The Bounded Retransmission Protocol must be on time! (Full version)

Pedro R. D'Argenio[1]*, Joost-Pieter Katoen[2], Theo C. Ruys[1], and G. Jan Tretmans[1]

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

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

August, 1997

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.

1991 Mathematics Subject Classification: 68Q60, 68Q22.
1991 CR Categories: C.2.2, D.2.4, F.3.1.
Keywords: verification, model checking, real-time, bounded retransmission protocol, UPPAAL, SPIN.

Note: An extended abstract of this report has been published in the *Proceedings of the third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'97*, Enschede, The Netherlands, April 1997 [8].

Additional material can be found on *http://wwwtios.cs.utwente.nl/~dargenio/brp/*

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

Contents

1 Introduction

An important activity within the field of protocol engineering is to validate whether a protocol functions as intended. Given a service that the system is supposed 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 [10]. Formal methods can support this activity to a large extent. By providing formal specifications S and P of the service and protocol, respectively, and formally characterizing the "conforms to" relation (denoted **sat**), protocol verification amounts to proving that P **sat** S.

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 [19] 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 [12]. 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 untimed BRP verifications of [12, 14, 15, 23] 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 [7]. In particular, we could obtain tight constraints on the timing aspects of the BRP under which it conforms to the FTS.

2 The file transfer service

2.1 Informal description

As for many transmission protocols, the service delivered by the BRP behaves like a buffer, i.e., it reads data from one client to be delivered at another one. There are two features that make the behavior much more complicated than a simple buffer. Firstly, the input is a *large file* (that can be modeled as a list), which is delivered in small

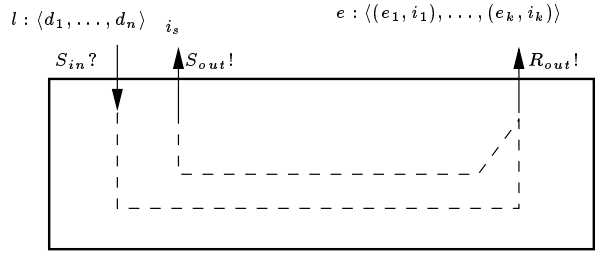
chunks. Secondly, there is a *limited amount of time* for each chunk to be delivered, so we cannot guarantee an eventually successful delivery within the given time bound. It is assumed that either an initial part of the file or the whole file is delivered, so the chunks will not be garbled and their order will not be changed. Both the sender and the receiver obtain an *indication* whether the whole file 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, I_OK, or I_NOK. I_OK is used if d_i is the last element of the file. I_FST is used if d_i is the first element of the file *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 file, 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 acknowledgement 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 file.

This completes the informal FTS description which we adopted from [12]. Remark that it is unclear from this 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.

2.2 Formal specification



Schematic view of the FTS.

Signatures of the input and output:

$S_{in} : l = \langle d_1, \ldots, d_n \rangle$ for $n > 0$
$S_{out} : i_s \in \{$ I_OK, I_NOK, I_DK $\}$
$R_{out} : \langle (e_1, i_1), \ldots, (e_k, i_k) \rangle$ for $0 \leqslant k \leqslant n$,
$i_j \in \{$ I_FST, I_INC, I_OK, I_NOK $\}$
for $0 < j \leqslant k$

The FTS is considered to have two "service access points": one at the sender side and the other at the receiver side. 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}.

In Table 1 we specify the FTS in a logical way, i.e., by stating properties that should be satisfied by the service. These properties define relations between input and output. Note that 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 conforms to the FTS if it satisfies all listed properties.

Table 1: Formal specification of the FTS.

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

For $k > 0$ we have the following requirements. **(1.1)** 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, no restriction is imposed on the accompanying chunk e_j. **(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 the file exceeds one then **(1.2)** requires i_1 to be I_FST, indicating that e_1 is the first chunk of the file and more will follow. **(1.3)** requires that the indications of all chunks, apart from the first and last chunk, equal I_INC. The requirement concerning the last chunk (e_k, i_k) consists of three parts. **(1.4.1)** requires

e_k to be accompanied with either I_OK or I_NOK. **(1.4.2)** states that if $i_k = \mathsf{I\_OK}$ then k should equal n, indicating that all chunks of the file have been received correctly. **(1.4.3)** requires that the receiving client is not notified in case an error occurs before delivery of the first chunk. **(1.5)** through **(1.7)** specify the relationship between indications given to the sending and receiving client. **(1.5)** and **(1.6)** state when the sender and receiver have corresponding indications. **(1.7)** requires a "don't know" indication to only appear after delivery of the last-but-one chunk d_{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}.)

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. This gives rise to **(2.1)** and **(2.2)**.

Remark that there is no requirement concerning the limited amount of time available to deliver a chunk 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 service requirement.

From the service specification some interesting properties can be deduced. They provide insight in the behavior of the service and increase confidence in the correctness of the service specification.

Lemma. *The FTS specification satisfies the following properties for* $k > 0$*:*

1. $i_1 = \mathsf{I\_FST} \;\Rightarrow\; (k > 1 \;\wedge\; n > 1)$
2. *for all* j *such that* $0 < j \leqslant k$, $(i_j = \mathsf{I\_NOK} \;\vee\; i_j = \mathsf{I\_OK}) \;\Rightarrow\; j = k$
3. $i_1 \neq \mathsf{I\_NOK}$
4. $1 < k < n \;\Rightarrow\; i_k = \mathsf{I\_NOK}$
5. $k = 1 \Rightarrow n = 1$

For the proof of this lemma see Appendix A.

3 The bounded retransmission protocol

3.1 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 acknowledgement from the receiver, or for a timeout. In case an acknowledgement 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 acknowledgement 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 acknowledgement 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.[1] 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 [12]). 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 us suppose that the maximum delay in the channel K (and L) is TD and that timer T_1 expires if an acknowledgement 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

[1] Later on we will show that this lower bound is not sufficient.

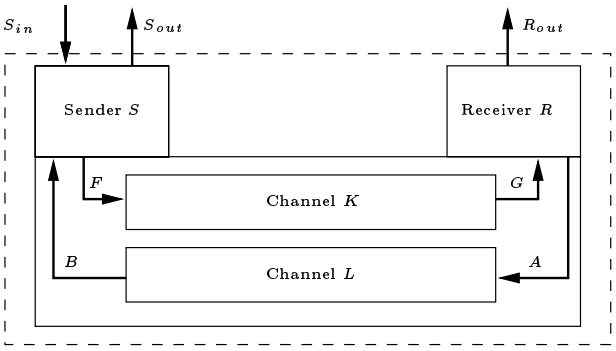
δ denotes the processing time in the receiver R. **(A1)** thus requires knowledge about the processing speed of the receiver and the delay in the line.

(A2) In case of abortion, 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. 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!).

3.2 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. Acknowledgements ack are sent via A and B using L.



Schematic view of the BRP.

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

$F, G : (b, b', ab, d_i)$
with $ab \in \{0, 1\}$,
$b, b' \in \{\text{true}, \text{false}\}$
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. 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. *Communications* can thus be considered as 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 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[2]. Evaluation of a guard, taking a transition and executing its associated actions constitute 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 labeled circles and 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). 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 retransmissions has been reached (i.e., $rc = \mathsf{MAX}$), S moves to an *error* state while resetting x and emitting an I_DK or 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 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

[2]By "halting system" we mean *time deadlock* (see [24, 11]). Time deadlock is a theoretical phenomenon which originates when one of the system components must synchronize with a second component before meeting some deadline, but the second component is not yet ready. As the reader can see, this is a contradictory situation and it is considered to be catastrophic. The time deadlock phenomenon should not be present in correct systems.

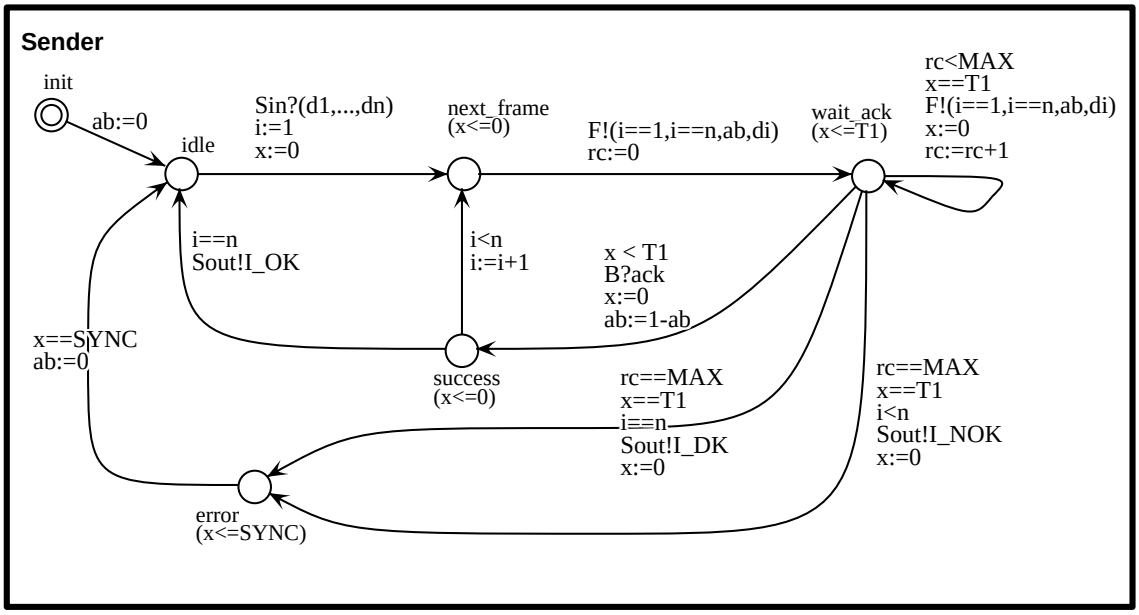


Figure 1: Timed automaton for sender S.

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, 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, we remark that 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 receiver is depicted in Figure 2. 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 appropriate indication is sent to the receiving client, an *ack* is sent via A, *exp_ab* is toggled, 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

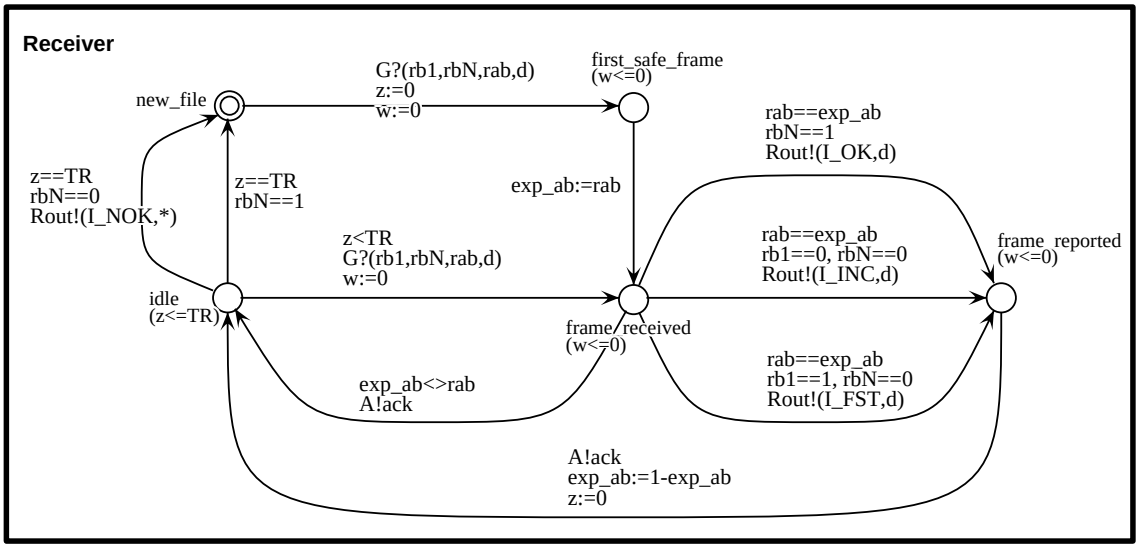


Figure 2: Timed automaton for receiver R.

timer z expires (i.e., $z = \mathsf{TR}$) 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 acknowledgement to arrive at S) while an acknowledgement generated by R is possibly still to come.

4 UPPAAL

UPPAAL [4, 5] 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 3.2. 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]. UPPAAL facilitates the graphical description of timed automata by using Autograph. The output of Autograph is compiled into textual format, which is checked for syntactical correctness. The textual representation is one of the inputs to UPPAAL's verifier. This 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 can be violated. UPPAAL also provides a simulator that allows a graphical visualization of possible dynamic behaviors 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. An overview of UPPAAL is depicted in Figure 3.

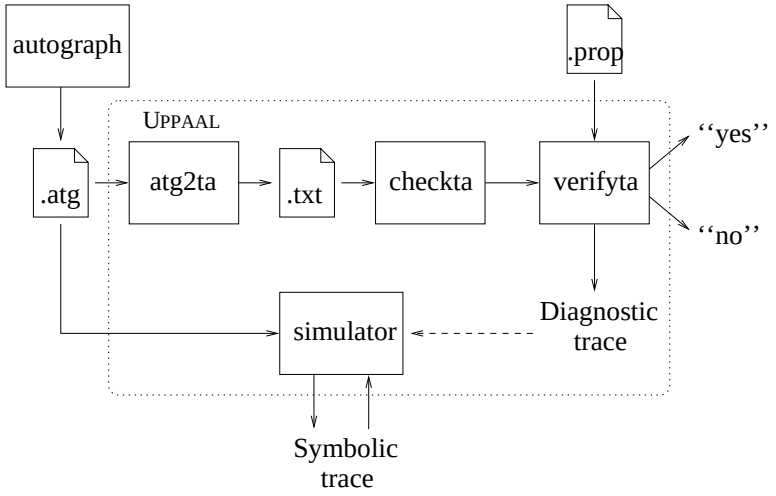


Figure 3: Overview of UPPAAL.

