible to have both a frame and an acknowledgment in transit at the same time. This property is of interest, since it allows one to verify the protocol more efficiently by changing the process $K \,||\, L$, where $||$ denotes independent parallelism, into a smaller process with one state and one clock less.

Assumption **(A1)** states that no premature timeouts should occur. It can easily be seen that timer T_1 (i.e., clock x) of sender S does not violate this if it respects the two-way transmission delay (i.e., $\mathsf{T1} > 2 \cdot \mathsf{TD}$) plus the processing delay of the receiver R (which due to the presence of urgency equals 0). It remains to be checked under which conditions timer T_2 of receiver R does not generate premature timeouts. This amounts to checking that R times out whenever the sender has indeed aborted the transmission of the file. Observe that a premature timeout appears in R if it moves from state *idle* to state *new_file* although there is still some frame of the previous file to come. We therefore check that in state *first_safe_frame* receiver R can only receive first chunks of a file (i.e., $rb1 = 1$) and not remaining ones of previous files:

$$\forall\Box\ (R.\mathit{first\_safe\_frame}\ \Rightarrow\ rb1 = 1) \tag{2}$$

We have checked that this property holds whenever $\mathsf{TR} \geqslant 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$.

Assumption **(A2)** states that sender S starts the transmission of a new file only after R has properly reacted to the failure. For our model this means that if S is in state *error*, eventually, within SYNC time units, R resets and is able to receive a *new_file*. Although this cannot be expressed in UPPAAL logic, we can check a weaker property:

$$\forall\Box\ ((S.\mathit{error}\ \wedge\ x = \mathsf{SYNC})\ \Rightarrow\ R.\mathit{new\_file}) \tag{3}$$

This property turns aut to be equivalent to what we want to prove since the sender S keeps idling in state *error* while clock x evolves from 0 to SYNC. Property (3) is satisfied under the condition that $\mathsf{SYNC} \geqslant \mathsf{TR}$. This means that **(A2)** is only fulfilled if this condition on the values SYNC and TR is respected.

Summarizing, we were able to check with UPPAAL that assumptions **(A1)** and **(A2)** are fulfilled only if the following constraints hold

$$\boxed{\mathsf{T1} > 2 \cdot \mathsf{TD} \quad \text{and} \quad \mathsf{SYNC} \geqslant \mathsf{TR} \geqslant 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}} \tag{4}$$

(Remark that SYNC is a constant in the sender S, while TR is a constant used in receiver R.) These results show the importance of timing aspects for the correctness of the BRP.

Protocol verification. In order to verify that the protocol satisfies the FTS specification of Section 2, we consider the clients at each side of the protocol and a simple check automaton, called *File*, which indicates whether the receiving client *RC* and the sending client *SC* are dealing with the same file. The *File* process checks the condition $k > 0$. The auxiliary automata are depicted in Figure 4.

When trying to verify the correctness of the BRP using UPPAAL we encounter the following problems. Firstly, the properties constituting the FTS specification of Section 2 are relations between inputs and outputs related to the transmission of a single file. Therefore, these properties are not invariant and can hardly be expressed using the property language of UPPAAL that requires an always ($\Box$) or ever ($\Diamond$) modal operator at "top" level. Secondly, since we decided to remove

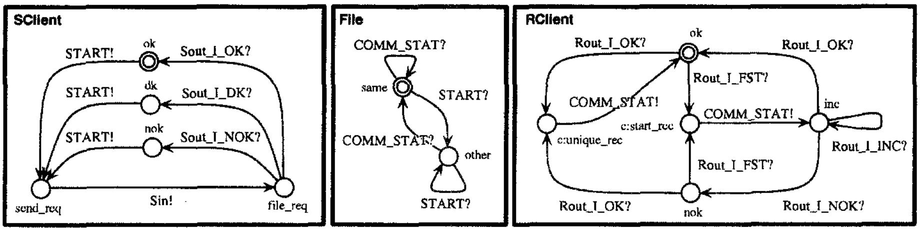


Fig. 4. Auxiliary automata (general).

