Modeling and Verifying a Bounded Retransmission Protocol

Pedro R. D'Argenio*, Joost-Pieter Katoen, Theo Ruys, and Jan Tretmans
Faculty of Computing Science
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{dargenio,katoen,ruys,tretmans}@cs.utwente.nl

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 retransmission of a frame, 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. A comparison between these approaches is made and our experiences are reported.*

1 Introduction

The engineering of communication protocols is known to be a complex task. An important activity within the field of protocol engineering is to validate whether a given protocol functions as intended. That is, given a service that the system is proposed to offer to its users the problem is to check whether a certain protocol "conforms to" this service. This activity is known as protocol verification or validation [7]. Formal methods can support this activity to a large extent. By providing formal specifications, S and P say, of the service and protocol, respectively, and formally characterizing the "conforms to" relation (denoted **sat**) protocol verification amounts to proving that P **sat** S.

For moderate examples—like cache consistency [5] and leader election protocols [6]—such proofs can be carried out manually, but for more complex protocols this approach is hardly applicable. For example, the presence of real-time aspects complicates a protocol significantly and for those cases manual verification becomes hard to accomplish. Therefore, a number of algorithms and tools for automatic verification, among which tools for model checking, have been constructed in the last decade. These techniques facilitate the automatic verification of properties, usually stated in some dialect of modal logic, with respect to a protocol specified as a system of communicating finite-state automata. Examples of such tools are SPIN [16] for untimed systems and UPPAAL [3] for timed systems.

SPIN is a verification tool for classical finite-state automata that communicate via channels. It is capable of verifying assertions over data and simple linear-time temporal logic formulae (so-called never claims). SPIN uses the dedicated modeling language PROMELA and uses advanced techniques such as bit-state hashing to increase the verification efficiency.

UPPAAL is capable of verifying safety and bounded liveness properties of real-time systems modeled as networks of timed automata. Timed automata [1] are an extension of the classical finite-state automata with clock variables. UPPAAL uses on-the-fly verification techniques and 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 [3].

This paper concerns a file transfer service and a given bounded retransmission protocol (BRP), a protocol used in one of Philips' products. It addresses the correctness of this protocol with respect to the file transfer service. The BRP is based on the well-known alternating bit protocol but allows for a bounded number of retransmission of a frame, i.e., part of a file, only. So, eventual delivery is not guaranteed and the protocol may abort the file transfer. Timers are involved in order to detect the loss of frames and the abortion of transmission.

We specify the service 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

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

"external behavior" specification of [8]. 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 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 verifications of the BRP using process algebra [8], PVS [10], and I/O-automata [11] that focussed on the data aspects rather than on the timing aspects of the BRP. In order 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.

This paper is further organized as follows. Sections 2 and 4 present an informal description of the service and protocol, respectively. These descriptions are taken from the problem description in the call for contributions to the COST 247 workshop in Maribor, June 1996. Section 3 provides a formal service specification using first-order predicate logic and addresses some simple properties that follow from this logic specification. Section 5 presents the formal specification of the BRP. Section 6 concerns our experiences with using UPPAAL for verifying the correctness of the BRP. Section 7 deals with our verification efforts using SPIN. Finally, Section 8 summarizes our main results and compares the two verification approaches used in this paper.

2 Informal service specification

As any transmission protocol, the BRP behaves like a buffer, i.e., it reads data from one client to be delivered at another one. There are two distinguishing features that make the behavior much more complicated than a simple buffer. Firstly, the input is a *large data packet* (that can be modeled as a list), which is delivered in small chunks. Secondly, there is a *limited amount of time* for each chunk to be delivered, so we cannot guarantee an eventually successful delivery within the given time bound. It is assumed that either an initial part of the list or the whole list is delivered, so the chunks will not be garbled and their order will not be changed. Of course, both the sender and the receiver want an *indication* whether the whole list has been delivered successfully or not.

The input (the list $l = d_1, \ldots, d_n$) is read on the "input" port. Ideally, each d_i is delivered on the "output" port. Each chunk is accompanied by an indication. This indication can be I_FST, I_INC, or I_OK. I_OK is used if d_i is the last element of the list. I_FST is used if d_i is the first element of the list *and more will follow*. All other chunks are accompanied by I_INC.

However, when something goes wrong, a "not OK" indication (I_NOK) is delivered without datum. Note that the receiving client does not need a "not OK" indication before delivery of the first chunk nor after delivery of the last one.

The sending client is informed after transmission of the whole list, or when the protocol gives up. An indication is sent out on the "input" port. 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. 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. After this indication, the protocol is ready to transmit a subsequent list.

This completes the original informal description of the file transfer service. We remark that it is unclear from this service description which indication the sending client receives in case the receiving client does not receive any chunk. Since something went wrong an I_NOK indication is required, but from this indication the sending client may not deduce that the receiving client has the corresponding indication. This is because the receiving client does not receive an I_NOK indication before delivery of the first chunk. So, if the sending client receives an I_NOK either the receiving client received the same or did not receive anything at all.

3 Formal service specification

The file transfer service is considered to have three "service access points". The sending client inputs its file via S_{in} as a list of chunks $\langle d_1, \ldots, d_n \rangle$. We assume that $n > 0$, i.e., the transmission of empty files is not considered. The sending client receives indications i_s via S_{out}, while the receiving client receives pairs (e_j, i_j) of chunks and indications via R_{out}. We assume that all outputs with respect to previous files have been completed when a next file is input via S_{in}. Outputs S_{out} and R_{out} can appear in either order. Figure 1 provides an overview of the file transfer service. The signatures of the input and output are:

$S_{in} : l = \langle d_1, \ldots, d_n \rangle$ 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 $0 \leqslant k \leqslant n$ and
$\quad i_j \in \{ \text{I\_FST}, \text{I\_INC}, \text{I\_OK}, \text{I\_NOK} \}$ for $0 < j \leqslant k$

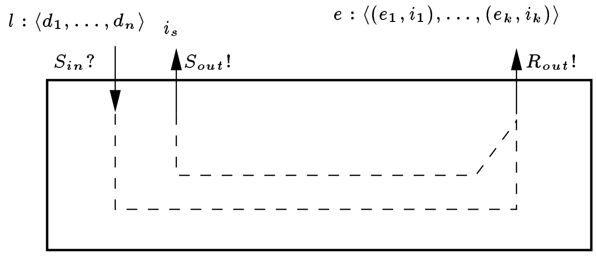

Figure 1: Schematic view of file transfer service.