In the current version (i.e., β-release 1.99), UPPAAL is able to check only reachability properties. Properties are terms in the language defined by the following syntax:

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

where a is 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 1.99 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).

4.1 Protocol model in UPPAAL

The UPPAAL models of the sender S and the receiver R are a straightforward adaptation of the specifications given in Section 3.2; see Figure 4. (Nomenclature is similar to Section 3.2 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.

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 4).

[3]Notice that in Figure 4, values like n, TD or MAX have not yet been instantiated, but they must get a concrete value for each UPPAAL verification run. (See Section 4.3.)

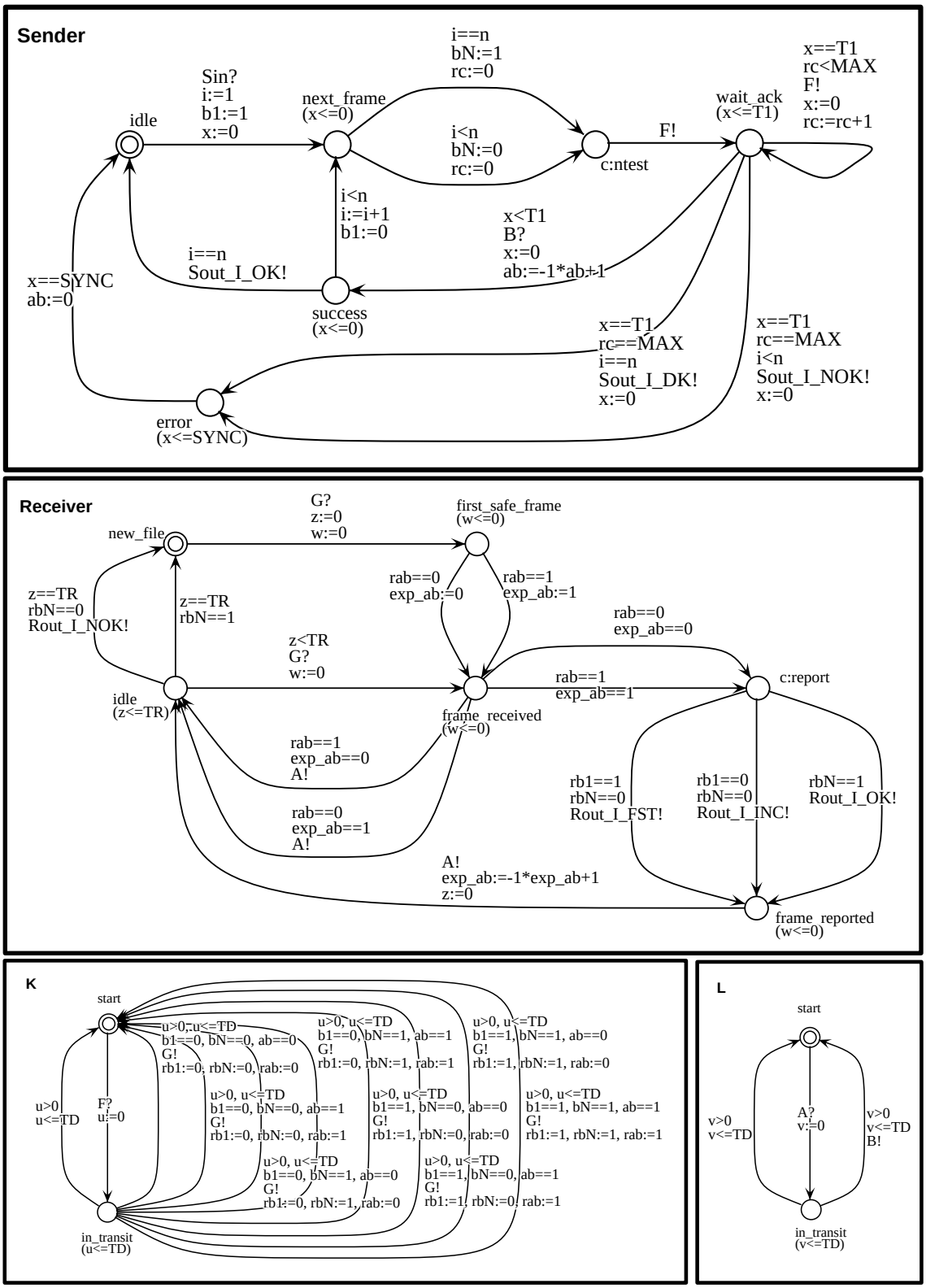


Figure 4: The protocol in UPPAAL.

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 number of states and transitions. This induces two main problems. Firstly, the original simple specification 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 number of locations, clocks, and variables. Therefore, we decided to abstract from 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 to global variables. Due to the above mentioned restriction on integer assignments, however, we had to expand 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 4.

We use the so-called *committed locations* [3]. Committed locations are states which introduce the notions of atomicity and urgency. 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).

4.2 Deducing time constraints

In this section we derive a tight constraint under which the modeling of channels K and L as one-place buffers is justified. In addition, we present tight 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 be reached when in location *in_transit* a new message is put in the channel (see Figure 5). 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

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

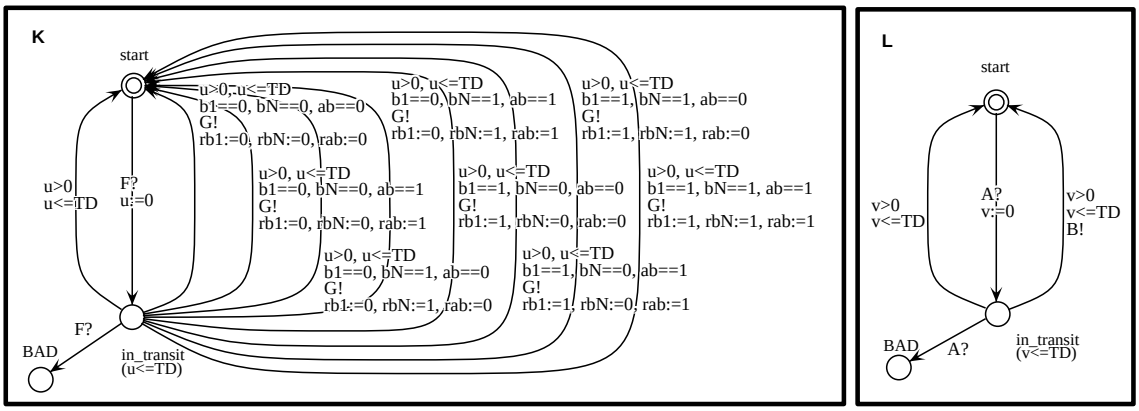


Figure 5: Channel with checking state for overflow

could also 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 acknowledgement 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 process *Lines* (see Figure 6) which is a smaller process with one location and one clock less. In [7] we performed this in order to reduce the memory consumption of a previous version of UPPAAL. This "trick" was not necessary in our latest work using UPPAAL 1.99.

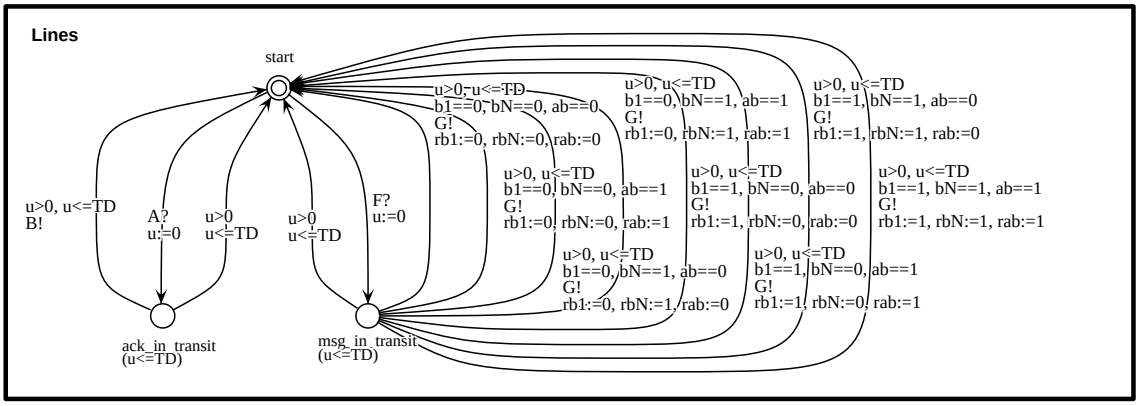


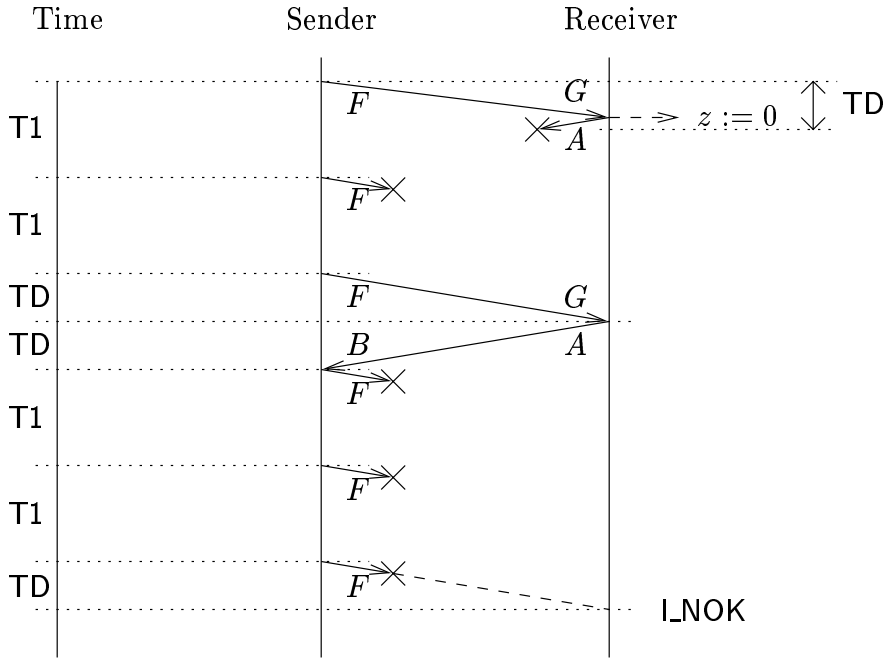
Figure 6: Alternative channels.

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 $\mathsf{TR} \geqslant 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$. In order to conclude this, the simulator of UPPAAL together with the diagnostic traces have been of big help. We were able to try different values for TD, T1, and MAX, and thus, to study the behavior of the protocol. Figure 7 depicts the longest trace that makes the protocol loose the connection when $\mathsf{MAX} = 2$ and $n \geqslant 2$. The $\times$ after sending a frame represents that it is lost in some of the channels K or L. Notice that frames are received within TD time units but they always take some time to travel through the channel. In particular, in the transmission of the first frame, the difference in time between synchronization on F and synchronization on G cannot be 0.



$= 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$

Figure 7: Loosing the connection.

From the figure, it is clear that the receiver should not timeout (strictly) before $2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$ units of time since this is the last time a frame can arrive.

Premature timeout would induce the receiver to abort the connection when there is still the possibility of some frame to arrive. As a result, property (2) would be disproved.

We remark that Figure 7 is just our way of depicting this situation (somehow based on Message Sequence Charts [20]). UPPAAL notation is textual. In particular, the simulator would show the trace as a symbolic trace, i.e. a trace in which the clock values are identified with regions. Instead, the diagnostic trace would be a concrete trace, that is, clocks have a real value in this case.

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*. This can be expressed in Timed-CTL [2] as

$$\forall\ (S.error\ \mathcal{U}_{\leqslant \mathsf{SYNC}}\ R.new\_file) \tag{3}$$

Unfortunately, the property language of UPPAAL does not support this type of formula. Therefore, we checked the following property:

$$\forall \Box\ ((S.error\ \wedge\ x = \mathsf{SYNC})\ \Rightarrow\ R.new\_file) \tag{4}$$

The difference between properties (3) and (4) is that (3) requires that *S.error* is true until *R.new_file* becomes true, while (4) 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—(4) implies (3). Property (4) is satisfied under the condition that $\mathsf{SYNC} \geqslant \mathsf{TR}$. This means that **(A2)** is 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 if the following constraints hold

$$\boxed{\mathsf{T1} > 2 \cdot \mathsf{TD} \qquad \text{and} \qquad \mathsf{SYNC} \geqslant \mathsf{TR} \geqslant 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}} \tag{5}$$

(Remark that SYNC and T1 are constants 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.

4.3 Protocol verification

In order to verify that the protocol satisfies the FTS specification of Section 2.2, we consider two additional automata that represent the clients at each side of the protocol: the sender client (*SC*) and the receiver client (*RC*). Besides, we need 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 $k > 0$. These auxiliary automata allow us to express the requeriments of Section 2.2 as properties of their states. They are depicted in Figure 8.

When trying to verify the correctness of the BRP using UPPAAL we encounter the following problems. Firstly, the properties constituting the FTS specification of

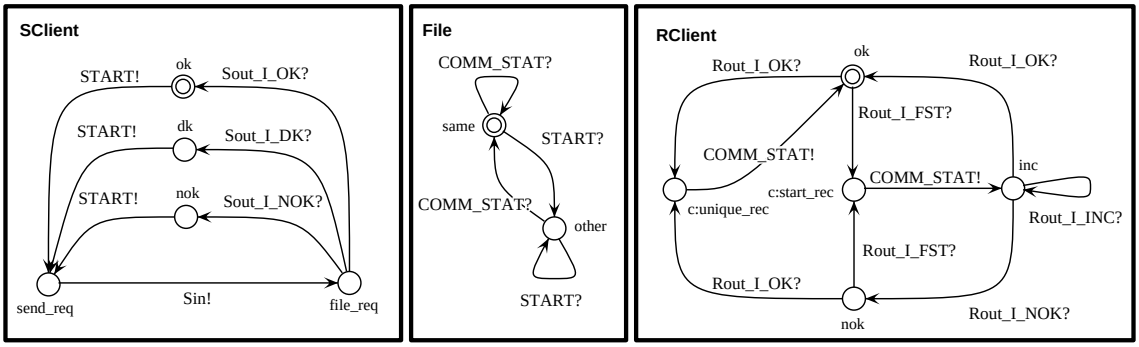


Figure 8: Auxiliary automata (general).

