Exploring Properties of a Bounded Retransmission Protocol with VIS

Robert Meolic, Tatjana Kapus and Zmago Brezočnik
Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia

It is of great interest for users of communication protocols to have a proof that they are correct and reliable to use. In order to prove a protocol correct formally, the protocol and the required properties must be formally described. This paper reports on verifying properties of a bounded retransmission protocol for large data packets. It is used to transfer files via a lossy communication channel. We emphasize timing properties of the protocol. We specified the protocol in Verilog and stated properties in a computation tree logic. The verification was carried out automatically by model checking. We used the non-commercial tool VIS which made it possible to introduce nondeterministic choice of data packets length and a realistic notion of time.

Keywords: formal verification, bounded retransmission protocol, model checking, CTL, Verilog, VIS

1 Introduction

Verifying communication protocols is known to be a difficult challenge for verification technologies. Some protocols are simple enough to be proved correct manually. However, there are many reasons, why such proofs cannot be carried out for most of them. One reason is the presence of real-time aspects, which complicates the proof significantly.

A simple well-known protocol for transferring data reliably over an unreliable data channel is the alternating bit protocol. It is widely used as a theoretical example (e.g. [Mil89]). Its drawback is the absence of a mechanism to stop retrying to transfer a datum to the receiving side of the channel if the number of retransmissions exceeds a chosen number due to successive corruptions. One nontrivial protocol extension of the alternating bit protocol, which uses timeouts and aborts transmission following a bounded number of retransmission attempts, is the bounded retransmission protocol (BRP), introduced by Philips [Dam97].

The BRP has been developed for the 4^{th} generation infra-red remote control systems. It has already been studied using different approaches. We know about the verification using the CADP toolbox [Mat96], a tool for checking safety properties of lossy channel systems [AK96], Uppaal and SPIN [Dea96, Dam97], Murϕ and PVS [HS96], μCRL and Coq [GdP96], I/O-automata and Coq [Hea94], and automatic abstraction using PVS [GS97]. Only [Dea96] and [Dam97] consider real-time aspects of the protocol. All the other case studies use tricks and assumptions about proper timing. Our paper reports on specifying BRP in Verilog and verifying it with VIS, which turned out to be a capable tool compared to some others.

Verilog is a powerful hardware description language for the description, verification, simulation, and synthesis of electronic circuits [Vea96]. It is a public domain, C-based language, and it is an IEEE standard since 1995. To specify our communication protocol we needed only a small subset of its capability.

VIS (Verification Interacting with Synthesis) is a powerful BDD-based tool for verification, synthesis, and simulation of finite-state systems [VIS, Vea96]. It has been developed jointly at Berkeley, Boulder, and Austin. VIS operates on an intermediate low-level language BLIF-MV, which allows to describe hierarchical symbolic sequential systems with nondeterminism. A very good compiler from Verilog to BLIF-MV is included [Che94].

Work supported by Ministry of Science and Technology, Republic of Slovenia.

Probably the most efficient and popular approach to automatic formal verification is model checking. VIS enables model checking using Fair CTL [Vea96]. CTL (Computation Tree Logic) is a propositional temporal logic of branching time. It is used to describe properties of systems. If model checking fails, VIS can report the failure with a counterexample.

Further in this paper we first introduce the BRP service and BRP protocol. In section 3 we present a model of the BRP in Verilog. In section 4 we describe formal verification using CTL formulas and in section 5 we discuss timing properties of the BRP. We conclude with an evaluation of our work.

2 The Bounded Retransmission Protocol

2.1 Service specification

A producer successively wants to deliver large data packets (e.g. files) to a consumer through a lossy channel. Each data packet consists of an arbitrary, nonzero number of small chunks. The chunks of a data packet are delivered to the consumer in sequence. There is a limited amount of time available for delivering each chunk to the consumer. If the time passes for a chunk, delivery of the data packet stops. So, for each data packet, either the complete data packet or just a prefix thereof is delivered to the consumer, and then delivery of the next data packet starts. Both the producer and the consumer are to be notified whether the complete data packet has been delivered or not.

For each data packet sent, the producer gets one of the following three confirmations. If it is sure that the complete packet has been delivered to the consumer, the producer gets a confirmation I_{OK}. Unsuccessful delivery is notified by $\mathsf{I}_{\mathsf{NOK}}$. It gets I_{DK} if the last chunk has been sent but it is not sure if it has been delivered to the consumer.