Below we define a service specification in a logical way, i.e., by stating properties that should be satisfied by the service. These properties are relations between input and output. A protocol conforms to the file transfer service if it satisfies all these properties.

Under the condition that $k > 0$ we have the following requirements. The first requirement states that each correctly received chunk e_j equals d_j, the chunk sent via S_{in}. In case the notification i_j indicates that an error occurred, we do not pose any restriction on the accompanying chunk e_j. Notice that in the original problem description an $\mathsf{I\_NOK}$ must be associated with an empty chunk ε, i.e., $i_j = \mathsf{I\_NOK} \Rightarrow e_j = \varepsilon$; for simplicity, we allow e_j to have an arbitrary content.

$$\textbf{(1.1)} \qquad \forall\, 0 < j \leqslant k : i_j \neq \mathsf{I\_NOK} \Rightarrow e_j = d_j$$

The next three requirements address the constraints concerning the received indications via R_{out}, i.e., i_j. If the number n of chunks in the file exceeds one then we require the first received indication to be $\mathsf{I\_FST}$, indicating that e_1 is the first chunk of the file and more will follow:

$$\textbf{(1.2)} \qquad n > 1 \Rightarrow i_1 = \mathsf{I\_FST}$$

The indications of all chunks, apart from the first and last chunk, equal $\mathsf{I\_INC}$:

$$\textbf{(1.3)} \qquad \forall\, 1 < j < k : i_j = \mathsf{I\_INC}$$

The requirement concerning the last chunk (e_k, i_k) consists of three parts. The first requires the last chunk to be accompanied with either $\mathsf{I\_OK}$ or $\mathsf{I\_NOK}$:

$$\textbf{(1.4.1)} \qquad i_k = \mathsf{I\_OK} \vee i_k = \mathsf{I\_NOK}$$

If the last chunk has indication $\mathsf{I\_OK}$ then k should equal n, indicating that all chunks of the file have been received correctly:

$$\textbf{(1.4.2)} \qquad i_k = \mathsf{I\_OK} \Rightarrow k = n$$

Before delivery of the first chunk the receiving client is not notified in case an error occurs:

$$\textbf{(1.4.3)} \qquad i_k = \mathsf{I\_NOK} \Rightarrow k > 1$$

This concludes the requirements with respect to the indications delivered at the receiving client. The next three requirements specify the relationship between indications given to the sending and receiving client. After an $\mathsf{I\_OK}$ (or $\mathsf{I\_NOK}$) at the sender, the receiver has the corresponding indication.

$$\textbf{(1.5)} \qquad i_s = \mathsf{I\_OK} \Rightarrow i_k = \mathsf{I\_OK}$$

$$\textbf{(1.6)} \qquad i_s = \mathsf{I\_NOK} \Rightarrow i_k = \mathsf{I\_NOK}$$

A "don't know" indication can only appear after delivery of the last-but-one chunk e_{n-1}. This means that the number of indications received by the receiving client must equal n. (Either this last chunk is received correctly or not, and in both cases an indication (+ chunk) is present at R_{out}.)

$$\textbf{(1.7)} \qquad i_s = \mathsf{I\_DK} \Rightarrow k = n$$

This completes the requirements for the case $k > 0$.

In case $k = 0$ the sender should receive an indication $\mathsf{I\_DK}$ 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 $\mathsf{I\_NOK}$ if and only if n exceeds one.

$$\textbf{(2.1)} \qquad i_s = \mathsf{I\_DK} \Leftrightarrow n = 1$$

$$\textbf{(2.2)} \qquad i_s = \mathsf{I\_NOK} \Leftrightarrow n > 1$$

This completes our formal specification of the file transfer service. Remark that there is no requirement at service level concerning the limited amount of time available to deliver a chunk d_j to the receiving client as mentioned in the informal service description. The reason for this is that this is considered as a protocol requirement rather than a requirement of the service.

From the service specification some interesting properties can be deduced. The proofs of these properties are straightforward and are omitted.

LEMMA. The file transfer service satisfies the following properties for $k > 0$ and $0 < j \leqslant k$:

1. $i_1 = \mathsf{I\_FST} \Rightarrow (k > 1 \wedge n > 1)$
2. $(i_j = \mathsf{I\_NOK} \vee i_j = \mathsf{I\_OK}) \Rightarrow j = k$
3. $i_1 \neq \mathsf{I\_NOK}$
4. $1 \leqslant k < n \Rightarrow i_k = \mathsf{I\_NOK}$
5. $k = 1 \Rightarrow n = 1$.

4 Informal protocol specification

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

Sender S reads a list to be transmitted and sets the retry counter to 0. Then it starts sending the elements of the list 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 list. The second bit indicates whether the datum is the last item of the list. 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 list) the sending client is informed of correct transmission, or the next element of the list 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 list 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 list 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 list, then a fresh frame will be the first of the subsequent list and a repeated frame will still be the last of the old list.

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 list has been given up. The receiving client is informed, provided the last element of the list has not just been delivered. Note that if transmission of the next list 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 list has been interrupted by the sender. So its delay must exceed MAX times the delay of T_1. Assume that the sender does not start reading and transmitting the next list 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. It is important to note that two significant assumptions are being 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 $\mathsf{T1}$ 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

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—which apparently is the case in the protocol at hand. We, therefore, consider **(A2)** as an unrealistic assumption and believe that this assumption originates from the modeling choices made by others that studied the BRP in an untimed setting [8, 10, 11]. 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!).

5 Formal protocol specification

The BRP consists of a sender S and a receiver R communicating through channels K and L (see Figure 2).

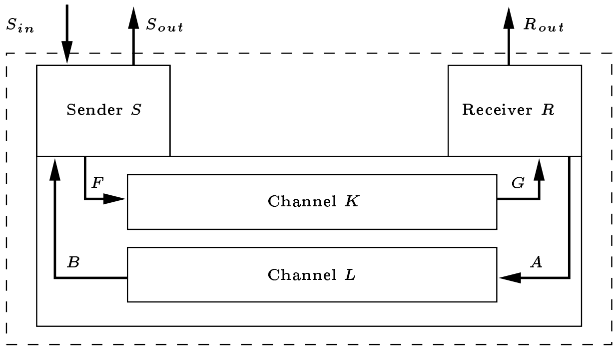

Figure 2: Schematic view of the BRP.