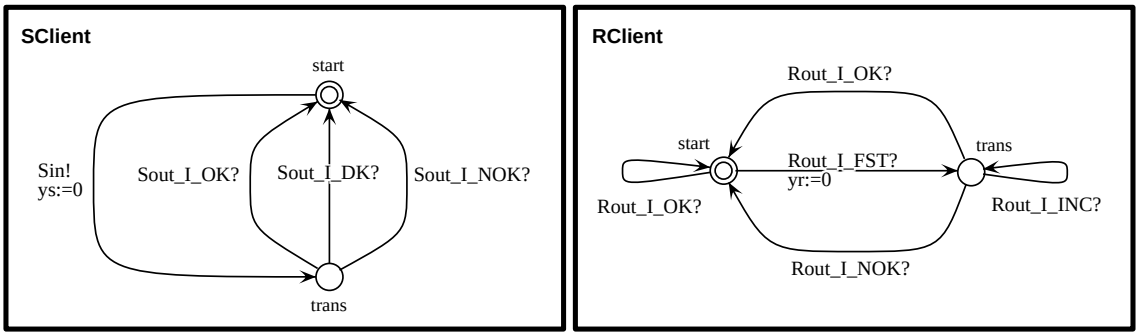


Figure 9: Auxiliary automata (bounded retransmission).

Section 2.2 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 have decided to remove 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 $SC.idle = (SC.ok \vee SC.dk \vee SC.nok)$ and $RC.idle = (RC.ok \vee RC.nok)$. 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 9. 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.

Table 2: Properties in UPPAAL.

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

The properties were proven in the following settings:

$\mathsf{TD} = 1$	$\mathsf{MAX} \in \{1,2,3\}$	$\mathsf{TR} = 2 \cdot \mathsf{MAX} \cdot \mathsf{T1} + 3 \cdot \mathsf{TD}$
$\mathsf{T1} = 3$	$n \in \{1,2,3\}$	$\mathsf{SYNC} = \mathsf{TR}$

For properties 9. and 10, and considering $n \geqslant 3$, we obtain that they are satisfied if and only if T and T' satisfy the following constraints.

$$T \geqslant (n \cdot \mathsf{MAX} + 1) \cdot \mathsf{T1} + (n \Leftrightarrow 1) \cdot 2 \cdot \mathsf{TD}$$
$$T' \geqslant (n \Leftrightarrow 1) \cdot \mathsf{MAX} \cdot \mathsf{T1} + (2n \Leftrightarrow 3) \cdot \mathsf{TD} + \mathsf{TR}$$

For the case of $n < 3$, these constraints are not tight. This is not surprising since, in this case, the sequence of chunks does not cover all the possibilities. In particular, notice that there is no chunk with $b1 = 0$ and $bN = 0$. The reader may check how those constraints were obtained by referring to Appendix B.

We have also played with some slight modifications of the channels, namely using constraints $u = \mathsf{TD}$ or $u \leqslant \mathsf{TD}$ instead of $0 < u \leqslant \mathsf{TD}$ in order to get more insight in the protocol. We have not reported results for this case since we consider the constant delay or possibility of 0 delay in the channel to be unrealistic.

As an example of the performance of UPPAAL, we give in Table 3 some results of time usage for the verification of a set of properties. The set of properties includes properties 1. to 6. in Table 2, and properties (1), (2), and (4). We include the information of verifying the protocol under three different instantiations: $n = 3$ and $\mathsf{MAX} \in \{1,2,3\}$. The other values are taken as before. We run UPPAAL under SunOS Release 5.5 on a SPARCstation 10, Model 402, with 96 MB and 210 MB of main and virtual memory, respectively. The left column shows the performance of the verification running in normal mode. The values in the right column were taken under the optimization mode of UPPAAL[4]. This mode only makes sense when verifying a set of properties rather than only one. Column "User" gives the time in seconds of CPU time devoted to the user's process, i.e., the verification process. Column "Sys" gives the time in seconds of CPU time consumed by the kernel on behalf of the user's process. Column "Normal" gives, in minutes and seconds, the elapsed (wall clock) time since the verification process started

[4]Flag `-T` of `verifyta`.

Table 3: Time usage by UPPAAL verifications

MAX	Normal mode User	Sys	Normal	CPU	Optimization mode User	Sys	Normal	CPU
1	59.36	0.26	1:01.94	96.2%	8.24	0.25	0:08.67	97.9%
2	151.35	0.49	2:36.81	96.8%	20.68	0.26	0:23.05	90.8%
3	330.39	0.73	5:40.24	97.3%	43.16	0.36	0:44.20	98.4%

to run until it finished. Column "CPU" gives the percentage of CPU usage, that is, the "User" time plus the "Sys" as a percentage of "Normal". The time values were obtained by using the UNIX command `time`.

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 [18, 19]. 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 [17, 18] at the expense of completeness.

In this section we summarize the validation efforts with SPIN. The annotated PROMELA models which we used during our validations are discussed in Appendix C.

To get confidence in the service specification from Section 2.2, we have written a PROMELA specification for the FTS. Each requirement of Section 2.2 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 following 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 10 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.

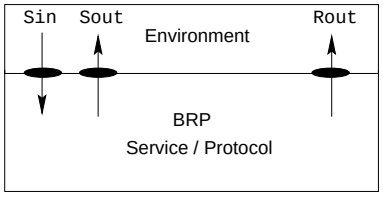


Figure 10: SPIN's validation model of the BRP and FTS.

The FTS description in PROMELA is obtained by a straightforward translation of the "external behavior" specification of [12] given in the process algebra μCRL [13]. In page 22 we have included the PROMELA `proctype` definition of the *Service* process.

PROMELA can handle data more easily than UPPAAL. In particular, UPPAAL only supports synchronization of processes without value passing. Therefore, to validate the data part of the BRP, PROMELA and SPIN are much more suitable than UPPAAL. Although PROMELA (SPIN version 2.7.7) is not able to handle timing aspects of the specification, we may simulate situations which depend on time by adding extra synchronization, relying on the results already obtained using UPPAAL. The BRP description in PROMELA is based on the formal protocol description of Section 3.2. Like for our UPPAAL verification we modeled channels K and L as one-place buffers. Figure 11 shows the structure of the BRP PROMELA description in terms of processes (represented as boxes) and channels (represented as arrows).

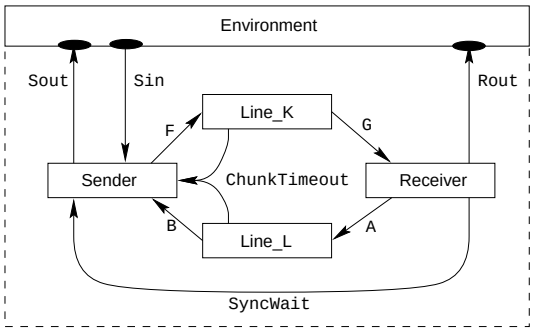


Figure 11: Structure of the BRP in PROMELA.

We used "tricks", analogous to [12] (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)** which have been already asserted in Section 4.2. 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 acknowledgement does not arrive in time at the sender S. So, if a

```
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 -> 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
}
```

frame is lost in channel K or its acknowledgement 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 aborted 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, this assumption is ensured by 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 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 SPIN version 2.9.7 on a Pentium 120Mhz computer with 64Mb RAM and 64Mb swap space, running Linux 2.0.27. SPIN managed to explore the complete state space and reported no errors with respect to the properties in Table 1. Details on the validation results can be found in Appendix C.

6 Other verifications of the BRP

The modeling and verification of the BRP has been the subject of several other papers. Groote & v.d. Pol [12] 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 behavior specification, also given in μCRL, by means of algebraic manipulation. Their proof boils down to finding an appropriate set of recursive (guarded) equations which has both the protocol and the external behavior specification as a solution. Using the recursive specification principle, which says that each guarded recursive equation has a unique solution, this proves the equivalence. Part of the proofs were checked using the proof-assistant Coq.

Helmink, Sellink & Vaandrager [15] 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 [14] 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 [23] translated the μCRL specifications of [12] into the process algebra LOTOS, and proved that the FTS and BRP specifications are branching bisimulation equivalent using the ALDÉBARAN tool. This tool can check several forms of bisimulation equivalence between labeled transition systems. 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 (simple) 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. For a time-dependent protocol like the BRP, timing analysis is necessary to establish complete correctness. Untimed analysis can only establish partial correctness.

7 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. The tools used for the verification were the protocol validation tool SPIN [18, 19], and the real-time verification tool UPPAAL [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. The protocol was formalized in Section 3.2 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. Due to the restrictions of the property language of UPPAAL (e.g., restricted use of variables) some service 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 protocol specification of Section 3.2. 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 (**(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. We provided tight timing constraints for which these assumptions are fulfilled (and so, they become valid assumptions). We obtained the constraints as follows. We started with weak constraints inspired by our intuition, checked these, and by interpreting the results, we sharpened the constraints a bit and repeated this procedure. For this purpose the simulator in UPPAAL has been of great help by visualizing the diagnostic traces obtained when running specifications that do not satisfy (5), see page 17. Diagnostic traces obtained for models that slightly violate (5) have helped to obtain the final constraints.

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 [12]. The main goal was to check the consistency of our service requeriments 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 [12, 14, 15, 23].

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 first 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. The use and need for a variety of verification methodologies has also been recognized by [26, 21].

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 danger of choosing language-oriented solutions in protocol modeling instead of concentrating on the bare protocol problems.

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 (in)famous alternating bit protocol, but which is still manageable. Although the BRP looks a bit simple at first sight (which made us under-estimate the effort necessary to model it), its timing intricacies make it an interesting example, especially for real-time verification tools such as RT-Spin [27], Kronos [9], and HyTech [16].

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.

A Properties of the FTS specification

Lemma. *The FTS specification satisfies the following properties for* $k > 0$*:*

1. $i_1 = \mathsf{I\_FST} \;\Rightarrow\; (k > 1 \;\wedge\; n > 1)$
2. *for all* j *such that* $0 < j \leqslant k$, $(i_j = \mathsf{I\_NOK} \;\vee\; i_j = \mathsf{I\_OK}) \;\Rightarrow\; j = k$
3. $i_1 \neq \mathsf{I\_NOK}$
4. $1 < k < n \;\Rightarrow\; i_k = \mathsf{I\_NOK}$
5. $k = 1 \;\Rightarrow\; n = 1$

Proof.

1. $i_1 = \mathsf{I\_FST}$
   $\Rightarrow$ $\{\}$
   $i_1 \neq \mathsf{I\_OK} \;\wedge\; i_1 \neq \mathsf{I\_NOK} \;\wedge\; k \neq 0$
   $\Rightarrow$ $\{\mathbf{(1.4.1)}\}$
   $k \neq 1 \;\wedge\; k \neq 0$
   $\Rightarrow$ $\{\text{calculus; } k \leqslant n\}$
   $k > 1 \;\wedge\; n > 1$

2. $i_j = \mathsf{I\_NOK} \;\vee\; i_j = \mathsf{I\_OK}$
   $\Rightarrow$ $\{\mathbf{(1.3)}\}$
   $j = 1 \;\vee\; j = k$

$\Leftrightarrow$ $\{\}$
$(n = 1 \ \wedge \ j = 1) \ \vee \ (n > 1 \ \wedge \ j = 1) \ \vee \ j = k$
$\Rightarrow$ $\{i_j \neq \mathsf{I\_FST}; \mathbf{(1.2)}\}$
$(n = 1 \ \wedge \ j = 1) \ \vee \ j = k$
$\Rightarrow$ $\{(n = 1 \ \wedge \ j = 1) \ \Rightarrow \ j = k\}$
$j = k$

3. $i_1 = \mathsf{I\_NOK}$
$\Rightarrow$ $\{\mathbf{(1.4.3)}\}$
$1 > 1$
$\Rightarrow$ $\{\}$
FALSE !!