For each data packet being delivered, each chunk received by the consumer is accompanied by an indication. The indication is $\mathsf{I}_{\mathsf{FST}}$ if the chunk is the first but not the last one of the data packet. It is $\mathsf{I}_{\mathsf{INC}}$ if the chunk is an intermediate one, and I_{OK} if it is the last one of the data packet. In the case delivery has been aborted after delivering at least one chunk of the data packet to the consumer, it gets an indication $\mathsf{I}_{\mathsf{NOK}}$.

2.2 Protocol specification

Our description of the BRP is similar to that from [Dea96]. The protocol consists of $Producer$, $Consumer$, $Sender$, $Receiver$, $Channel_K$, $Channel_L$, and three timers (Figure 1). $Sender$ gets the data packets from $Producer$ and sends them as a sequence of chunks through lossy $Channel_K$. $Sender$ accompanies each chunk of a data packet by an indication $\mathsf{I}_{\mathsf{FST}}$, $\mathsf{I}_{\mathsf{INC}}$, or I_{OK}. $\mathsf{I}_{\mathsf{FST}}$ is used for the first but not the last chunk of the data packet. I_{OK} is used for the last chunk of the data packet. All other chunks are accompanied by $\mathsf{I}_{\mathsf{INC}}$. $Receiver$ passes received chunks together with the accompanied indication to $Consumer$. It also acknowledges every received chunk through $Channel_L$. The chunks and the acknowledgements can be lost. After all the chunks have been properly acknowledged, $Sender$ notifies $Producer$ by confirmation I_{OK} and waits for a new data packet.

$Sender$ sends one chunk at a time. It starts $Timer_X$ immediately after sending a chunk and waits for an acknowledgement from $Receiver$ or a timeout of $Timer_X$. If the acknowledgement does not arrive in time, $Timer_X$ times out and $Sender$ sends the chunk again, but only if the time limit for sending one chunk is not reached. The time limit is in fact represented by a maximum number of retransmissions allowed.

If the number of unsuccessful retransmissions reaches the maximum number, $Sender$ aborts transfer of the data packet currently being sent. $Sender$ notifies $Producer$ about abortion of delivery by confirmation $\mathsf{I}_{\mathsf{NOK}}$ or I_{DK}. The latter is used if there was an error during the delivery of the last chunk. After the abortion, $Sender$ also starts $Timer_{SYNC}$ and waits for its timeout before getting ready to deliver a new data packet. This is needed because $Sender$ must wait until $Receiver$ properly reacts to abortion.

The protocol uses an alternating bit as a part of every chunk sent in order to detect duplicates of an already received chunk at $Receiver$. If successive received chunks of the same data packet have equal alternating bits, $Receiver$ assumes such a chunk has already been received and does not pass it to $Consumer$. However, $Receiver$ acknowledges every chunk.

$Receiver$ starts $Timer_Z$ every time it receives a *new* chunk. If $Timer_Z$ times out, it means that

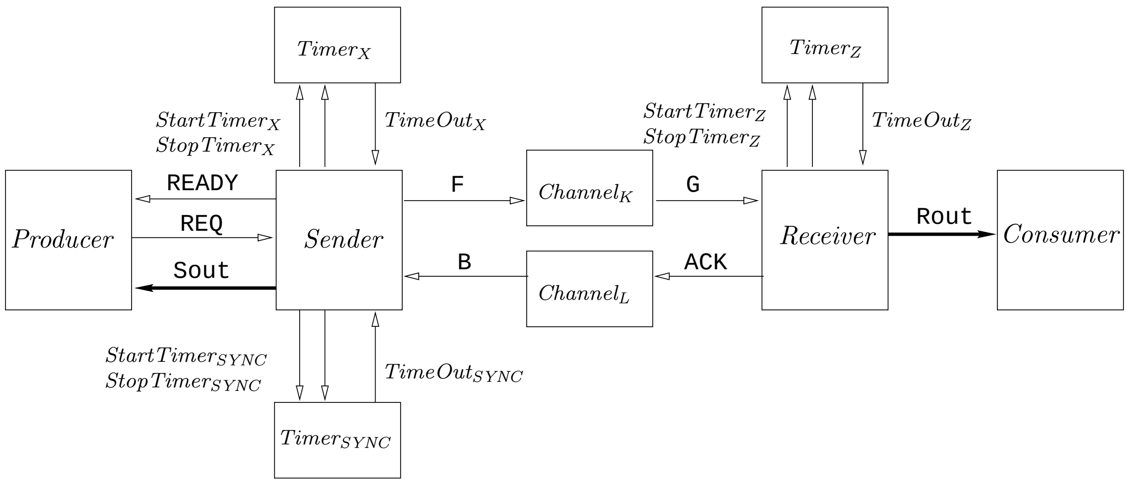

Figure 1: A shematic view of the BRP specification

no new chunk has arrived for a long time and therefore, $Receiver$ may be sure that $Sender$ has aborted the delivery. $Receiver$ notifies $Consumer$ about the abortion of delivery by $\mathsf{I}_{\mathsf{NOK}}$, unless the last received chunk has been the last one of the data packet, and returns to its initial state.