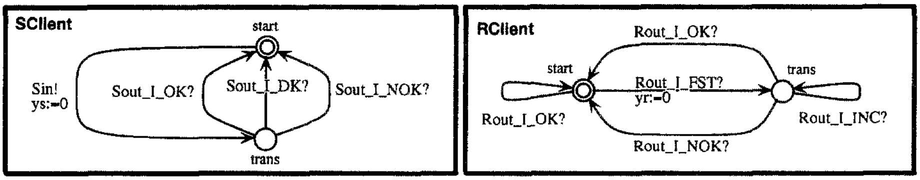


Fig. 5. Auxiliary automata (bounded retransmission).

the data from our specification, we are unable to check properties concerning the transmitted data, like property (**1.1**).

The properties that we checked are enumerated in Table 2. There, we abbreviate $SC.idle = (SC.ok \lor SC.dk \lor SC.nok)$ and $RC.idle = (RC.ok \lor RC.nok)$. Properties 1. and 2. are weakened versions of properties (**1.5**) and (**1.6**), respectively. Property 3. is related to (**2.1**) and (**2.2**). Properties 4. and 5. relate the sender S and the sending client, while 6. relates the receiver R and the receiving client. In particular, when we take $n = 1$ we proved properties 7. and 8. which are related to (**1.4.3**) and (**2.2**).

Properties 9. and 10. address the fact that the sending and receiving client, respectively, are involved in the transfer of a file for only a bounded amount of time (T and T', respectively). For this purpose, we changed the clients according to Figure 5. The clients have only two locations: *trans* indicates that the respective client recognized that a file transfer is currently in progress; *start* is the state in which a file transfer can be started.

The properties were proven in the following setting:

$$\begin{array}{lll} \mathsf{TD} = 1 & \mathsf{MAX} \in \{1,2,3\} & \mathsf{TR} = 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD} \\ \mathsf{T1} = 3 & n \in \{1,2,3\} & \mathsf{SYNC} = \mathsf{TR} \end{array}$$

For properties 9. and 10., and fixing $n = 3$ and $\mathsf{MAX} = 3$, we obtain 34 and 42 as minimal values for T and T' respectively.

Table 2. Properties in UPPAAL.

1. $\forall\Box\ File.same \Rightarrow \neg(SC.ok \land RC.nok)$	6. $\forall\Box\ R.new\_file \Rightarrow RC.idle$
2. $\forall\Box\ File.same \Rightarrow \neg(SC.nok \land RC.ok)$	7. $\forall\Box\ \neg SC.nok$ (if $n = 1$)
3. $\forall\Box\ \neg(File.other \land SC.ok)$	8. $\forall\Box\ \neg RC.nok$ (if $n = 1$)
4. $\forall\Box\ SC.idle \Rightarrow (S.idle \lor S.error)$	9. $\forall\Box\ \neg(SC.trans \land y_s > T)$
5. $\forall\Box\ (S.idle \lor S.error) \Rightarrow \neg SC.file\_req$	10. $\forall\Box\ \neg(RC.trans \land y_r > T')$

5 SPIN

SPIN is a validation tool for classical finite-state automata, called *processes*, that communicate via channels. It is capable of verifying assertions over data and simple linear-time temporal logic formulas (so-called never claims). SPIN uses the dedicated modeling language PROMELA [13]. It is able to perform random or interactive simulations of the system's execution or to generate a C program that performs an exhaustive validation of the system's state space. Large validation runs, for which an exhaustive validation is not feasible, can be validated in SPIN with a *bit-state hashing* technique [13] at the expense of completeness.