4. $k \neq n$
$\Rightarrow$ $\{\mathbf{(1.4.2)}\}$
$i_k \neq \mathsf{I\_OK}$
$\Rightarrow$ $\{\mathbf{(1.4.1)}\}$
$i_k = \mathsf{I\_NOK}$

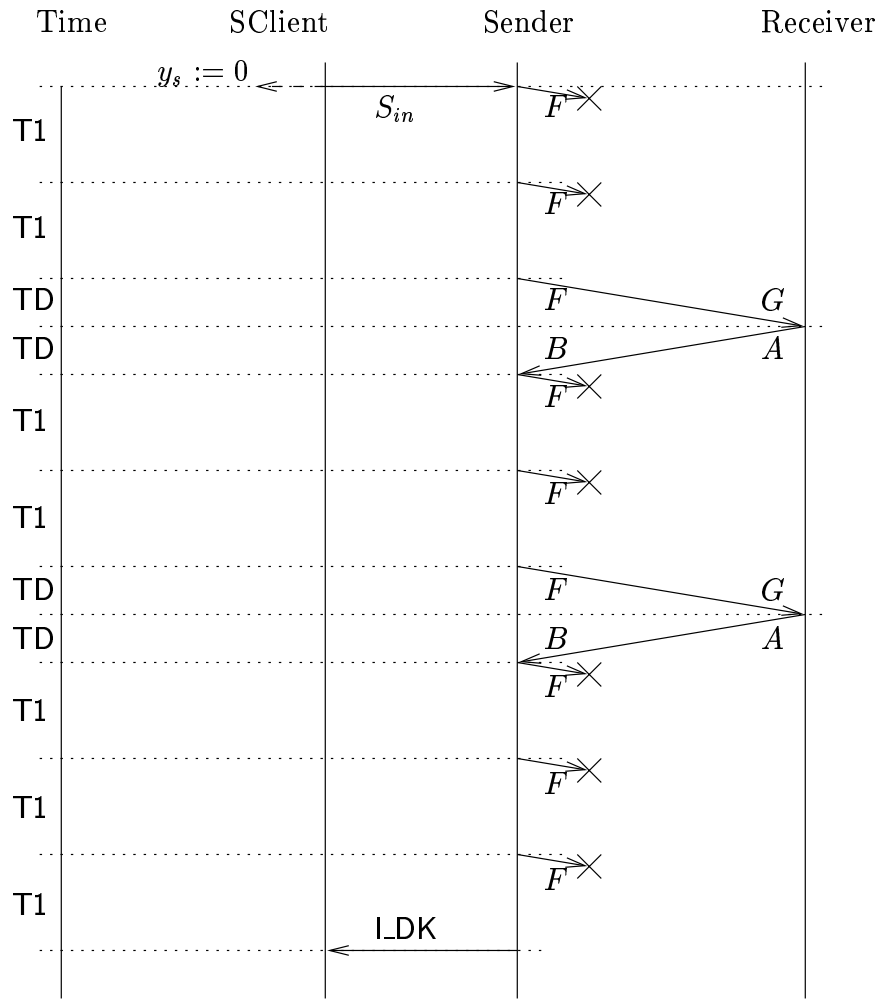
5. $k = 1$
$\Rightarrow$ $\{\mathbf{(1.4.3)}\}$
$i_k \neq \mathsf{I\_NOK}$
$\Rightarrow$ $\{\mathbf{(1.4.1)}\}$
$i_k = \mathsf{I\_OK}$
$\Rightarrow$ $\{\mathbf{(1.4.2)}\}$
$n = 1$ □

B Calculating the tightest values for T and T'

Figure 12 depicts the longest trace[5] in which the Sender is involved in a communication. This time is measured by clock y_s. In this figure we have the following valuations: $\mathsf{MAX} = 2$ and $n = 3$. Notice that each chunk consumes as much time as possible: $\mathsf{MAX} \Leftrightarrow 1$ failed transmissions $((\mathsf{MAX} \Leftrightarrow 1) \cdot \mathsf{T1})$, and a successful transmission of maximal transmission delay ($2 \cdot \mathsf{TD}$ for the first $n \Leftrightarrow 1$ chunks and $\mathsf{TD} > 2 \cdot \mathsf{TD}$ for the last frame which could be either lost as in the picture, or received but unsuccessfully acknowledged).

Similarly, Figure 13 depicts the longest trace in which the Receiver is involved in a communication. This time is measured by clock y_r. In this case we used the following valuations: $\mathsf{MAX} = 2$ and $n = 4$.

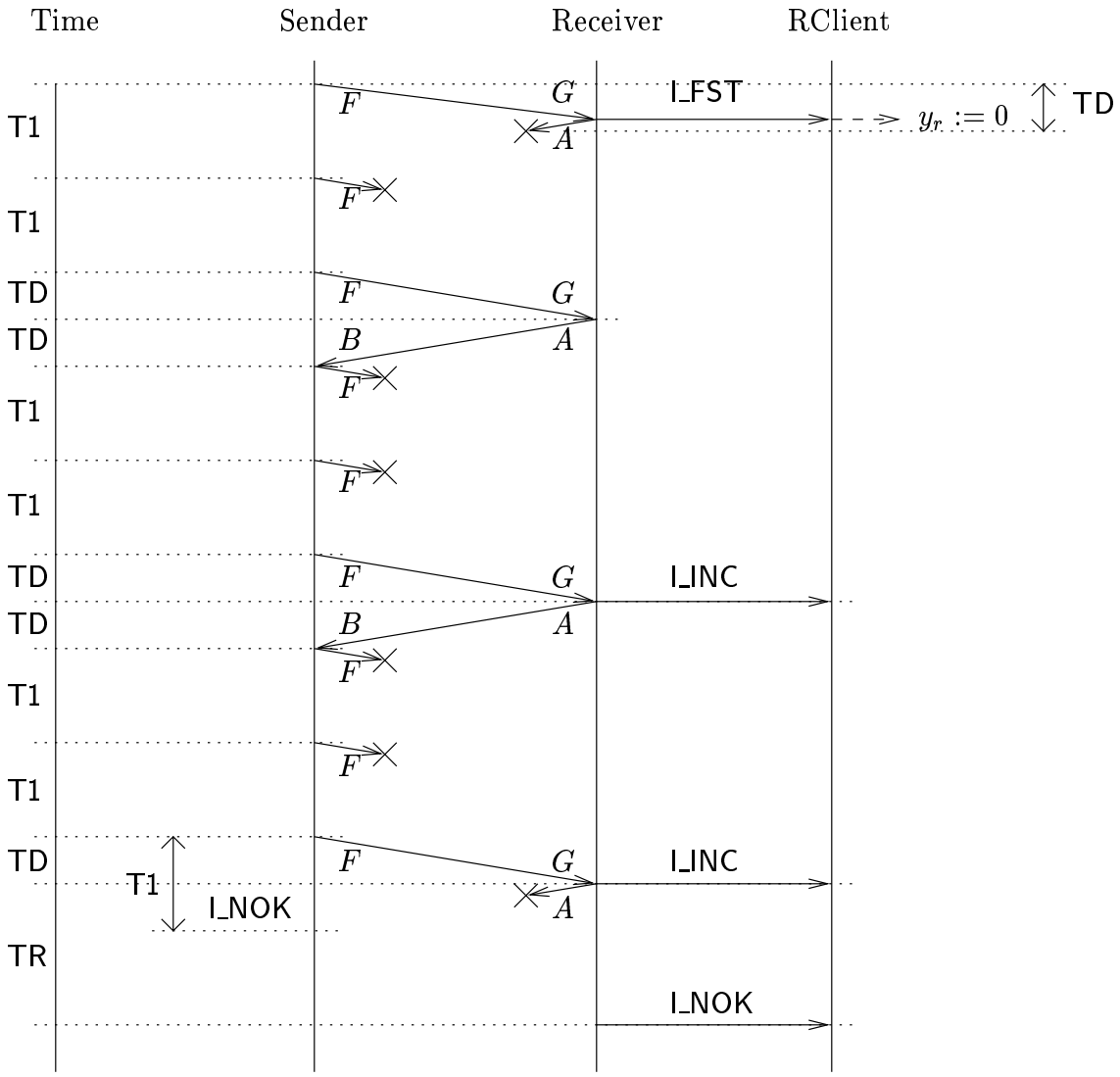
[5] 'Longest trace' in the sense of longest in time.



$= (n \Leftrightarrow 1) \cdot (\mathsf{MAX} \cdot \mathsf{T1} + 2 \cdot \mathsf{TD}) + (\mathsf{MAX} + 1) \cdot \mathsf{T1}$

$= (n \cdot \mathsf{MAX} + 1) \cdot \mathsf{T1} + (n \Leftrightarrow 1) \cdot 2 \cdot \mathsf{TD}$

Figure 12: Calculating the upper bound of the sending time.



$= (n \Leftrightarrow 2) \cdot (\mathsf{MAX} \cdot \mathsf{T1} + 2 \cdot \mathsf{TD}) + (\mathsf{MAX} \cdot \mathsf{T1} + \mathsf{TD}) + \mathsf{TR}$

$= (n \Leftrightarrow 1) \cdot \mathsf{MAX} \cdot \mathsf{T1} + (2n \Leftrightarrow 3) \cdot \mathsf{TD} + \mathsf{TR}$

Figure 13: Calculating the upper bound of the receiving time.

C Promela models

In this appendix we describe the PROMELA models of the Bounded Retransmission Protocol (BRP), which we used in our validation efforts using SPIN.

We give a *literate* description of our PROMELA versions of the BRP model. We will focus on the deviations from our "timed automata" model (as defined in Section 3.2), which we adopted in our "untimed" PROMELA specifications.

To make this appendix more readable, some notions, which have already been discussed earlier in this report, will be repeated in this appendix, though we will not recall the rationale behind the models, which have already been discussed in depth in Sections 2.1 and 3.1.

Reading the literate models

The models of the BRP are presented as *literate programs* [22, 25]. Literate programming is the act of writing computer programs primarily as documents to be read by human beings, and only secondarily as instructions to be executed by computers [25]. A literate program combines source code and documentation in a single file. Literate programming *tools* then parse such a file to produce either readable documentation or compilable source code.

For our modelling work we have used `noweb`, developed by Norman Ramsey. `noweb` [25] is a literate programming tool like Knuth's `WEB`, only simpler. Unlike `WEB`, `noweb` is independent of the programming language to be literated.

This appendix contains PROMELA *code chunks* intertwined by *document chunks*, like the one you are reading now. What follows is a code chunk.

30.1 ⟨*sample code chunk* 30.1⟩≡

```
proctype Silly()
{
  ⟨Silly's body 30.2⟩
}
```

In this code fragment, ⟨*sample code chunk*⟩ is defined. The name of a code chunk always has a *page.n* suffix, where *n* indicates the *n*-th chunk definition on page *page*. `noweb` also places this tag in the left margin of the definition of the code chunk. These cross-reference tags make it easy to locate the definitions of code chunks in the document.

When the name of a code chunk appears in the definition of another code chunk, it stands for the corresponding replacement text. In our simple example, ⟨*sample code chunk*⟩ uses the code chunk ⟨*Silly's body*⟩, which is defined as follows.

30.2 ⟨*Silly's body* 30.2⟩≡ (30.1) 31 ▷

```
do
:: skip
od ;
```

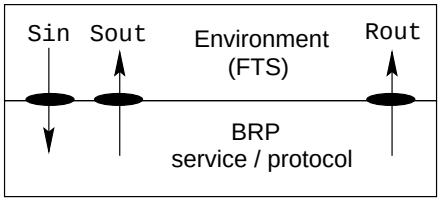


Figure 14: Service access points (SAPs) between the BRP and its Environment

In the right margin of the definition of a chunk, between `(..)` brackets, the tag of the code chunk is listed, which uses the particular code chunk. In this case, this is the tag of ⟨*sample code chunk*⟩.

It's possible and common practice to give the same name to several different code chunks. Continuing our example, we can expand our `Silly` process as follows.

31 ⟨*Silly's body* 30.2⟩+≡ (30.1) ◁30.2

```
assert(0) ;
```

The + ≡ here indicates that the code chunk ⟨*Silly's body*⟩ has appeared before. The PROMELA code following + ≡ will be appended to the previous replacement text for the same name.

When such continuations of code chunk definitions are used, `noweb` provides more information in the right margin; it indicates the previous definition (◁) and the next definition (▷) of the same code chunk.

C.1 Introduction

To get confidence in the *logical* service specification of the FTS as defined in Section 2.2 we have written an alternative *service* specification of the BRP in PROMELA. This *process algebraic* service description of the BRP is obtained by a straightforward translation of the "external behaviour" specification in [12].

Figure 14 shows an overview of the communication between the BRP and its environment. Either the *service* description of the BRP or the *protocol* description of the BRP can be "plugged" in. The environment drives both the service and the protocol and as such it is comparable with the processes `SClient`, `RClient` and `File` of the UPPAAL validation model of the BRP as defined in Section 4.3.

The environment feeds the BRP with the file chunks to be transferred at `Sin` and receives the indications at `Sout` and `Rout`. After receiving all indications of the transmission of a single file, the `Environment` process checks the validity of the indications against our *logical* service specification of the FTS.

The *environment* of the BRP is discussed in Section C.3. The description of the BRP *service* is discussed in Section C.4, while the various PROMELA models of the BRP *protocol* are presented in Section C.5. But first, in the next Section, we define the PROMELA data structures to be used by all processes.

C.2 Common data structures

This section describes the PROMELA data structures and constants which are common to both the service and protocol description of the BRP.

32.1 ⟨*prelude* 32.1⟩≡ (41.4 46.1 55 58)
```
⟨m4 macros 32.2⟩
⟨constants 33.4⟩
⟨macros 42.2⟩
⟨types 33.1⟩
⟨channels 33.3⟩
⟨globals 32.3⟩
```

To be more specific, ⟨*prelude*⟩ embodies the PROMELA constants, macros, types, channels and shared variables which are used by both the service and the protocol specification of the BRP.

The fragment ⟨*m4 macros*⟩ needs some explanation. In Section C.3.1 we will see that the macro facilities of the preprocessor `cpp` are not sufficient to implement an efficient randomizing function. To implement an optimized randomizing scheme, we had to resort to the more powerful UNIX macro processor, `m4`.

File chunks and indications The constant `MAX_CHUNKS` holds the maximum number of chunks in a file.

32.2 ⟨*m4 macros* 32.2⟩≡ (32.1) 37▷
```
define('MAX_CHUNKS', 4)
```

The fragment above is the `m4` way to say

```
#define MAX_CHUNKS  4
```

In Section C.3.1, the `m4` constant `MAX_CHUNKS` will be used to generate a file of random length, filled with random data chunks.

Note that in PROMELA, array indexing starts at `0`, while in our logical description of the FTS (see Section 2.2) and our model of the FTS (see Section 3.2) array indexing starts at `1`. To be as close as possible to our models of the FTS and the BRP (e.g. to minimize the changes of "off-by-one-bugs") we also start counting at `1` in our PROMELA specifications. This means that the actual maximum number of chunks in a file is `MAX_CHUNKS-1`.

32.3 ⟨*globals* 32.3⟩≡ (32.1)
```
byte    n ;
byte    d[MAX_CHUNKS] ;
```

Conceptually, the list of file chunks to be sent by the sending part of the environment is local to the `Environment` process and should be communicated as a whole at `Sin`. In our model, however, we represent this file chunk by a global array `d[1..n]`. This array `d` is filled by the `Environment`, and its values are used by the underlying layer.

33.1 ⟨*types* 33.1⟩≡ (32.1) 33.2 ▷

```
mtype {
  Ifst, Iinc, Iok, Idk, Inok
} ;
```

The message type `mtype` enumerates all possible indications which the environment can receive at `Sout` and `Rout`.