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

$$F, G : (b, b', ab, d_i) \text{ with } b, b' \in \{\,\text{true}, \text{false}\,\}, \quad ab \in \{\,0, 1\,\} \text{ and } 0 < i \leqslant n$$

$$A, B : ack$$

Our starting-point for modeling and verifying the BRP is a specification of the BRP in terms of a network of timed automata. The model of timed automata [1] is an extension of the classical finite-state automaton model with *clock variables*. 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. 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. *Communications* can thus be considered as (possibly delayed) distributed assignments. That is, for processes P and Q connected via C, variables x_i and expressions E_i of corresponding type ($0 < i \leqslant k$), the execution of $C?(x_1, \ldots, x_k)$ in P and $C!(E_1, \ldots, E_k)$ in Q establishes the multiple assignment $x_1, \ldots, x_k := E_1, \ldots, E_k$ in P.

Transitions consist of an (optional) guard and zero or more actions. Depending on 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 (and time passes implicitly). 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 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 circles, initial states as double-lined circles. Transitions are denoted by directed, labeled arrows. A list of guards denotes the conjunction of its elements.

The channels K and L are modeled as depicted in Figure 3. In the *start* state K is waiting for a message to be received via F. If such a message arrives, clock u is reset and the process can either lose the message (upper transition) or may pass on the just received message via G after a delay larger than 0 and at most TD time units (lower transition). Notice that both K and L have a capacity of one message only. This simplification is justified by the fact that the BRP is a variant of the 'stop-and-wait' protocol where a next message is sent only after the receipt of an acknowledgment.

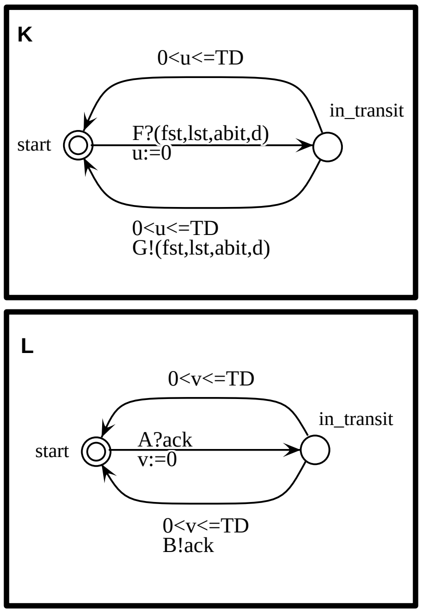

Figure 3: Timed automata for channels K and L.

The sender S (see Figure 4) has three system variables: $ab \in \{\,0, 1\,\}$ indicating the alternating bit that accompanies the next chunk to be send, 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 transmit d_i. Clock variable x is used to model timer T_1 and to make certain transitions urgent (see below).

In the *idle* state S waits for a new file to be received via S_{in}. On receipt of a new file it sets i to one, and resets clock x. Going from state *next_frame* to *wait_ack* chunk d_i is transmitted with the corresponding information and rc is reset. In state *wait_ack* there are several possibilities: in case the maximum number of transmissions has been reached (i.e., $rc = \mathsf{MAX}$), S moves to an *error* state while resetting x and emitting an $\mathsf{I\_DK}$ or $\mathsf{I\_NOK}$ indication to the sending client (via S_{out}) depending on whether d_i is the last chunk or not; if $rc < \mathsf{MAX}$, either an *ack* is received (via B) within time (i.e., $x < \mathsf{T1}$) and S moves to the *success* state while alternating ab, or timer x expires (i.e., $x = \mathsf{T1}$) and a retransmission

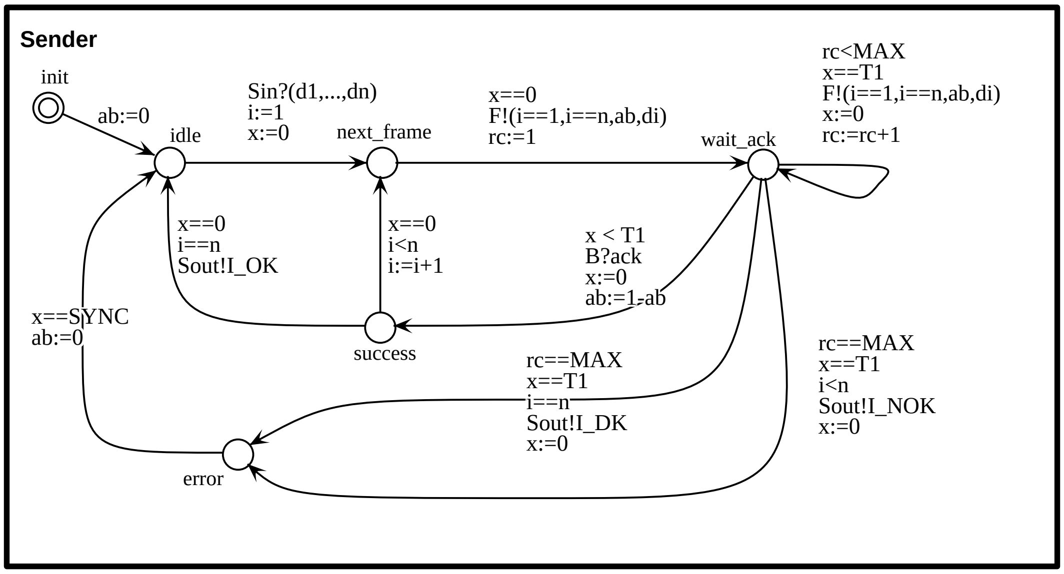

Figure 4: Timed automaton for sender S.

is initiated (while incrementing *rc*, but keeping the same alternating bit). If the last chunk has been acknowledged, S moves from state *success* to state *idle* indicating the successful transmission of the file to the sending client by I_OK. If another chunk has been acknowledged, i is incremented and x reset while moving from state *success* to state *next_frame* where the process of transmitting the next chunk is initiated.

Two remarks are in order. First, notice that transitions leaving state s, say, containing a guard $x = 0$ are executed immediately, 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 protocol's correctness. 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.

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.