To get confidence in the service specification from Section 2, we have written a PROMELA specification for the FTS. Each requirement of Section 2 is translated into a sequence of PROMELA statements involving *assertions*, which are boolean conditions attached to a state that must be fulfilled when a process reaches that state. For example, requirement **(1.1)** is translated into the PROMELA assertion:

```
byte j=0;
do :: j++ ;
      if :: (j>k)  -> break
         :: (j<=k) -> if :: (e[j].ind != Inok) -> assert(e[j].val == d[j])
                         :: else -> skip
                      fi
      fi
od
```

Figure 6 shows an overview of our validation model used in SPIN. Either the FTS or the BRP description in PROMELA can be "plugged" into this model. The *Environment* process inputs the file to be transferred at S_{in} (i.e., the list of chunks) and receives the indications at S_{out} and R_{out}. When all indications of the transmission of a single file have been produced, the *Environment* process checks the validity of the indications.

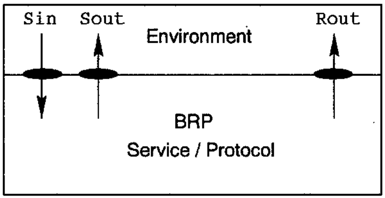


Fig. 6. SPIN's validation model of the BRP and FTS.

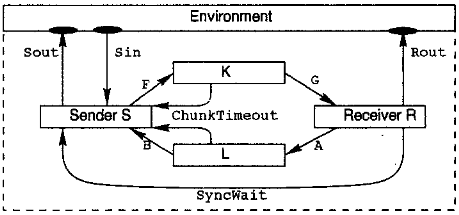


Fig. 7. Structure of the BRP in PROMELA.

The FTS description in PROMELA is obtained by a straightforward translation of the "external behavior" specification of [9] given in the process algebra μCRL. Below we have included the PROMELA `proctype` definition of the *Service* process.

```
proctype Service (chan Sin, Sout, Rout)
{ byte j,k ;
  do :: Sin?(d[1],...,d[n]) -> j=0; k=0;
     do :: j++ ;
           if :: skip -> k++;
                         if :: (j==n) -> Rout!(Iok,d[j])
                            :: (j!=n) && (k==1) -> Rout!(Ifst,d[j])
                            :: (j!=n) && (k>1)  -> Rout!(Iinc,d[j])
                         fi ;
                         if :: (j==n) -> if :: skip -> Sout!Iok; break
                                            :: skip -> Sout!Idk; break
                                         fi
                            :: (j!=n) -> if :: skip -> skip
                                            :: skip -> Sout!Inok; k++; Rout!Inok; break
                                         fi
                         fi
              :: skip -> if :: (k==0) -> if ::  (j==n) -> Sout!Idk
                                            ::  (j!=n) -> Sout!Inok
                                         fi
                            :: (k>0) -> k++;
                                        if :: (j==n) -> Sout!Idk; Rout!Inok
                                           :: (j!=n) -> Sout!Inok; Rout!Inok
                                        fi
                         fi ;
                         break
           fi
     od
  od
}
```

PROMELA can handle data more easily than UPPAAL. UPPAAL, e.g., only supports synchronization of processes without value passing. Therefore, a protocol like the BRP where the transmission of data is crucial, is more easily modeled and can be more extensively verified in PROMELA than in UPPAAL. On the other hand, PROMELA (SPIN version 2.7.7) lacks one important ingredient for the realistic modeling of the BRP: the notion of *time*. The BRP description in PROMELA is based on the formal protocol description from Section 3. Like for our UPPAAL verification we modeled channels K and L as one-place buffers. Figure 7 shows the structure of the BRP PROMELA description in terms of processes (represented as boxes) and channels (represented as arrows).