33.2 ⟨*types* 33.1⟩+≡ (32.1) ◁33.1

```
typedef etype {
  byte    ind ;
  byte    val ;
} ;
```

The `etype` type encapsulates a tuple *(ind,val)* which is 'received' by the `Environment` at `Rout`. The variable `ind` is the indication `Ifst`, `Iinc`, `Iok` or `Inok`, whereas `val` is the data value of the chunk.

Service access points The actual communication between the environment and the BRP at the service access points (see Figure 14) is modelled by the following handshake channels.

33.3 ⟨*channels* 33.3⟩≡ (32.1)

```
chan Sin  = [0] of {bit} ;
chan Sout = [0] of {mtype} ;
chan Rout = [0] of {mtype, byte} ;
```

At `Sout`, the environment only receives indications of type `mtype`. At `Rout`, the environment receives tuples *(ind,val)*.

We model the communication at `Sin` by a single `bit`. The actual file of chunks that should be communicated at `Sin` is represented by the global array `d` (see Section C.3).

33.4 ⟨*constants* 33.4⟩≡ (32.1)

```
#define REQ         1
```

The symbolic name `REQ` is used to represent the request at `Sin` to send a new file.

Atomicity In the timed automata model of the BRP, the transitions between two states are atomic. In our PROMELA model we can force this atomicity using `atomic` clauses around the transitions. For readability reasons, however, we have chosen to limit the use of `atomic` constructs to those cases where atomicity is really needed.

After studying the complete model of the BRP in this appendix, it will become clear that the untimed PROMELA version of the BRP behaves like a *token game*. The process that has the token (i.e. a data chunk or an acknowledgement) is the only process that can proceed. By passing the token, the process will reactivate another process, whilst stopping itself.

Because of this *token passing* nature of the BRP, the state space is not affected too much by the non-atomicity of the transitions in our PROMELA model.

C.3 Environment process

The proctype `Environment` models the environment of the BRP. See Figure 14 on page 31 which shows the service access points `Sin`, `Sout` and `Rout` between the BRP and the `Environment`.

34 ⟨*Environment process* 34⟩≡ (41.4 46.1)

```
proctype Environment()
{
  ⟨Environment: locals 35.1⟩

  do
  :: Sin!REQ -> atomic {
       ⟨Check previous file 39.1⟩
       ⟨Generate new file 35.3⟩
     }
     Sin!REQ

  :: Sout?sInd

  :: Rout?i(v) -> d_step {
       k++ ;
       e[k].ind = i ;
       e[k].val = v ;
     }
  od
}
```

The `Environment` process works as follows. The body of `Environment` is an infinite `do`-loop, which has three guards: for each service access point (namely `Sin`, `Sout` and `Rout`) there is a seperate guard.

`Sin`: The `Environment` tries to handshake with the underlying service or protocol on `Sin` to start the transmission of a new file. If the synchronisation with the underlying layer succeeds, we are certain, that the `Environment` has received all `Sout` and `Rout` indications of the *previous* file.

So before really starting with the next file, we first check whether the transmission of the previous file was consistent with the requirements. This is done in ⟨*Check previous file*⟩[6].

In ⟨*Generate new file*⟩ a new file of chunks is generated. The variable `n` gets a value and the array `d[1..n]` is filled with random values.

[6] This is valid because of assumption **(A2)** we ensure that the sender does not start to transmit the first chunk of a new file if the receiver did not abort or finish successfully.

To ensure that the actual transmission of the file does *not* start *before* the results of the previous file are checked and the new file is generated, a *second* `Sin!REQ` is offered after generating the new file. The `Sender` will have to wait for this second `Sin!REQ` before it can start sending the first chunk of the new file.

Instead of using two `Sin!REQ`s it would have been more elegant if we could make the rendez-vous atomic:

```
atomic {
  Sin!REQ ->
  ...
}
```

Unfortunately, this does not work in SPIN (version 2.9.x). The reason for this is that in PROMELA, the sending entity of a handshake *initiates* the rendez-vous, whereas the receiving entity *finishes* the handshake. This means that only the 'receiving handshake' can be part of an atomic sequence.

Consequently, a single atomic `Sin?REQ` in the `Environment` process would work correctly. We felt, however, that this was not appropriate in the `Environment` process; the *receiving* intuition behind the `?` operator does not reflect the role of the `Environment`: to initiate the sending of a file.

`Sout:` At the `Sout` channel, the `Environment` receives the indication of the sending entity of the underlying layer. This indication is stored in the local variable `sInd`.

35.1 ⟨*Environment: locals* 35.1⟩≡ (34 56.1) 35.2 ▷

```
byte    sInd ;
```

`Rout:` At the `Rout` channel, the `Environment` receives the indication tuples *(ind,val)* of the receiving entity of the underlying layer. These tuples are stored in the array `e[1..k]`.

35.2 ⟨*Environment: locals* 35.1⟩+≡ (34 56.1) ◁35.1 36.1 ▷

```
byte    k, i, v ;
etype   e[MAX_CHUNKS] ;
```

In ⟨*Check previous file*⟩ the array `e[1..k]` is checked against the array of chunks `d[1..n]`, which was transmitted by the underlying layer. The temporary variables `i` and `v` are used to receive the message at `Rout`.

C.3.1 Generating a file

In ⟨*Generate new file*⟩ the variable `n` is set and the array `d[1..n]` is filled.

35.3 ⟨*Generate new file* 35.3⟩≡ (34)

```
⟨Random n 36.2⟩
⟨Fill array d 38⟩
k = 0 ;
```

The variable `k` is reset, to indicate that we did not receive any indications at `Rout`. That is, the invariant "`e[1..k]` has been received at `Rout`" is satisfied.

36.1 ⟨*Environment: locals* 35.1⟩+≡ (34 56.1) ◁35.2

```
byte    j ;
```

In PROMELA, the most straightforward and efficient way to assign a random value to a variable is probably the following PROMELA fragment using an `if` statement:

```
if
:: n=1
:: n=2
:: n=3
fi ;
```

where the variable `n` gets a random value in the range `1..3`. In our model, the maximum value for `n` is `MAX_CHUNKS-1`. Unfortunately, we cannot write a `if` statement using this symbolic constant, because we do not know how many guards there have to be.

On the other hand, we can use PROMELA ordinary looping construct `do` to set `n` to an arbitrary value in the range `1..MAX_CHUNKS-1`. So a possible implementation of the ⟨*Random n*⟩ could be:

```
j = 1 ;
do
::  j <  MAX_CHUNKS-1 -> j++
::  j <  MAX_CHUNKS-1 -> n = j; break
::  j >= MAX_CHUNKS-1 -> n = MAX_CHUNKS-1; break
od ;
```

It is however not surprising that compared with the straightforward `if` construction this `do`-construct is very inefficient with respect to the state space.

For that reason we looked for another way to generate the *randomizing* `if` construction. And that is where the `m4` macro processor came in. With `m4` it is possible to define looping constructs which depend on earlier defined constants.

36.2 ⟨*Random n* 36.2⟩≡ (35.3 56.2)

```
if
forloop(‘i’, 1, eval(MAX_CHUNKS-1), ‘:: n = i
’)fi ;
```

In ⟨*Random n*⟩, we use a `m4` macro `forloop`, to create the following `if` statement, that assigns `n` a random value in the range `1..MAX_CHUNKS-1`:

```
if
::  n = 1
::  n = 2
::  ...
::  n = MAX_CHUNKS-1
fi ;
```

Where ... represent the guards between 2 and MAX_CHUNKS-1. It is not necessary to understand the construction used in ⟨*Random n*⟩; it is just a trick to generate the most efficient randomizing PROMELA construct.

Below the m4 macro forloop is defined, which is to be included in ⟨*m4 macros*⟩[7].

37 ⟨*m4 macros* 32.2⟩+≡ (32.1) ◁32.2

```
define(‘forloop’,
       ‘pushdef(‘$1’, ‘$2’)_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)popdef(‘$1’)’)
define(‘_forloop’,
       ‘$4‘’ifelse($1, ‘$3’, ,
       ‘define(‘$1’, incr($1))_forloop(‘$1’, ‘$2’, ‘$3’, ‘$4’)’)’)
```

Again, it is not needed to understand these m4 macro definitions.

Now that we have a random n, we also want to fill the array d[1..n] with random data chunks. Again, the most efficient PROMELA construction makes use of if statements:

```
if :: d[1] = 1  :: d[1] = 2  :: ...  :: d[1] = n  fi ;
if :: d[2] = 1  :: d[2] = 2  :: ...  :: d[2] = n  fi ;
...
if :: d[n] = 1  :: d[n] = 2  :: ...  :: d[n] = n  fi ;
```

where the array d[1..n] is filled with random values in the range 1..n[8]. Note, that even with m4 we cannot construct this PROMELA construct because n is not a constant but a variable.

So it seems that this time, we have to implement ⟨*Fill array d*⟩ using a very inefficient PROMELA looping construction like:

```
j = 0 ;
do
::  j++ ;
    if
    ::  (j <= n) -> i = 1 ;
                    do
                    ::  (i <  n) -> i++
                    ::  (i <= n) -> d[j] = i; break
                    ::  else     -> d[j] = n; break
                    od
    ::  (j > n)  -> break
    fi
od ;
```

[7] The forloop macro itself is shamelessly copied from the info file that is part of the GNU distribution of m4.

[8] In Section C.6 we will see that is not necessary to use n different chunks. In this section, however, we follow our intuitive but naive idea that, to show that the BRP is correct, it is sufficient to show that every randomly filled file chunks is correctly received at the receiver's side.

This implementation may be straightforward, but the double `do` loop makes it, again, very inefficient. However, if we don't mind a lengthy `if` statement, there is a way out. Earlier, we said that `m4` cannot help us here because `n` is a *run-time* variable. This is true, but instead of the actual value `n`, we know its upperbound (i.e. `MAX_CHUNKS-1`). This means, that we can use the PROMELA construction below, to fill the array `d[1..n]`. Note that this construction only depends on `MAX_CHUNKS`.

```
if
::  n == 1              -> if :: d[1] = 1  fi
::  n == 2              -> if :: d[1] = 1  :: d[1] = 2  fi ;
                           if :: d[2] = 1  :: d[2] = 2  fi
    ...
::  n == MAX_CHUNKS-1 -> if :: d[1] = 1  :: d[1] = 2  ...  :: d[1] = n fi ;
                           if :: d[2] = 1  :: d[2] = 2  ...  :: d[2] = n fi ;
                           if :: d[3] = 1  :: d[3] = 2  ...  :: d[3] = n fi ;
                           ... ;
                           if
                           :: d[MAX_CHUNKS-1] = 1
                           :: d[MAX_CHUNKS-1] = 2
                           ...
                           :: d[MAX_CHUNKS-1] = n
                           fi
fi
```

It is not elegant, but it is the most efficient PROMELA construction with respect to the state space. Again we use the `m4` macro `forloop` to generate the `if` construction as presented above.

38 ⟨*Fill array d* 38⟩≡ (35.3)

```
if forloop(‘i’, 1, eval(MAX_CHUNKS-1), ‘
:: n==i -> forloop(‘j’, 1, i, ‘
     if forloop(‘k’, 1, i, ‘
     :: d[j] = k’)
     fi ifelse(j,i,‘’,‘;’)’)
’)fi ;
```

Needless to say that it is again not necessary to understand this `m4` construction; the bottom line is that it fills the array `d[1..n]` in the most efficient way.

C.3.2 Checking the requirements

The process algebraic service description and the protocol description of the BRP are checked against the *logical* service specification of the FTS as defined in Section 2.2. In ⟨*Check previous file*⟩ we use the fact that the logical service requirements of the FTS are just constraints on the (list of) chunks offered at `Sin` and the indications received at `Sout` and `Rout`. In our model, each requirement has been translated into a sequence of 1 or more PROMELA statements involving assertions. We use the same numbering of the requirements as in Section 2.2.

39.1 ⟨*Check previous file* 39.1⟩≡ (34 56.1)

```
if
::  (n == 0) -> skip
::  (n >  0) -> if
                ::  (k >  0) -> ⟨Requirements for k > 0 39.2⟩
                ::  (k == 0) -> ⟨Requirements for k = 0 41.2⟩
                fi
fi ;
```

The first time when ⟨*Check previous file*⟩ is 'called', there is no file to check, because there is no previous file. We take care of that by initially setting `n` to `0`. In all other cases, `n` will be greater than `0`.
⟨*Requirements for* $k > 0$⟩ consists of all requirements that should hold when $k > 0$. The requirement

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

is represented by the following PROMELA fragment:

39.2 ⟨*Requirements for* $k > 0$ 39.2⟩≡ (39.1) 39.3▷

```
/* requirement 1.1 */
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 ;
printf("Finished checking requirement 1.1\n") ;
```

The requirement

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

is represented by the following PROMELA fragment:

39.3 ⟨*Requirements for* $k > 0$ 39.2⟩+≡ (39.1) ◁39.2 40.1▷

```
/* requirement 1.2 */
if
::  (n > 1) -> assert(e[1].ind == Ifst)
::  else    -> skip
fi;
printf("Finished checking requirement 1.2\n") ;
```

The requirement

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

is represented by the following PROMELA fragment:

40.1 ⟨*Requirements for* $k > 0$ 39.2⟩+≡ (39.1) ◁39.3 40.2 ▷

```
/* requirement 1.3 */
j=2;
do
::  (j >= k) -> break
::  (j <  k) -> assert(e[j].ind == Iinc) ; j++
od ;
printf("Finished checking requirement 1.3\n") ;
```

The requirement

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

is represented by the following PROMELA fragment:

40.2 ⟨*Requirements for* $k > 0$ 39.2⟩+≡ (39.1) ◁40.1 40.3 ▷

```
/* requirement 1.4.1 */
assert((e[k].ind == Iok) || (e[k].ind == Inok)) ;
printf("Finished checking requirement 1.4.1\n") ;
```

Requirements

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

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

are represented by the following PROMELA fragment:

40.3 ⟨*Requirements for* $k > 0$ 39.2⟩+≡ (39.1) ◁40.2 41.1 ▷

```
if
::  (e[k].ind == Iok)  -> assert(k == n)          /* 1.4.2 */
::  (e[k].ind == Inok) -> assert(k > 1)           /* 1.4.3 */
::  else -> skip
fi ;
printf("Finished checking requirement 1.4.2 and 1.4.3\n") ;
```

Requirements

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

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

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

are represented by the following PROMELA constructs:

41.1 ⟨*Requirements for* $k > 0$ 39.2⟩+≡ (39.1) ◁40.3

```
if
::  (sInd == Iok)  -> assert(e[k].ind == Iok)     /* 1.5 */
::  (sInd == Inok) -> assert(e[k].ind == Inok)    /* 1.6 */
::  (sInd == Idk)  -> assert(k == n)              /* 1.7 */
::  else -> skip
fi ;
printf("Finished checking requirement 1.5, 1.6 and 1.7\n")
```

⟨*Requirements for* $k = 0$⟩ consists of all requirements that should hold when $k = 0$. The requirement

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

is represented by the following PROMELA fragment:

41.2 ⟨*Requirements for* $k = 0$ 41.2⟩≡ (39.1) 41.3▷

```
/* requirement 2.1 */
assert( ((sInd == Idk) && (n == 1)) ||
        ((sInd != Idk) && (n != 1)) ) ;
printf("Finished checking requirement 2.1\n") ;
```

And finally, the requirement

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

is represented by the following PROMELA statement:

41.3 ⟨*Requirements for* $k = 0$ 41.2⟩+≡ (39.1) ◁41.2

```
/* requirement 2.2 */
assert( ((sInd == Inok) && (n > 1)) ||
        ((sInd != Inok) && (n <= 1)) ) ;
printf("Finished checking requirement 2.2\n")
```

C.4 BRP Service

In this section, we discuss a *process algebraic* service description of the BRP in PROMELA.