In state *new_file*, R is waiting for the first chunk of a new file to arrive. Immediately after the receipt of such chunk *exp_ab* is set to the just received alternating bit and R enters the state *frame_received*. If the expected alternating bit agrees with the just received alternating bit (which, due to the former assignment to *exp_ab* is always the case for the first chunk) then an *ack* is sent via A, *exp_ab* is toggled, an appropriate indication is sent to the receiving client, and clock z is reset. R is now in state *idle* and waits for the next frame to arrive. If such frame arrives in time (i.e., $z <$ TR) then it moves to the state *frame_received* and the above described procedure is repeated; if timer z expires (i.e., $z =$ TR) then in case R did not just receive the last chunk of a file an indication I_NOK (accompanied with an 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)**. For example, if we allowed an arbitrary delay in state *frame_received* then the sender S could generate a timeout (since it takes too long for an acknowledgment to arrive at S) while an acknowledgment generated by R is possibly still to come.

6 Uppaal

Uppaal [3, 4] is a tool suite for symbolic and compositional model checking of real-time systems. It has been developed in collaboration between BRICS at Ålborg University and the Department of Computing Systems at Uppsala University. Systems in Uppaal are described as networks of timed automata [1], i.e., the different parts of the system are represented as timed automata and they are (implicitly) combined in parallel where handshake synchronization is assumed, as described in the previous section. The use of data is restricted to clocks and integers (rather than sys-

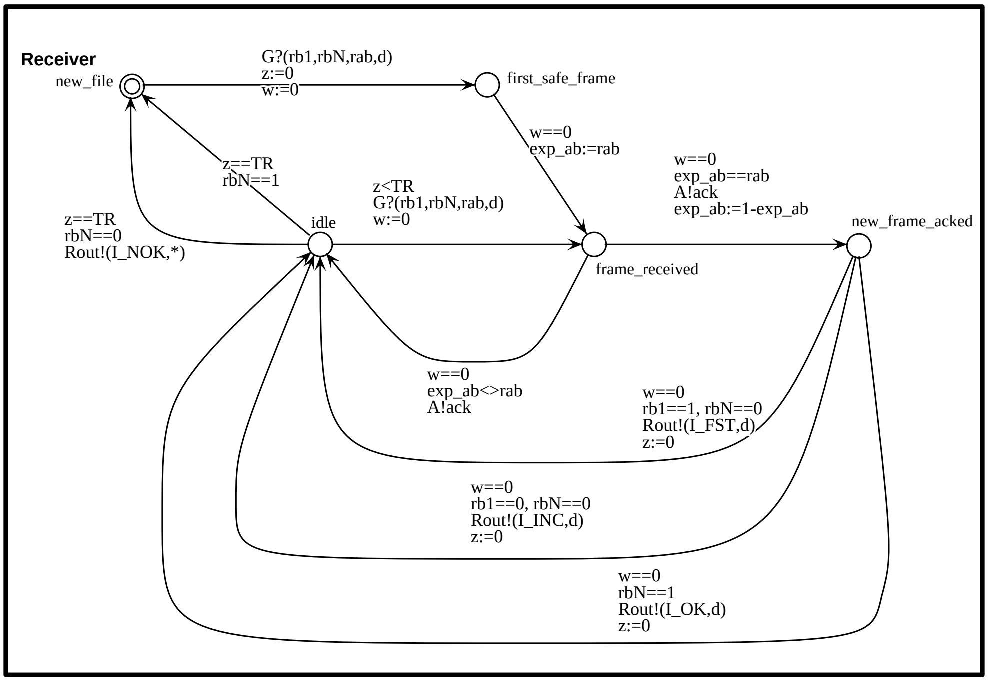

Figure 5: Timed automaton for receiver R.

tem variables of arbitrary type) and no value passing is allowed at synchronization.

Graphical description of timed automata is possible by using Autograph[1]. The output of Autograph is compiled into textual format, which can be checked for syntactical correctness by `checkta`. The textual representation is one of the inputs to the verifier `verifyta`. `verifyta` is the program that determines the satisfaction of a given property. If a property is not satisfied, `verifyta` is able to report a diagnostic trace that indicates how it can be violated. An overview of UPPAAL is depicted in Figure 6.

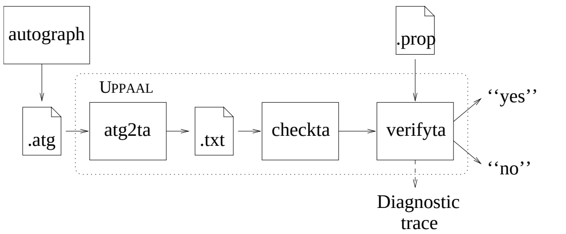

Figure 6: Overview of UPPAAL.

In the current version[2], UPPAAL is able to check only simple reachability properties. Properties are terms in the language defined by the following syntax:

$$\phi ::= \forall\Box\beta \mid \exists\Diamond\beta \qquad \beta ::= a \mid \beta \wedge \beta \mid \neg\beta$$

[1]Autograph is developed at CMA Ecole des Mines, Sophia-Antipolis, France.

[2]We were using UPPAAL version 0.99.

where a is either a state of a component, i.e., one of the states of any of the timed automata, or a clock constraint, which has the form $x \sim n$ where x is a clock, n an integer, and $\sim \in \{\leqslant, \geqslant, =\}$.

6.1 Protocol specification

The specification of the protocol in UPPAAL is a rather straightforward adaptation of the specification given in Section 5; see Figures 7 through 9. The following considerations have been taken into account.

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 $\sim \in \{\leqslant, \geqslant, =\}$ a non-strict comparison operator. Due to the absence of strict comparison operators we had to modify some guards. For example, the constraints on clock x in sender S had to be weakened such that in state *wait_ack* a non-deterministic choice appears in case an acknowledgment arrives via B and x equals T1.