We used "tricks", analogous to [9] (and others, see Section 6), to model the impact of the timers T_1 and T_2. These "tricks", in fact, are needed to fulfill the assumptions **(A1)** and **(A2)**. Timers T_1 and T_2 that are used in the sender S and receiver R, respectively, are modeled as follows: Timer T_1 expires when an acknowledgment does not arrive in time at the sender S. So, if a frame is lost in channel K or its acknowledgment is lost in channel L, the timer T_1 in S will timeout, eventually. In the PROMELA model, channel ChunkTimeout is used between sender S and channels K and L. A message is either successfully transmitted, or it is lost, in which case S is notified via ChunkTimeout. To illustrate this, we include the PROMELA code for the process that models L:

```
bit b ;
do :: A?b -> if :: B!b
                :: skip -> ChunkTimeout!1
             fi
od
```

Receiver R uses timer T_2 that expires when the transmission of file has been aborted by sender S. According to assumption (**A2**) "the sender does not start reading and transmitting the next file before the receiver has properly reacted to the failure". In UPPAAL, we implemented this assumption using two timers: one at the sender's side and one at the receiver's side. When either one of the timers expires, the process at hand will wait sufficiently long (i.e., SYNC time units) to be sure that the other process has timed out as well. In PROMELA we forced this assumption using a handshake channel SyncWait between processes S and R. After a failure, the failing process will offer a handshake synchronization on this channel. Eventually, the other process will engage in this rendez-vous synchronization.

For the verification of the BRP we used the following parameters: $n = 3$ (with up to 3 different data items) and MAX $= 2$. For the validation of the BRP with PROMELA we used the same computer on which we carried out our UPPAAL verification. This machine proved to be adequate for a complete reachability analysis of the FTS and the protocol model; SPIN managed to explore the complete state space and reported no errors with respect to the properties on Table 1.

6 Further discussions

Other verifications of the BRP. The modeling and verification of the BRP has been the subject of several other papers. Groote & v.d. Pol [9] specify the BRP in μCRL, a combination of process algebra and abstract data types, and prove this specification to be branching bisimulation equivalent (a strong notion of weak bisimulation) to an external behaviour specification, also given in μCRL. Part of the proofs were checked using the proof-assistant Coq.

Helmink, Sellink & Vaandrager [11] analyze the BRP in the setting of I/O-automata, automata that distinguish between input, output, and internal actions and which allow all possible inputs in each state. Refinement, in particular inclusion of fair traces, is used as a correctness criterion. In addition, they prove that the BRP is deadlock-free. The safety part of the proofs were mechanically checked using Coq.

Havelund & Shankar [10] use a combination of model checking and theorem proving techniques for proving the correctness of the BRP. They first analyze a scaled-down version of the BRP using Murϕ, a state exploration tool, 'translate' this description into the theorem prover PVS and generalize the result to the full BRP, and finally, abstract from this complete specification (while preserving some essential properties) so as to facilitate model checking. By means of abstraction the unboundedness of the message data, retransmission bound, and file length is eliminated. They used SMV, Murϕ, and an extension of PVS with the modal μ-calculus for the final model checking.

Mateescu [14] translated the μCRL specifications of [9] into the process algebra LOTOS, and proved that the FTS and BRP specifications are branching bisimulation equivalent using the ALDÉBARAN tool. In addition, he checked some protocol invariants, encoded in ACTL (an action-based variant of CTL), using the prototype model checker XTL.

Our SPIN validation—though using a new logical service specification—can be considered to be similar to all above mentioned approaches, since it focuses on the data aspects of the BRP (as all others). In order to mimic the timers of the BRP in an untimed setting strong assumptions, cf. **(A1)** and **(A2)**, must be made, and "tricks" must be applied in order to fulfill these assumptions. This holds for all untimed verifications discussed above. Our analysis with UPPAAL shows that these assumptions only hold if certain relations between time-out values are established. This shows that timing is crucial for the correct functioning of the BRP.

Concluding remarks. This paper reported on the analysis and verification of a bounded retransmission protocol (BRP) using the protocol validation tool SPIN [13], and the real-time verification tool UPPAAL [4]. We used a Sun Sparc station with 96 MB of internal memory.

UPPAAL was used to check the timed automaton model of the protocol against the FTS requirements. Due to the restrictions of the property language of UPPAAL (e.g., restricted use of variables) some FTS requirements had to be adapted. Apart from some small modifications (basically due to the use of committed locations and restrictions on conditions, variables, and value passing in UPPAAL) the protocol model could be obtained in a straightforward way from our formal BRP specification of Section 3. We were able to find tight constraints under which the unbounded channels between the sending and receiving side can be faithfully modeled as one-place buffers. Most importantly, we could show that two assumptions in the informal protocol description are easily invalidated by choosing wrong time-out values. We provided tight timing constraints for which these assumptions are fulfilled (and so, they become valid assumptions).