41.4 ⟨*brp-serv.m4* 41.4⟩≡
⟨*prelude* 32.1⟩
⟨*Environment process* 34⟩
⟨*Service process* 42.1⟩
⟨*Service init* 44.2⟩

The PROMELA specification of the model of the service is captured in the file ⟨*brp-serv.m4*⟩[9], and consists of the ⟨*prelude*⟩, ⟨*Environment process*⟩ as defined in Section C.3, a `Service` process and a service specific `init` process.

[9]Please recall that the PROMELA specifications are first processed by `m4` before they are fed to SPIN

C.4.1 Service process

The process `Service` defined below is a process algebraic adaptation of the μCRL specification of the *external behaviour* of the BRP as found in [12].

42.1 ⟨*Service process* 42.1⟩≡ (41.4 55)

```
proctype Service()
{
  byte j, k, v ;

  do
  ::  Sin?REQ ;
      Sin?REQ ;
      j = 0 ;
      k = 0 ;

      do
      ::  j++ ;
          v = d[j] ;
          if
          ::  tau -> ⟨Correct delivery of chunk at Rout 43.1⟩
          ::  tau -> ⟨Loss of a chunk 44.1⟩
          fi
      od
  od
}
```

The variable `j` is used as a chunk counter at the *sender*'s side, whereas the variable `k` is used as a chunk counter at the *receiver*'s side of the BRP service. The variable `v` holds the chunk to be transmitted. At the start of the inner `do`-loop, the values of `j` and `k` are such that the following invariants hold:

- the values `d[1..j]` have been sent to the receiver;
- the tuples `e[1..k]` have been offered at `Rout`.

where the arrays `d` and `e` have the same meaning as in section C.3. Note, however, that only the array `d` is accessible to `Service`.
The `Service` process uses the construct `tau`, which is just a redefinition of the statement `skip`:

42.2 ⟨*macros* 42.2⟩≡ (32.1)

```
#define tau         skip
```

`Service` is a non-terminating process. After receiving the pair of `REQ`s from the `Environment` process, the `Service` process just lists all *acceptable behaviour* of the BRP. That is, a chunk `v` is either correctly delivered at `Rout` or the chunk `v` is lost.

43.1 ⟨*Correct delivery of chunk at Rout* 43.1⟩≡ (42.1)

```
k++;
⟨Offer indication at Rout 43.2⟩
⟨Offer indication at Sout 43.3⟩
```

When we know that a chunk has been delivered correctly at the receiver's side, an indication has to be issued at `Rout` and optionally, an indication has to be issued at `Sout`.

43.2 ⟨*Offer indication at Rout* 43.2⟩≡ (43.1)

```
if
::  (j == n)              -> Rout!Iok(v)
::  (j != n) && (k == 1) -> Rout!Ifst(v)
::  (j != n) && (k >  1) -> Rout!Iinc(v)
fi ;
```

A chunk that is correctly received at the receiver's side is offered at `Rout` together with the suitable indication.

43.3 ⟨*Offer indication at Sout* 43.3⟩≡ (43.1)

```
if
::  (j == n)  -> if
                  ::  tau -> Sout!Iok; break
                  ::  tau -> Sout!Idk; break
                  fi
::  (j != n) ->  if
                  ::  tau -> skip
                  ::  tau -> Sout!Inok; k++; Rout!Inok; break
                  fi
fi
```

If `j==n` and the acknowledgement of the chunk was received at the sender's side, the sender should issue an `Iok` indication. If the acknowledgement was lost, the sender is not sure whether the complete file was correctly received at the receiver's side, so an `Idk` indication is issued at `Sout`. If `j!=n` and the acknowledgement is received by the sender, nothing happens at the service access points and the next chunk should be sent. If instead, the acknowledgement was lost, both the sender (at `Sout`) and the receiver `Rout` indicate the error to the `Environment`. Note that only when `j!=n` and the acknowledgement is correctly received at the sender's side, the `Service` process resumes the sending of the current file. In all other situations the process `break`s out of the `do`-loop.

44.1 ⟨*Loss of a chunk* 44.1⟩≡ (42.1)

```
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
```

When a chunk is lost (between the sender and the receiver), this is considered to be an error. If `k==0` and `j==n` (i.e. `n==1`), the sender is not sure whether the complete file (i.e. a single chunk) was received at the receiver side, and hence a `Idk` indication is issued. If `j!=n` it is clearly an error and `Inok` is offered. Note that nothing is offered at `Rout` because the receiver does not know that a new file has been sent. If `k>0` both the sender (at `Sout`) and the receiver (at `Rout`) report the error to the `Environment`. Again, there is distinction between `j==n` and `j!=n`.

After the loss of a chunk, the `Service` `break`s out of the `do`- loop to get ready for a new file.

C.4.2 `init`

44.2 ⟨*Service init* 44.2⟩≡ (41.4 55)

```
init {
  atomic {
    run Environment() ;
    run Service() ;
  }
}
```

The `init` process of the service just starts the `Service` process and the `Environment` processes.

C.4.3 Validation results

Directives file For all verification runs with SPIN we use a single data file (i.e. `brp-directives.dat`), which contains for each PROMELA specification of the BRP:

- the directives for the `C` compiler to build the `pan` analyser; and
- the run-time options for the `pan` analyser

From this *directives* file, a simple script generates the necessary commands to build the `pan` analyser and the `make` program runs this analyser to obtain the verification results.

Apart from directives and options which are specific to a particular specification, some *general* directives and options have been used for all verification runs:

45.1 ⟨*brp-directives.dat* 45.1⟩≡ 45.2 ▷

```
general
-D_POSIX_SOURCE -DSAFETY -DNOCLAIM -DNOFAIR
-c1
```

The first line identifies the PROMELA specification to be checked. The label `general` means that the directives that follow are to be used by all validations runs. The second line contains the directives for the `C` compiler, which builds the `pan` analyser. The third line holds the options which are used to execute the `pan` analyser.

The `_POSIX_SOURCE` directive indicates that the source to be compiled by the `C` compiler conforms to the Posix standard. The `SAFETY` directive is used because no liveness properties are checked in our validations. The `NOCLAIM` directive specifies that even if a PROMELA never claim is present, it is unabled. The `NOFAIR` directive disables the code for weak-fairness. Finally, the `-c1` option to the `pan` analyser specifies that the analyser should stop at the first error.

Machine All validation results presented in this Appendix are obtained with SPIN version 2.9.7, on a Pentium 120Mhz computer with 64Mb RAM and 64MB swap space, running Linux 2.0.29.

Directives for the service For the service description of the BRP, the following directives have been used:

45.2 ⟨*brp-directives.dat* 45.1⟩+≡ ◁45.1 54.3 ▷

```
brp-serv
-DMEMCNT=25
-w19 -m200000
```

The `MEMCNT` directive sets an upperbound to the amount of memory to be used by the analyser: `2^25` bytes (= 32Mb). The `-w19` option reserves a hashtable with `2^19` (= 512K) entries. The `-m200000` option limits the search depth to `200000` steps.

Results Running the `pan` analyser produces the following (stripped) output:

```
(Spin Version 2.9.7 -- 18 April 1997)  [run on 17-June-97 12:29:33]

State-vector 56 byte, depth reached 125892, errors: 0
  374898 states, stored
  264167 states, matched
  639065 transitions (= stored+matched)
  676883 atomic steps
hash conflicts: 245207 (resolved)
(max size 2^19 states)
```

```
Stats on memory usage (in Megabytes):
23.993  equivalent memory usage for states (stored*(State-vector + overhead))
        compressed State-vector = 36 byte + 8 byte overhead
16.537  actual memory usage for states (compression: 68.92%)
2.097   +memory used for hash-table (-w19)
4.800   +memory used for DFS stack (-m200000)
0.757   +memory used for other data structures
24.100  =total actual memory usage

Output from 'time':
        Command being timed: "./brp-serv.pan -c1 -w19 -m200000"
        User time (seconds): 17.43
        System time (seconds): 0.73
```

C.5 BRP Protocol

In this section we describe our PROMELA model of the BRP. The model is a straightforward translation of the *timed automata* model, which we have presented in Section 3.2. This straightforward PROMELA model turned out to be not very efficient with respect to the state space of the model. For example, on our computer, SPIN was not able to perform any exhaustive search of the state space (see Section C.5.3). In C.7 we discuss an optimized version of the protocol, which was efficient enough to be analysed completely by SPIN.

Figure 15 shows the relationship between the processes that model the BRP and the PROMELA channels which are used for communication between the processes. The arrows indicate the sending and receiving part of a communication. All channels are rendez-vous channels, so the distinction between sender and receiver process should not matter. However, PROMELA treats the sender and receiver differently in case the communication is part of an `atomic` clause. Hence, for reasons of clarity we have indicated the role of the processes.
The PROMELA specification of the BRP is defined as follows.

46.1 ⟨*brp-prot.m4* 46.1⟩≡
  ⟨*prelude* 32.1⟩
  ⟨*Environment process* 34⟩
  ⟨*Protocol prelude* 46.2⟩
  ⟨*Protocol processes* 48.3⟩
  ⟨*Protocol init* 54.2⟩

C.5.1 Protocol channels

The ⟨*Protocol prelude*⟩ consists of the constants and the channels which are specific to the BRP protocol.

46.2 ⟨*Protocol prelude* 46.2⟩≡ (46.1)
  ⟨*Protocol constants* 49.2⟩
  ⟨*Protocol channels* 47.1⟩

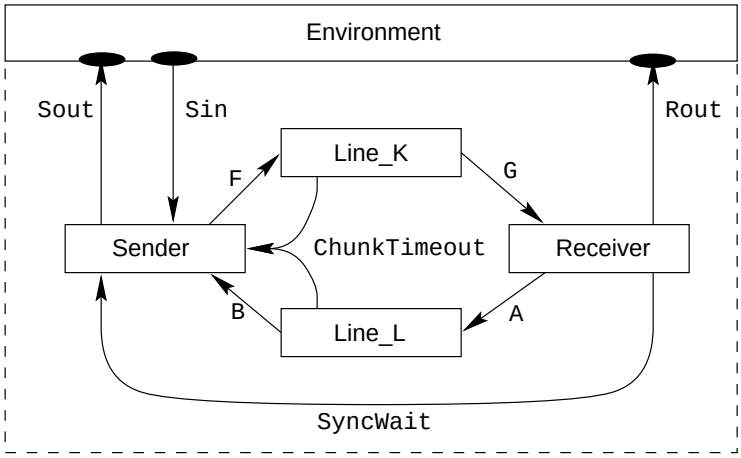


Figure 15: Relationship between the PROMELA processes of the BRP and the channels

Communication channels As in our formal specification of the BRP in Section 3.2, the line `K`, which is used to transmit the chunks, is connected to the sender and receiver using the channels `F` en `G`, respectively. The acknowledgement line `L` is connected to the sender and receiver using the channels `B` and `A`, respectively.

47.1 ⟨*Protocol channels* 47.1⟩≡ (46.2) 47.2 ▷

```
chan F = [0] of {bit, bit, bit, byte} ;
chan G = [0] of {bit, bit, bit, byte} ;
```

The channels `F` and `G` are used to exchange tuples of the form `(b1, bN, ab, d)`, where `b1` is a bit indicating whether this chunk is the first one, `bN` is a bit indicating whether this chunk is the last one, `ab` is the alternating bit and `d` models the data chunk itself.

47.2 ⟨*Protocol channels* 47.1⟩+≡ (46.2) ◁47.1 48.1 ▷

```
chan A = [0] of {bit} ;
chan B = [0] of {bit} ;
```

The channels `A` and `B` may only exchange the acknowledgement bit.

Recall that the lines `K` and `L` of the BRP act like one-place buffers. In this straigthforward translation from our timed automata to PROMELA, we use the processes `Line_K` and `Line_L` to implement the one-place buffer schemes (see Section 4.1). In Section C.7 we will replace these line processes and the channels `F`, `G`, `A` and `B` by PROMELA channels of length `1`, which are just buffers of length `1`.