The number of states, clocks and system variables, as well as the value of the constants directly determine the size of the state space (also known as the region space) of a real-time system. Although UPPAAL uses compositional techniques to attack the problem of region space explosion, it is sensible to such a problem. 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. We also deal with a fixed file length and

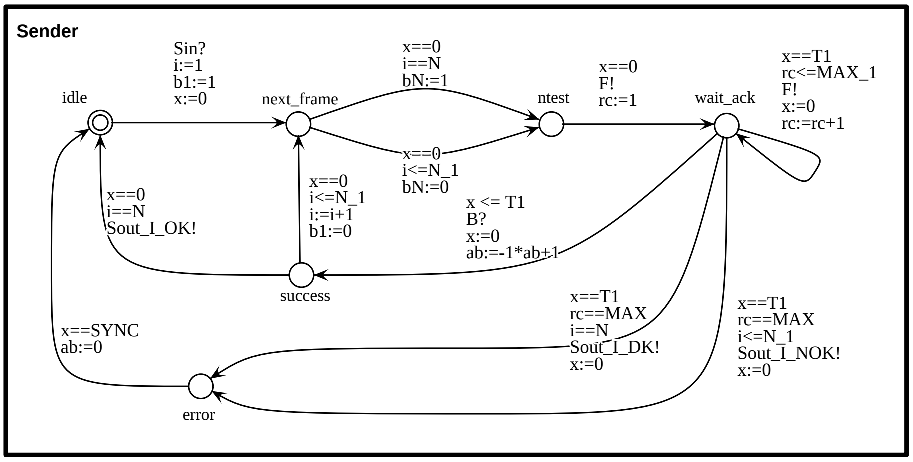

Figure 7: Sender S in UPPAAL.

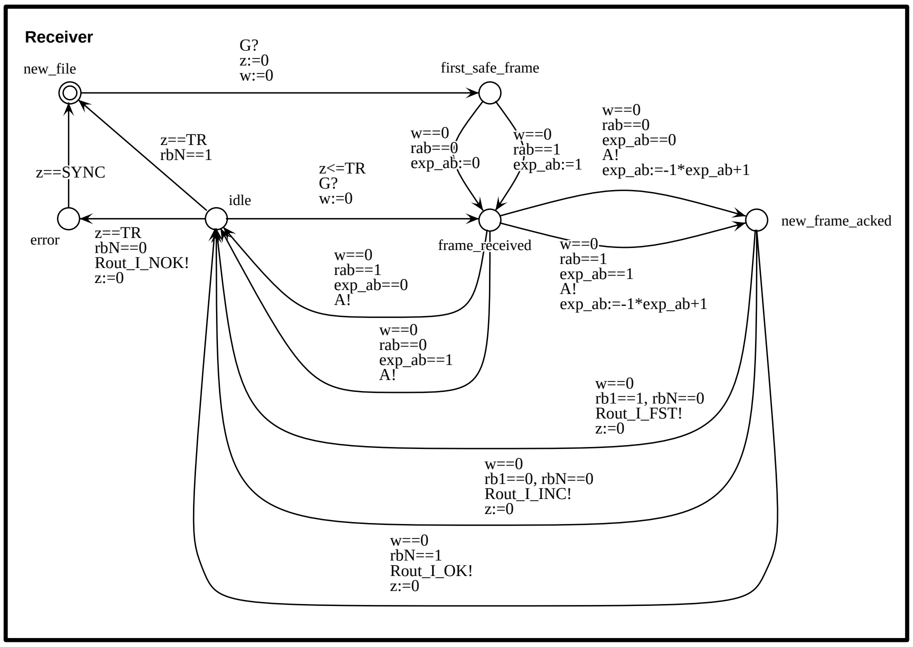

Figure 8: Receiver R in UPPAAL.

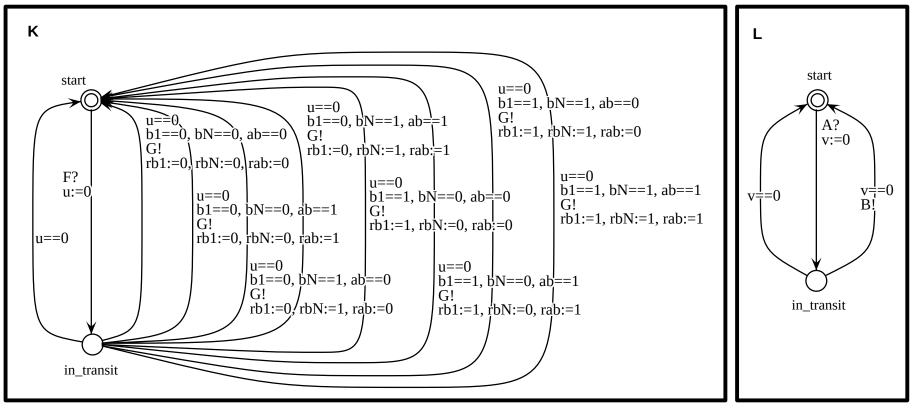

Figure 9: Channels K and L in UPPAAL.

moreover, we fixed the transmission delay on 0.

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 that can be received via G; this resulted in 8 transitions, see Figure 9.

Nomenclature is similar to Section 5 except for minor changes (e.g., *Sout_I_OK?* instead of *Sout?I_OK*). Constants N_1 and $\mathsf{MAX\_1}$ denote $N-1$ and $\mathsf{MAX}-1$, respectively.

6.2 Protocol properties

In Section 4 we considered it unrealistic to assume that in case of a failure sender S starts the transmission of a new file only after R has properly reacted to the failure, cf. assumption **(A2)**. Instead, we stated that this should be a property of our design rather than an assumption. This property can be expressed in Timed-CTL [2] as

$$\forall\ (S.error\ \mathcal{U}_{\leqslant \mathsf{SYNC}}\ (R.error\ \vee\ R.new\_file)) \quad (1)$$

That is, if S is in state *error*, eventually, within SYNC time units, either R moves to an *error* state or it is (already) able to receive a *new_file*. Unfortunately, the property language of UPPAAL does not support this type of formula. Therefore, we checked the following property:

$$\forall\Box\ (S.error\ \wedge\ x = \mathsf{SYNC})\ \Rightarrow\ (R.error\ \vee\ R.new\_file) \quad (2)$$

The difference between properties (1) and (2) is that (1) requires that *S.error* is true until *R.error* or *R.new_file* becomes true, while (2) does not take into account what happens when time passes, but considers only the instant for which $x = \mathsf{SYNC}$. Provided that S is in state *error* while clock x evolves from 0 to SYNC—which is obviously the case—(2) implies (1). Property (2) is satisfied under the condition that $\mathsf{SYNC} > \mathsf{TR}$. This means that **(A2)** only holds if this condition on the values SYNC and TR is respected; it shows *the importance of timing aspects for the correctness of the BRP.*

In order to reduce the complexity of the region space of our protocol specification the following property turned out to be useful. Property

$$\forall\Box\ \neg\,(K.in\_transit\ \wedge\ L.in\_transit) \quad (3)$$

states that it is impossible to have both a message and an acknowledgment in transit at the same time. This property could be proven using UPPAAL. Recall that the number of states and clocks in our specification determines the size of the region space. It is known that the region space depends exponentially on the number of clocks [1]. Property (3) allows us to reduce the process $K \,||\, L$, where $||$ denotes independent parallelism, into process *Lines*, cf. Figure 10. Compared to the original specification of $K \,||\, L$ this reduces one clock and one state.