With SPIN both the service and the protocol were verified. For the service we checked our requirements specification against a behavioral model in PROMELA (the modeling language for SPIN), which was straightforwardly derived from the μCRL service description in [9]. The main goal was to check the consistency of our service description against the μCRL description. The behavioral model as well as the requirements were easily expressed in PROMELA. Subsequently, a protocol model in PROMELA was built, and verified against the requirements description of the service. The main problem with SPIN was that it cannot deal with the real-time aspects of the protocol, so tricks and assumptions about timer behavior and resynchronization had to be made in the same way as in other verifications of the BRP in an untimed setting [9, 10, 11, 14].

Using different tools with different characteristics turned out to be advantageous, and the tools should not be considered as competing, but as complementary. Describing the protocol in different formalisms gives extra insight, and it certainly helps in distinguishing between problems caused by the protocol, and problems which are modeling problems, specific to a particular formalism.

Altogether, the BRP turned out to be an interesting exercise in protocol verification. Its timing intricacies make it an interesting example, especially for real-time verification tools such as RT-SPIN [15], KRONOS [8], and HYTECH [12].

Acknowledgements: We would like to thank Paul Pettersson, Kim Larsen and Wang Yi for keeping us up to date on the developments of UPPAAL and for suggestions.

References

1. R. Alur and D.L. Dill. A theory of timed automata. *Th. Comp. Sc.*, 126:183–235, 1994.
2. R. Alur, T. Henzinger and E.D. Sontag, editors. *Hybrid Systems III*, LNCS 1066, Springer-Verlag, 1996.
3. J. Bengtsson, D. Griffioen, K. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T.A. Henzinger, editors, *Proc. of CAV'96*, LNCS 1102, pages 244–256. Springer-Verlag, 1996.
4. J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for the automatic verification of real-time systems. In [2], pages 232–243.
5. Z. Brezocnik and T. Kapus, editors. *Proceedings of COST 247 Int. Workshop on Applied Formal Methods in System Design.* University of Maribor Press, 1996.
6. P.R. D'Argenio, J-P. Katoen, T. Ruys, and J. Tretmans. Modeling and Verifying a Bounded Retransmission Protocol. In [5], pages 114–128.
7. P.R. D'Argenio, J-P. Katoen, T. Ruys, and J. Tretmans. The Bounded Retransmission Protocol must be on time!. Report CTIT 97-03, University of Twente, 1997.
8. C. Daws, A. Olivero, S. Tripakis and S. Yovine. The Tool KRONOS. In [2], pages 208–219.
9. J.F. Groote and J. van de Pol. A bounded retransmission protocol for large data packets. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology*, LNCS 1101, pages 536–550. Springer-Verlag, 1996.
10. K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M-C. Glaudel and J. Woodcock, editors, *Proc. of FME'96*, LNCS 1051, pages 662–681. Springer-Verlag, 1996.
11. L. Helmink, M.P.A. Sellink, and F.W. Vaandrager. Proof checking a data link protocol. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, LNCS 806, pages 127–165. Springer-Verlag, 1994.
12. T.H. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HYTECH. In E. Brinksma et. al, editors, *Proc. of TACAS'95*, LNCS 1019, pages 41–71. Springer-Verlag, 1995.
13. G.J. Holzmann. *Design and validation of computer protocols.* Prentice Hall, Englewood Cliffs, 1991.
14. R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In [5], pages 98–114.
15. S. Tripakis and C. Courcoubetis. Extending PROMELA and SPIN for real time. In T. Margaria and B. Steffen, editors, *Proc. of TACAS'96*, LNCS 1055, pages 329–348. Springer-Verlag, 1996.