Timing channels As discussed in Section 3.2, the BRP uses several timers to control the correct functioning of the sender and receiver. For example, a timer T_1 at the sender side is used which expires when an acknowledgement does not arrive in time. Because the current version of PROMELA does not have a notion of time, we have used "tricks", analogous to [12] to model timing and synchronisation of the BRP. These "tricks", in fact, correspond to the assumptions **(A1)** and **(A2)** of Section 3.1.

Assumption **(A1)** states that "premature timeouts are not possible". Thus, if the timer T_1 at the sender's side expires, it is certain that the acknowledgement from the receiver will not arrive anymore. We model this timer T_1 with a channel between the two lines `K` and `L` and the `Sender` process.

48.1 ⟨*Protocol channels* 47.1⟩+≡ (46.2) ◁47.2 48.2▷

```
chan ChunkTimeout = [0] of {bit} ;
```

When a message is lost in one of the unreliable lines `K` or `L` (i.e., a data chunk or an acknowledgement, respectively), the `Sender` will be notified via `ChunkTimeout` that an acknowledgement will not arrive anymore.

Assumption **(A2)** states that "in case of abort, sender S waits before starting a new file until receiver R reacted properly to abort". In our *timed* specification of the BRP, this assumption is ensured using two timers: one at the `Sender`'s side and one at the `Receiver`'s side. If the timer of one of the protocol entities expires, it will wait long enough to be sure that its peer entity has timed out as well.

48.2 ⟨*Protocol channels* 47.1⟩+≡ (46.2) ◁48.1

```
chan SyncWait     = [0] of {bit} ;
```

Again, we do not have such a mechanism in PROMELA, so again we use a handshake channel to synchronise the `Sender` and the `Receiver` after a failure. Note that this mimics the timer T_2 of [12].

C.5.2 Protocol processes

48.3 ⟨*Protocol processes* 48.3⟩≡ (46.1)

⟨*Sender process* 48.4⟩
⟨*Receiver process* 51.1⟩
⟨*Line processes* 53⟩

Below, the processes `Sender` and `Receiver` are defined. Naturally, the PROMELA definitions of both processes has been based on the definition of the processes as timed automata in Section 3.2.

Sender

48.4 ⟨*Sender process* 48.4⟩≡ (48.3 58)

```
proctype Sender()
{
```
⟨*Sender: locals* 48.5⟩
⟨*Sender: body* 49.1⟩
```
}
```

The `Sender` process uses some local variables and the process itself is defined in ⟨*Sender: body*⟩.

48.5 ⟨*Sender: locals* 48.5⟩≡ (48.4)

```
bit     ab ;
byte    rc ;
byte    i ;
```

The variable `ab` holds the current value of the *alternating bit*. The *retry counter* `rc` counts the number of times the last chunk has been retransmitted. The sequence number `i` indicates the i-th chunk of the current file.

49.1 ⟨*Sender: body* 49.1⟩≡ (48.4) 49.3 ▷

```
start:
  ab = 0 ;
  goto idle ;

idle:
  Sin?REQ ;
  Sin?REQ ;
  i = 1 ;
  goto next_frame ;

next_frame:
  F!(i==1),(i==n),ab,d[i] ;
  rc = 0 ;
  goto wait_ack ;
```

The `Sender` starts by setting the alternating bit to `0`. Then it moves to the `idle` state where it waits for the pair of `REQ`s from the environment (see Section C.3). In state `next_frame` the next chunk of a file is sent. For the first chunk, `i` is equal to `1`. The `Sender` then goes to the `wait_ack` state and waits for the acknowledgement from the `Receiver`.

49.2 ⟨*Protocol constants* 49.2⟩≡ (46.2 59.1) 50.1 ▷

```
#define ACK          1
```

49.3 ⟨*Sender: body* 49.1⟩+≡ (48.4) ◁49.1 50.2 ▷

```
wait_ack:
  if
  ::  B?ACK ->
        ab = 1-ab ;
        goto success

  ::  ChunkTimeout?SHAKE ->
        if
        ::  (rc < MAX)  -> rc++ ;
                           F!(i==1),(i==n),ab,d[i] ;
                           goto wait_ack
        ::  (rc >= MAX) -> goto error
        fi
  fi ;
```

If the Sender receives an acknowledgement ACK from the Receiver, then the sending of the last chunk was successful. The alternating bit is inverted, and the process goes to success. If the last chunk was lost in K or the acknowledgement for the last chunk was lost in L, the Sender is notified via ChunkTimeout?SHAKE. If the number of retries (rc) is still lower than MAX, the Sender will try to send the same chunk again, otherwise it gives up and goes to the error state.

50.1 ⟨*Protocol constants* 49.2⟩+≡ (46.2 59.1) ◁49.2

```
#define MAX         2
#define SHAKE       1
```

The constant MAX is the maximum number of *retransmission* attempts of the Sender to transmit a single chunk. The constant SHAKE is just a symbolic constant to model rendez-vous synchronisation (i.e., handshaking) of two processes.

50.2 ⟨*Sender: body* 49.1⟩+≡ (48.4) ◁49.3 50.3▷

```
success:
  if
  ::  (i == n) -> Sout!Iok ;
                  goto idle
  ::  (i <  n) -> i++ ;
                  goto next_frame
  fi ;
```

If i==n, the i-th chunk was the last chunk of the file. In this case the Environment is notified and the Sender goes back to the idle state. If it was not the last chunk (i!=n), the Sender sends the next chunk.

50.3 ⟨*Sender: body* 49.1⟩+≡ (48.4) ◁50.2

```
error:
  if
  ::  (i == n) -> Sout!Idk
  ::  (i != n) -> Sout!Inok
  fi ;
  SyncWait!SHAKE ;
  SyncWait?SHAKE ;
  ab = 0 ;
  goto idle ;
```

The Sender can only end up in the error state if it did not receive an acknowledgement for its i-th chunk. After notifying the environment of the error, the Sender waits for the Receiver to time out as well. The double synchronization on SyncWait is needed to fulfill assumption **(A2)**. We must make sure that the Environment will not start sending a new file (i.e. by synchronizing on Sin) before both the indications on Sout and Rout have been issued. While the Sender is waiting to synchronize on its second SyncWait message, the Receiver has time to offer its indication at Rout. After this second synchronization, the Sender will proceed to the idle state, where it may *receive* a new file at Sin.

Receiver Next we define the `Receiver` process of the BRP.

51.1 ⟨*Receiver process* 51.1⟩≡ (48.3 58)

```
proctype Receiver()
{
⟨Receiver: locals 51.2⟩
⟨Receiver: body 51.4⟩
}
```

51.2 ⟨*Receiver: locals* 51.2⟩≡ (51.1) 51.3 ▷

```
bit    b1, bN, ab ;
byte   v ;
```

The variables `b1`, `bN`, `ab` and `v` are the placeholders for the chunks received from the `Sender` at channel `G`. The bit `b1` is true if the chunk is the *first* chunk of the file, the bit `bN` is true if the chunk is the *last* chunk of the file, the bit `ab` is the alternating bit and the byte `v` is the data chunk itself.

51.3 ⟨*Receiver: locals* 51.2⟩+≡ (51.1) ◁51.2

```
bit    exp_ab ;
```

The variable `exp_ab` is the *expected* alternating bit in the next chunk from the sender.

51.4 ⟨*Receiver: body* 51.4⟩≡ (51.1) 52.1 ▷

```
new_file:
  if
  ::  G?b1,bN,ab,v   -> goto first_safe_frame
  ::  SyncWait?SHAKE ;
      SyncWait!SHAKE -> goto new_file
  fi ;

first_safe_frame:
  exp_ab = ab ;
  goto frame_received ;
```

The `Receiver` only ends up in the state `new_file` if the protocol is started for the first time or when the `Receiver`'s *timer* has expired. After receiving the first chunk of the `Sender`, it moves to the state `first_safe_frame`.

In the state `first_state_frame`, after receiving the first chunk of a new file, the alternating bit is initialized according to the `ab`-field found in the chunk from the `Sender`. Note the two synchronizations on `SyncWait` in the state `new_file`. These synchronizations on the *timed* channel `SyncWait` do not appear in our original timed specification of the BRP. They are needed in our PROMELA version of the BRP because, in case of an abort, the `Sender` always wants to synchronize with the `Receiver`. Consider the case when the first chunk of a file never reaches the `Receiver`. The `Sender` will eventually decide that there is an error and wants to synchronize with the `Receiver`. The `Receiver`, however, is still awaiting the first chunk in the state `new_file`. For this special case, the synchronizations on `SyncWait` have been introduced.

52.1 ⟨*Receiver: body* 51.4⟩+≡ (51.1) ◁51.4 52.2▷

```
frame_received:
  if
  ::  (ab != exp_ab) ->
        A!ACK ;
        goto idle

  ::  (ab == exp_ab) ->
        if
        ::  ( b1 && !bN) -> Rout!Ifst(v)
        ::  (!b1 && !bN) -> Rout!Iinc(v)
        ::  (        bN) -> Rout!Iok(v)
        fi ;
        goto frame_reported
  fi ;
```

After receiving a chunk (*frame*) with a wrong alternating bit, an acknowledgement is sent, but the `Environment` is not notified. The process moves to the state `idle` where it awaits a chunk with the correct alternating bit. After receiving a chunk with the expected alternating bit, the chunk is offered to the `Environment`. The process goes to the state `frame_reported` where the acknowledgement of the chunk is sent.

52.2 ⟨*Receiver: body* 51.4⟩+≡ (51.1) ◁52.1 52.3▷

```
frame_reported:
  A!ACK ;
  exp_ab = 1-exp_ab ;
  goto idle ;
```

After reporting the reception of the last chunk to the `Environment`, the `Receiver` acknowledges the chunk to the `Sender`, updates the alternating bit and goes to the `idle` state where it awaits a new chunk.

52.3 ⟨*Receiver: body* 51.4⟩+≡ (51.1) ◁52.2

```
idle:
  if
  ::  G?b1,bN,ab,v ->
        goto frame_received

  ::  SyncWait?SHAKE ->
        if
        ::   bN -> skip
        ::  !bN -> Rout!Inok
        fi ;
        SyncWait!SHAKE ;
        goto new_file
  fi ;
```

In the `idle` state, the `Receiver` can either receive a chunk from the `Sender` or, in case of an abort, synchronize with the `Sender` on `SyncWait`. If the `Receiver` has received a chunk, the validity of this chunk is checked in the state `frame_received`. In an abort situation, the `Receiver` will offer an `Inok` indication to the environment. However, if `bN` is *true* (i.e. the last correctly received chunk was the last of a file), the `Receiver` will already have issued its `Iok` to the `Environment` process. In this case the `Receiver` does not do anything. This is the situation in which the acknowledgement of the last chunk has never reached the `Sender`. The `Receiver` also moves to state `new_file` because it can no longer rely on the value of the alternating bit.

The double synchronization on `SyncWait` around the (possible) notification on `Rout` is very important here; it makes this notification *atomic*[10] with respect to the `Sender` process. As mentioned earlier, this *atomicity* is needed to fulfill assumption **(A2)**. For the `Receiver` this means that when synchronizing with the `Sender`, it should offer its indication (if any) to `Rout`, *before* the `Sender` can proceed to starting a new file at `Sin`.

Note that when the timed automata model is well timed, i.e., the conditions of Section 4.2 are satisfied, the `Receiver`'s timer may expire while waiting for a new file in the `idle` state. The `Receiver` then moves to the `new_file` state, *independently* of the `Sender`: the `Sender` cannot even time out in its `idle` state. We cannot model this behaviour in PROMELA: when there are no errors, the `Receiver` will always await the first chunk of the next file in the `idle` state. Note that a `tau`-transition to `new_file` does not help us here. It would mean that in the `idle` state the `Receiver` could always move to the `new_file` state.

Lines The processes `Line_K` and `Line_L` model the (unreliable) lines between the `Sender` and the `Receiver`. A message that is sent over one of these lines is either succesfully delivered at the other side or it is lost. In the latter case, the `Sender` is notified using the `ChunkTimeout` channel.

53 ⟨*Line processes* 53⟩≡ (48.3) 54.1 ▷

```
proctype Line_K()
{
  bit     b1, bN, ab ;
  byte    v ;

  do
  ::  F?b1,bN,ab,v -> if
                      ::  G!b1,bN,ab,v
                      ::  tau -> ChunkTimeout!SHAKE
                      fi
  od
}
```

[10] Note that it is not possible to use a single `SyncWait?` event (instead of two synchronizations) enclosed in an `atomic` clause. In PROMELA, the `atomic` step may be broken by the notification on `Rout`. If the synchronization on `Rout` is not executable, the atomicity will be lost.

54.1 ⟨*Line processes* 53⟩+≡ (48.3) ◁53

```
proctype Line_L()
{
  bit     b ;

  do
  ::  A?b -> if
             ::  B!b
             ::  tau -> ChunkTimeout!SHAKE
             fi
  od
}
```

Init The protocol's `init` just starts the processes that model the BRP and its environment.

54.2 ⟨*Protocol init* 54.2⟩≡ (46.1)

```
init {
  atomic {
    run Sender() ;
    run Receiver() ;
    run Line_K() ;
    run Line_L() ;
    run Environment() ;
  }
}
```

C.5.3 Validation results

Directives file Part of the directives file has been presented in Section C.4.3. For the protocol specification of the BRP, the following directives have been used:

54.3 ⟨*brp-directives.dat* 45.1⟩+≡ ◁45.2 57.3▷

```
brp-prot
-DMEMCNT=26 -DBITSTATE
-w26 -m700000
```

The `MEMCNT` directive reserves `2^26` bytes (= 64Mb) of memory for the `pan` anylyser. The `BITSTATE` directive selects SPIN's supertrace state space search. The `-w26` option instructs the `pan` analyser to reserve `2^26` (= 64M) entries in the hashtable. The `-m700000` option limits the search depth to `700000` steps.