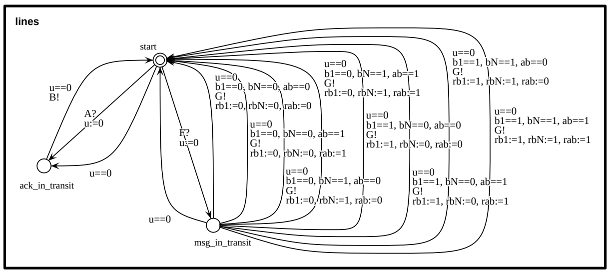

Figure 10: Alternative channels.

6.3 Protocol verification

In order to verify that the protocol satisfies the service specification of Section 3, 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 11.

When trying to verify the correctness of the BRP using UPPAAL we encounter the following problems. Firstly, the properties constituting the service specification of Section 3 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, properties in UPPAAL 0.99 may only contain clocks and states, but no system variables. Thirdly, in order to cope with the region space problem we had to remove the data from our specification. Therefore, we were unable to check properties concerning the transmitted data, like property **(1.1)**.

The properties that we proved are enumerated in Table 1. The rightmost column indicates which properties were satisfied ($\surd$) and which ones we were unable to prove ($\times$), basically due to insufficient memory. Properties 1. and 2. are weakened versions of properties **(1.5)** and **(1.6)**, respectively. Property 3.

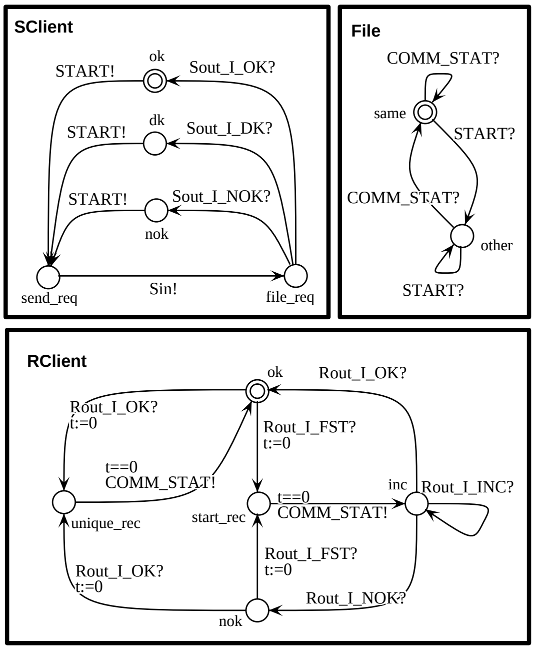

Figure 11: Auxiliary automata.

is related to **(2.1)** and **(2.2)**. Properties 4. and 5. relate the sender S and the sending client, while 6. and 7. relate the receiver R and the receiving client.

The proved properties were proven with the following settings:

$$n = 3 \qquad \mathsf{T1} = 1 \qquad \mathsf{MAX} = 2$$
$$\mathsf{TR} = 2 \qquad \mathsf{SYNC} = 3$$

Table 1: Properties in UPPAAL.

1.	$\forall\Box\ \mathit{File.same} \Rightarrow \neg(\mathit{SC.ok} \wedge \mathit{RC.nok})$	√
2.	$\forall\Box\ \mathit{File.same} \Rightarrow \neg(\mathit{SC.nok} \wedge \mathit{RC.ok})$	×
3.	$\forall\Box\ \neg(\mathit{File.other} \wedge \mathit{SC.ok})$	×
4.	$\forall\Box\ \mathit{SC.idle} \Rightarrow (\mathit{S.idle} \vee \mathit{S.error})$	√
5.	$\forall\Box\ (\mathit{S.idle} \vee \mathit{S.error}) \Rightarrow \neg\,\mathit{SC.file\_req}$	√
6.	$\forall\Box\ \mathit{R.new\_file} \Rightarrow \mathit{RC.idle}$	√
7.	$\forall\Box\ \mathit{R.error} \Rightarrow \mathit{RC.nok}$	×

6.4 Results and problems

Apart from detecting some errors in previous specifications we were able to prove with UPPAAL that assumption **(A2)** is fulfilled only if $\mathsf{SYNC} > \mathsf{TR}$. (Remark that SYNC is a constant in the sender S, while TR is a constant used in receiver R). This shows that the correctness of the BRP depends on the time values used within the protocol.

The assumption of 0-delay in the transmission channels K and L reduces the size of the region space significantly (in fact, we were unable to prove properties when positive delays were introduced). As a result of these zero delays, however, the receiver R may detect a transmission failure, while sender S has not aborted (and probably will not abort) the transmission. This problem is resolved by introducing the state *error* in receiver R and having a delay of SYNC time units when moving from state *error* to state *new_file*.

A second problem that we faced is the expressivity of the property language. There are some basic properties of timed systems, such as time-deadlock freeness, that we would have liked to prove, but UPPAAL 0.99 is not able to express. Our major problem, though, was the lack of an appropriate computer system to verify this protocol. Our machine (SunOS 5.5, 96 MB of main memory, 213 MB of virtual memory) easily runs out of memory.

7 SPIN

PROMELA [15, 16] is a language designed for the modeling and verification of communicating finite state automata. The finite state automata, called *processes*, act independently of each other and interact through the exchange of messages over *channels*. SPIN is an automated validation system for systems modeled in PROMELA. 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. Very large validation runs, for which an exhaustive validation is not feasible, can be validated in SPIN with a *bit-state hashing* technique [13, 14, 15] at the expense of completeness.