Results Running the pan analyser produces the following (stripped) output:

```
(Spin Version 2.9.7 -- 18 April 1997)  [run on 17-June-97 12:30:05]

State-vector 104 byte, depth reached 502803, errors: 0
1.65918e+06 states, stored
  980115 states, matched
2.63929e+06 transitions (= stored+matched)
1.68096e+06 atomic steps
hash factor: 40.4471 (expected coverage: >= 98% on avg.)
(max size 2^26 states)

Stats on memory usage (in Megabytes):
179.191 equivalent memory usage for states (stored*(State-vector + overhead))
8.389   memory used for hash-array (-w26)
19.600  +memory used for DFS stack (-m700000)
7.551   +memory used for other data structures
43.852  =total actual memory usage

Output from 'time':
        Command being timed: "./brp-prot.pan -c1 -w26 -m700000"
        User time (seconds): 92.18
        System time (seconds): 1.09
```

C.6 Optimized BRP Service

From Section C.4.3 we learned that even the model of the BRP Service results in a considerable state space. Much of this state space is caused by the way we generate *random* files (see Section C.3).

In this Section we describe a more efficient way to fill the files with *random* chunks. Naturally, we will also adopt this scheme in Section C.7, where we discuss an optimized version of the BRP Protocol.

55 ⟨*brp-serv-opt.m4* 55⟩≡
  ⟨*prelude* 32.1⟩
  ⟨*Optimized Environment process* 56.1⟩
  ⟨*Service process* 42.1⟩
  ⟨*Service init* 44.2⟩

The only thing changed with respect to our original service description of the BRP, is the ⟨*Optimized Environment process*⟩.

56.1 ⟨*Optimized Environment process* 56.1⟩≡ (55 58)

```
proctype Environment()
{
  ⟨Environment: locals 35.1⟩

  do
  ::  Sin!REQ -> atomic {
        ⟨Check previous file 39.1⟩
        ⟨Optimized Generate new file 56.2⟩
      }
      Sin!REQ

  ::  Sout?sInd

  ::  Rout?i(v) -> d_step {
        k++ ;
        e[k].ind = i ;
        e[k].val = v ;
      }
  od
}
```

And this ⟨*Optimized Environment process*⟩ only differs in the way a new file is generated.

56.2 ⟨*Optimized Generate new file* 56.2⟩≡ (56.1)

```
⟨Reset array d 57.1⟩
⟨Random n 36.2⟩
⟨Place random 1 57.2⟩
k = 0 ;
```

The general correctness property for a data transfer protocol is that if the protocol receives an infinite sequence of distinct messages, it outputs the same infinite sequence. Pierre Wolper [28] has shown that if the protocol is data-independent, checking an infinite sequence of distinct messages can be reduced to checking an infinite sequence of only three distinct messages (see also [18]).

To minimize the number of different "files" to be checked, we have adopted a scheme similar to [18]. Instead of using three 'colors', we use only two: `0` and `1`. After randomly generating the number of chunks of the file (i.e. `n`), the array `d` is filled with `0`s and a single `1` is randomly placed in `d[1..n]`. Because this `1` is randomly placed, SPIN will catch the following errors in a full state space search:

- *losing a chunk.* If the BRP can lose a message, there will be an `e[1..n]` which does not contain a `1`.
- *duplication of a chunk.* If the BRP can duplicate a chunk, there will be an `e[1..n]` which contains two `1`s.

- *reordering of chunks.* If the BRP can change the order of transmitted chunks, the `1` in `d[1..n]` will end up in a different place in `e[1..n]`.

In all these cases, the array of chunks sent (i.e. `d[1..n]`) is different from the array of chunks received (i.e. `e[1..n]`).

Below, the realization of this scheme in PROMELA is presented.

57.1 ⟨*Reset array d* 57.1⟩≡ (56.2)

```
forloop('i', 1, eval(MAX_CHUNKS-1), 'd[i]=0 ; ')
```

This just sets all elements of the array `d[1..MAX_CHUNKS-1]` to `0`.

57.2 ⟨*Place random 1* 57.2⟩≡ (56.2)

```
if forloop('i', 1, eval(MAX_CHUNKS-1), '
:: n==i ->
     if forloop('j', 1, i, '
     :: d[j] = 1')
     fi
')fi ;
```

In ⟨*Place random 1* ⟩, a single `1` is randomly placed into the array `d[1..n]`.

Intermezzo It should be noted that we have based our verification on two intuitive and informal ideas, which should be proved to conclude that our models of the BRP service and protocol are *really* correct for all possible files of chunks. These unproved, informal ideas are the following:

- It is enough to check files with a maximum length of `3` to conclude that the protocol holds for all files.
- Checking files that consist of `0`s and a randomly placed `1` is sufficient to catch all errors caused by loss, duplication and reordering.

In other words, the optimized versions cannot verify completely the correctness of the service or the protocol. However, they are quite helpful in increasing the confidence of the verification procedure.

C.6.1 Validation results

Directives file Part of the directives file has been presented in Section C.4.3. For the optimized service specification of the BRP, the following directives have been used:

57.3 ⟨*brp-directives.dat* 45.1⟩+≡ ◁54.3 60▷

```
brp-serv-opt
-DMEMCNT=22
-w15 -m10000
```

The `MEMCNT` directive limits the amount of memory to be used by the `pan` analyser to only `2^22` bytes (= 4Mb) of memory. The `-w15` option reserves `2^15` (= 32K) entries in the hashtable and the `-m10000` option limits the search depth of the `pan` analyser to `10000` steps.

Results Running the `pan` analyser produces the following (stripped) output:

```
(Spin Version 2.9.7 -- 18 April 1997)  [run on 17-June-97 12:31:55]

State-vector 56 byte, depth reached 9329, errors: 0
   20519 states, stored
    4842 states, matched
   25361 transitions (= stored+matched)
   16330 atomic steps
hash conflicts: 3264 (resolved)
(max size 2^15 states)

Stats on memory usage (in Megabytes):
1.313   equivalent memory usage for states (stored*(State-vector + overhead))
        compressed State-vector = 40 byte + 8 byte overhead
0.978   actual memory usage for states (compression: 74.45%)
0.131   +memory used for hash-table (-w15)
0.240   +memory used for DFS stack (-m10000)
0.237   +memory used for other data structures
1.498   =total actual memory usage

Output from 'time':
        Command being timed: "./brp-serv-opt.pan -c1 -w15 -m10000"
        User time (seconds): 0.81
        System time (seconds): 0.02
```

C.7 Optimized BRP Protocol

Naturally, for the optimized BRP protocol we use the same ⟨*Optimized Environment process*⟩ as developed in section C.6.

58 ⟨*brp-prot-opt.m4* 58⟩≡
  ⟨*prelude* 32.1⟩
  ⟨*Optimized Environment process* 56.1⟩
  ⟨*Optimized Protocol prelude* 59.1⟩
  ⟨*Sender process* 48.4⟩
  ⟨*Receiver process* 51.1⟩
  ⟨*Daemon process* 59.4⟩
  ⟨*Optimized Protocol init* 59.5⟩

In the optimized BRP protocol, we acknowledge the fact that the processes `Line_K` and `Line_L` are just `1`-place buffers. So these *processes* will be replaced by PROMELA channels of length `1`.

59.1 ⟨*Optimized Protocol prelude* 59.1⟩≡ (58)
⟨*Protocol constants* 49.2⟩
⟨*Optimized Protocol channels* 59.2⟩

59.2 ⟨*Optimized Protocol channels* 59.2⟩≡ (59.1) 59.3▷
```
chan ChunkTimeout = [0] of {bit} ;
chan SyncWait     = [0] of {bit} ;
```

The channels `ChunkTimeout` and `SyncWait` are the same channels as discussed in Section C.5.

59.3 ⟨*Optimized Protocol channels* 59.2⟩+≡ (59.1) ◁59.2
```
chan K  = [1] of {bit, bit, bit, byte} ;
chan L  = [1] of {bit} ;
chan F  = K ;
chan G  = K ;
chan A  = L ;
chan B  = L ;
```

By making the channels `F`, `G`, `A` and `B` reference the `1`-place bufffers `K` and `L` we can retain the original processes `Sender` and `Receiver`.

The original processes `Line_L` and `Line_K` also contained behaviour to model the lose of messages. Now that the channels `K` and `L` are used, we need an auxiliary process which "steals" messages from these lines:

59.4 ⟨*Daemon process* 59.4⟩≡ (58)
```
proctype Daemon()
{
  bit   b, b1, bN, ab ;
  byte  v ;

  do
  ::  atomic { K?b1,bN,ab,v -> ChunkTimeout!SHAKE }
  ::  atomic { L?b          -> ChunkTimeout!SHAKE }
  od
}
```

When the `Daemon` process steals a message from `K` or `L` the `Sender` process is notified via channel `ChunkTimeout`. Finally, the `init` process of the optimized BRP protocol is the following:

59.5 ⟨*Optimized Protocol init* 59.5⟩≡ (58)
```
init {
  atomic {
    run Environment() ;
    run Sender() ;
    run Receiver() ;
    run Daemon() ;
  }
}
```

C.7.1 Validation results

Directives file Part of the directives file has been presented in Section C.4.3. For the optimized protocol specification of the BRP, the following directives have been used:

60 ⟨*brp-directives.dat* 45.1⟩+≡ ◁57.3

```
brp-prot-opt
-DMEMCNT=25 -DCOLLAPSE
-w20 -m300000
```

The `MEMCNT=25` directive limits the total amount of memory to be used by the `pan` analyser to `2^25` bytes (= 32Mb). The `COLLAPSE` directive instructs the analyser to compress the state vector size. The `-w20` option reserves `2^20` (= 1M) entries in the hashtable. The `-m300000` option instructs the `pan` analyser to limit the search depth to `300000` steps.

Results Running the `pan` analyser produces the following (stripped) output:

```
(Spin Version 2.9.7 -- 18 April 1997)  [run on 17-June-97 12:32:09]

State-vector 92 byte, depth reached 257647, errors: 0
  846531 states, stored
  295259 states, matched
1.14179e+06 transitions (= stored+matched)
  497354 atomic steps
hash conflicts: 245680 (resolved)
(max size 2^20 states)

Stats on memory usage (in Megabytes):
88.039  equivalent memory usage for states (stored*(State-vector + overhead))
        compressed State-vector = 8 byte + 12 byte overhead
17.020  actual memory usage for states (compression: 19.33%)
4.194   +memory used for hash-table (-w20)
7.200   +memory used for DFS stack (-m300000)
3.753   +memory used for other data structures
32.079  =total actual memory usage

Output from 'time':
        Command being timed: "./brp-prot-opt.pan -c1 -w20 -m300000"
        User time (seconds): 95.40
        System time (seconds): 0.92
```

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, 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, *Computer Aided Verification, 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 R. Alur, T. Henzinger and E.D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 232–243. Springer-Verlag, 1996.

[5] J. Bengtsson, F. Larsson, P. Pettersson, W. Yi, P. Christensen, J. Jensen, P. Jensen, K.G. Larsen, and T. Sorensen. UPPAAL a tool suite for validation and verification of real-time systems (Preliminary draft). User guide, 1996. Available at `http://www.docs.uu.se/docs/rtmv/uppaal`.

[6] Z. Brezocnik and T. Kapus, editors. *Proceedings of COST 247, Int. Workshop on Applied Formal Methods in System Design.* Technical Report, University of Maribor, Slovenia, 1996.

[7] P.R. D'Argenio, J-P. Katoen, T. Ruys, and J. Tretmans. Modeling and verifying a Bounded Retransmission Protocol. In [6]. Also available as CTIT Technical Report 96-22.

[8] P.R. D'Argenio, J-P. Katoen, T. Ruys, and J. Tretmans. The Bounded Retransmission Protocol must be on time!. In E. Brinskma, editor, *Third Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, LNCS 1217, pages 416–431. Springer-Verlag, 1997.

[9] C. Daws, A. Olivero, S. Tripakis and S. Yovine. The tool KRONOS. In R. Alur, T. Henzinger and E.D. Sontag, editors, *Hybrid Systems III*, LNCS 1066, pages 208–219. Springer-Verlag, 1996.

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

[11] J.F. Groote. Specification and verification of real time systems in ACP In L. Logrippo, R.L. Probert and H. Ural, editors, *Protocol Specification, Testing and Verification X*, Ottawa, Canada, pages 261–274. North-Holland, 1990.

[12] 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.

[13] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Report CS-R9076, CWI, Amsterdam, The Netherlands, 1990.

[14] 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*, LNCS 1051, pages 662–681. Springer-Verlag, 1996.

[15] 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.

[16] T.H. Henzinger, P.-H. Ho, and H. Wong-Toi. A user guide to HyTech. In E. Brinksma et. al, editors, *First Int. Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'95)*, LNCS 1019, pages 41–71. Springer-Verlag, 1995.

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

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

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

[20] ITU-T. Recommendation Z.120: Message Sequence Chart (MSC). ITU - Telecommunication Standardization Sector, Geneva, Switzerland, 1996.

[21] H.E. Jensen, K.G. Larsen, and A. Skou. Modelling and analysis of a collision avoidance protocol using Spin and Uppaal. BRICS Report Series RS-96-24, 1996.

[22] D.E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1994.

[23] R. Mateescu. Formal description and analysis of a bounded retransmission protocol. In [6]. Also available as Technical Report no. 2965, INRIA, 1996.

[24] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In K.G. Larsen and A. Skou, editors, *Computer Aided Verification, CAV'91*, LNCS 575, pages 376–398. Springer-Verlag, 1991.

[25] N. Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994. Noweb is available at `http://www.cs.virginia.edu/~nr/noweb/`

[26] N. Shankar. Unifying verification paradigms. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, LNCS 1135, pages 22–40. Springer-Verlag, 1996.

[27] 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)*, LNCS 1055, pages 329–348. Springer-Verlag, 1996.

[28] P. Wolper. Specifying interesting properties of programs in propositional temporal logic. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages (POPL'86)*, pages 184–193, St. Petersburg Beach, Florida, January 1986. ACM.