Protocols can only be validated with respect to specific correctness requirements. PROMELA includes several constructs for specifying these correctness requirements:

- *assertions:* boolean conditions attached to a state that must be fulfilled when a process reaches that state;
- *validation labels*, namely:
  - *end* states labels, which mark states that are considered to be valid final states of the system;
  - *progress* state labels, which mark states that must be executed for the protocol to make progress;
  - *acceptance* state labels, which mark states that may not be part of a sequence of states which can be repeated infinitely often; and

- *temporal claims*, which define temporal orderings of properties of states.

The validation labels and temporal claims are the more powerful methods for expressing correctness requirements in PROMELA. For the verification of the BRP, however, the use of assertions turned out to be sufficient.

PROMELA is more data-oriented than the timed automata representation which is used in UPPAAL. UPPAAL, for instance, 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 more extensively verified in PROMELA than in UPPAAL. On the other hand, PROMELA lacks one important ingredient for the realistic modeling of the BRP: the notion of time.

7.1 Organization

To get confidence in our logical service requirements from Section 3, we have written a PROMELA specification for the file transfer service.

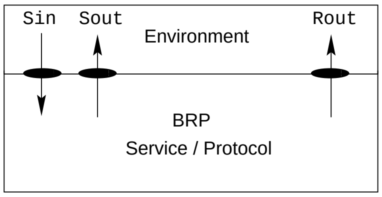

Figure 12: Service Access Points of the BRP.

Figure 12 shows an overview of the communication between the BRP and its environment. Either the PROMELA description of the file transfer service or the PROMELA protocol description of the BRP can be "plugged" in. The *Environment* process feeds the BRP with the file to be transferred at S_{in} (i.e., the list of chunks) and receives the indications at S_{out} and R_{out}. Furthermore, after receiving all indications of the transmission of a single file, the *Environment* process checks the validity of the indications.

The logical service requirements are just constraints on the (list of) chunks offered at S_{in} and the indications received at S_{out} and R_{out}. In our validation model, each requirement has been translated into a sequence of PROMELA statements involving assertions. For example, requirement

$$\textbf{(1.1)} \qquad \forall\, 0 < j \leqslant k : i_j \neq \mathsf{I\_NOK} \;\Rightarrow\; e_j = d_j$$

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
```

7.2 Service model

The service description in PROMELA is obtained by a straightforward translation of the "external behavior" specification in [8] in the process algebra μCRL [9]. Below we have included the PROMELA `proctype` definition of the *Service* process.

```
proctype Service (chan Sin, Sout, Rout)
{
  byte j,k,v ;

  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
}
```

7.3 Protocol model

The PROMELA model for the BRP is based on the formal protocol description as defined in Section 5. However, PROMELAversion 2.7.7 does not include a notion of time, so we had to use "tricks", analogous to [8], to model timing and synchronization of the BRP. These "tricks", in fact, correspond to the assumptions **(A1)** and **(A2)**.

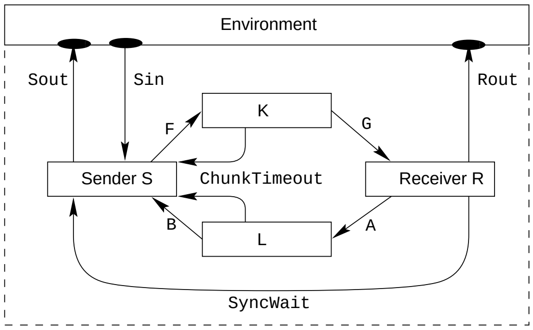

Figure 13: Structure of the BRP in PROMELA.

Figure 13 shows the structure of the BRP in terms of PROMELA processes and interconnecting channels. Like [8] we have modeled the timers T_1 and T_2 that are used in the sender S and receiver R, respectively, 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 body for the process that models channel 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 interrupted by sender S. It is stated in the description of the BRP that "the sender does not start reading and transmitting the next file before the receiver has properly reacted to the failure", cf. assumption **(A2)**. In UPPAAL, we implemented this assumption using two timers: one at the sender's side and one at the receiver's side. In case of a failure, 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 rendezvous synchronization. Note that this extra communication channel between S and R mimics the timer process T_2 used in [8].

7.4 Validation

In Section 7.2 we already stated that after receiving all indications of the transmission of a single file, the *Environment* process checks the results of the transmission against our logical service constraints.

For the verification of the BRP we used the following parameters:

max number of retries	2
max number of frames	3
number of different data items	3

For the validation of the BRP with PROMELA we used the same machine on which we carried out our UPPAAL verification. This machine proved to be adequate for a complete reachability analysis of the file transfer service; SPIN managed to explore the complete state space and reported no errors. Not surprisingly, the state space of the protocol model was much larger; we had to resort to SPIN's bit-state hashing features: SPIN reassured us that it had covered 98% of the state space, and it did not find any assertion violations.

8 Concluding remarks

In this paper we reported on the analysis and verification of a bounded retransmission protocol (BRP). As a starting point we used natural-language descriptions of the service and the protocol as given in Sections 2 and 4, respectively. The tools used for the verification were the protocol validation tool SPIN [15, 16], and the real-time verification tool UPPAAL [3, 4].

We started the modeling activity by making formalizations of both the service and the protocol. The service is a system in which files are transported from a sending entity to a receiving entity. Almost all its properties can be described by simple requirements relating the input at the sending side to the output at the receiving side. This was done in Section 3. The protocol was formalized in Section 5 as a collection of communicating timed automata. Whereas the service is time-independent, i.e., no reference needs to be made to timers or time-outs in its description, real-time aspects are of importance in the protocol description.

The tool UPPAAL was used to check the timed automaton model of the protocol against the requirements description of the service. To do this, the protocol automaton had to be adapted to cope with the restricted modeling language of UPPAAL (restricted form of conditions, no variables, and no value passing). Moreover, simplifications were necessary in order to prevent state-space explosion. Also the service requirements had to be adapted in order to cope with the restricted property language of UPPAAL. Nevertheless, with some additional assumptions (e.g., 0-delay lines), we were able to check some properties. Most notably, we could show that two assumptions in the protocol description ((**A1**): premature time-outs are not possible; and (**A2**): sender and receiver resynchronize after an abort) are easily invalidated by choosing wrong time-out values. Hence, these assumptions are not properties which can be assumed beforehand, but they are desirable properties which should be validated and verified by choosing the right time-out values in the protocol. With UPPAAL it was easily shown that the correctness of the BRP critically depends on the values of these time-outs.

The main problems with UPPAAL were the restricted expressiveness of its modeling and property languages, and the difficulties we had with its memory usage (running on SUN SPARC with 96 MB). Despite some simplifications in the model, such as assuming no delay in the lines (0-delay lines, Section 6), there were many properties that we could not check due to lack of memory.

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 description of the service in [8]. 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 the μCRL model of the protocol in [8]. Whereas with the service description a full state-space exploration was feasible, the protocol only allowed partial state-space exploration using the bit-state hashing technique of SPIN.

When comparing the verifications with the two different tools it can be noted that it was successful in the sense that with both tools we did find errors in our models. 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. Language-oriented solutions are less likely to be confused with pure protocol aspects (cf. the synchronization assumption (**A2**) in the untimed case, and the 0-line-delay assumption in the timed case). In this respect, it might be worth noting that the original natural-language description is not free from making the impression of being biased already towards the (untimed) model in μCRL of [8].

The benefit of using different modeling formalisms and tools is clearly shown by comparing the UPPAAL verification with the previously published verifications of BRP, which are all based on untimed models [8, 10, 11]. UPPAAL demonstrates that the BRP is a very time-dependent protocol for which time-out values are critical for its correctness. Untimed analysis needs heavy assumptions ((**A1**) and (**A2**)), and can therefore only establish partial correctness. For a time-dependent protocol like the BRP, timing analysis is necessary to establish complete correctness.

For building the verification models rather some effort was spent on dealing with specific language issues and tool inconveniences, which had nothing to do with the conceptual problems of the protocol. This aggravates the the danger of choosing language-oriented solutions in protocol modeling instead of concentrating on the bare protocol problems. In particular in UPPAAL, tricks were necessary to avoid the infamous 'out-of-memory' result. Sometimes these tricks required rather detailed knowledge about the internal functioning of the tool such as the construction of region spaces.

Memory consumption is a well-known bottleneck for the use of this kind of verification tools on realistic protocols. On our SUN SPARC with 96 MB of memory we already had large problems with verifying this relatively simple BRP. Use of swap space cannot compensate for lack of main memory since swapping decreases performance to unacceptable levels. As expected, UPPAAL needs more memory than SPIN. Moreover, SPIN can deal more easily with larger state-spaces due to its technique of bit-state hashing, however, this occurs at the expense of losing completeness in the verification.

With respect to the BRP itself it can be noted that its strong dependence on time-out values is usually not considered a desirable property for well-designed protocols. For example, the correctness of the alternating bit protocol, although usually timers are used in its description, does not depend on any of its time-out values. In the BRP, on the other hand, the correctness critically depends on the chosen time-out values, and the correct time-out values depend on the delay in the communication lines and the execution speed of the processors executing the protocol. So the protocol can become incorrect by taking a slower communication line, or by increasing the load on the protocol processors.

Altogether, the BRP turned out to be an interesting exercise in protocol verification, which is more complex than the infamous alternating bit protocol, but which is still manageable. Although the BRP looks a bit simple at first sight (which made us underestimate the effort necessary to model it), its timing intricacies make it an interesting example, especially for real-time verification tools. Such tools, like UPPAAL, are currently being developed by different research groups, and further experiments with these various tools could be interesting, e.g., with RT-SPIN (real-time verification on top of SPIN [19]), with KRONOS (real-time verification based on timed automata and Timed-CTL [18, 17]), and with HYTECH (verification of linear hybrid systems [12]).

References

[1] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.

[2] R. Alur, C. Courcoubetis, and D.L. Dill. Model checking in dense real time. *Information and Computation*, 104:2–34, 1993.

[3] 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 *Proceedings of the 4th DIMACS Workshop on the Verification and Control of Hybrid Systems*, *Lecture Notes in Computer Science*. Springer-Verlag, 1995.

[4] J. Bengtsson, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL in 1995. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 431–434. Springer-Verlag, 1996.

[5] E. Brinksma. Cache consistency by design. *Distributed Computing*, 1996 (to appear; also available as Technical Report 95-17, University of Twente, 1995)

[6] J. Brunekreef, J.-P. Katoen, R. Koymans, and S. Mauw. Design and analysis of dynamic leader election protocols in broadcast networks. *Distributed Computing*, 9:157–171, 1996.

[7] M.G. Gouda. Protocol verification made simple: a tutorial. *Computer Networks and ISDN Systems*, 25:969–980, 1993.

[8] 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*, volume 1101 of *Lecture Notes in Computer Science*, pages 536–550. Springer-Verlag, 1996.

[9] J.F. Groote and A. Ponse. The syntax and semantics of μCRL Report CS-R9076, Centre of Mathematics and Computer Science, 1990.

[10] K. Havelund and N. Shankar. Experiments in theorem proving and model checking for protocol verification. In M.-C. Glaudel and J. Woodcock, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, 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*, volume 806 of *Lecture Notes in Computer Science*, 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, W.R. Cleaveland, K.G. Larsen, T. Margaria, and B. Steffen, editors, *First Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, volume 1019 of *Lecture Notes in Computer Science*, pages 41–71. Springer-Verlag, 1995.

[13] G.J. Holzmann. On limits and possibilities of automated protocol analysis. In H. Rudin and C.H. West, editors, *Protocol Specification, Verification, and Testing VII*, pages 339–344. North-Holland, 1987.

[14] G.J. Holzmann. An improved protocol reachability analysis technique. *Software Practice and Experience*, 18(2): 137–161, 1988.

[15] G.J. Holzmann. *Design and validation of computer protocols*. Prentice Hall, Englewood Cliffs, 1991.

[16] G.J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, 25:981–1017, 1993.

[17] X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. An approach to the description and analysis of hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 149–178. Springer-Verlag, 1993.

[18] A. Olivero and S. Yovine. *Kronos: A Tool for Verifying Real-Time Systems — User's Guide and Reference Manual*. VERIMAG, Montbonnot Saint Martin, France, draft 0.0 edition, 1993.

[19] S. Tripakis and C. Courcoubetis. Extending PROMELA and SPIN for real time. In T. Margaria and B. Steffen, editors, *Second Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 329–348. Springer-Verlag, 1996.