3 Description in Verilog

3.1 Introductory remarks

A description in Verilog consists of one or more modules. The structure of a model is described by making instances of modules. All module instances run in parallel and are synchronized by an implicit clock. The interval between two clock signals is called *cycle*.

Modules have registers where data are stored. The default range of the data is one bit, but it can also be specified as vectors of n bits. VIS extends the standard Verilog to allow symbolic data using enumerated type, too. To improve readability of the model, macro definitions can be used to introduce constants.

Modules communicate with each other through wires. Each register in a module may drive one wire. By changing the value of the register the value of the wire is changed. The name of the wire is equal to the name of the register. The communication through the wires is instantaneous. A wire does not store values and has to be continuously driven. Only synchronous communication can be modeled with wires. There is a risk that a value disappears before it is used. Our model ensures that the values will always be caught with no need of it being present for more than 1 cycle.

The behaviour of a module is described using an `initial` block and an `always` block. The `initial` block gives the registers initial values. The `always` block contains assignments, decisions, and other statements. The statements are sequentially executed on each clock signal. Outside the `initial` and `always` blocks we may have `assign` statements that describe the combinational part of the module. They are also called *continuous assignments* as they determine wires which are continuously assigned the same boolean function of registers and other wires.

In our Verilog description of the BRP, modules correspond to entities of the protocol. Each entity is represented as a finite state machine (FSM), therefore all modules have a register named `state`, where the current state of the FSM is stored.

We say that the modules representing the BRP entities communicate by exchanging signals, indications, and confirmations. The presence of a signal on a wire is represented by value YES. Value NO means that there is no signal on the wire. Wires for carrying indications and confirmations are driven with values $\mathsf{I}_{\mathsf{FST}}$, I_{OK} etc. The value Z means that there is no indication or confirmation present.

3.2 The BRP model

Our Verilog description of the BRP is about 400 lines long including some comments. In order to explain the structure of modules we provide diagrams similar to process diagrams of SDL (e.g. [Bra96]). Only the `channel` module is given directly in Verilog due to different expression of nondeterminism in Verilog and SDL. The presented `channel` module serves also as an example of a description in Verilog.

The `channel` module (Figure 2) has registers `state` and `OUT`. The module communicates with other modules through wires `IN` and `OUT`, which carry signals. Wire `OUT` is driven by the `channel` module itself and is used to send signals to other modules. Wire `IN` is driven outside the `channel` module and is therefore used to get signals from other modules. The `channel` module must nondeterministically decide whether to lose a signal. In Verilog, this can be achieved by using a generator of random values. So, the `channel` module has a 2-bit wire `rand`, which is continuously driven by such a generator (construct `$ND`). The generator chooses a value between 0 and 3. The meaning of the chosen value depends on constant `SUCCESS` which is introduced as a macro. Changing this constant we can make the channel more or less reliable as it is explained in the continuation.

$\mathbf{Channel_K}$ and $\mathbf{Channel_L}$ are both instances of the `channel` module. Initially, both channels are in the `START` state with register `OUT` set to `NO`. If wire `IN` carries value `YES`, the channel goes to the `TRANSFER` state. In this state the channel checks the value of wire `rand`. If it is smaller than constant `SUCCESS`, the channel “decides” not to lose the input signal and sets register `OUT` to `YES`. Otherwise the value of register `OUT` remains `NO`. Then the channel returns to the `START` state. The assignment `OUT = NO` in the beginning of the `always` block causes that the signal `OUT` is present for only one cycle.

The `Producer` module (Figure 3) produces data packets. The number of chunks in each data packet is randomly chosen. It is stored in register `n` during the whole delivery of the packet. Notice that the contents of chunks is not modelled. Only the delivery request and the number of chunks in the data packet are transferred from $Producer$ to $Sender$.

```
‘define SUCCESS 2 //macro definition
typedef
 enum {YES, NO} boolean;
typedef
 enum {START,TRANSFER} channel_state;

module channel(clk,IN,OUT);
  input clk;
  input IN;   //input signal
  output OUT; //output signal

  channel_state reg state;
  boolean reg OUT;
  boolean wire IN;
  wire [2:1] rand; //2-bits vector

  initial begin
    state = START;
    OUT = NO;
  end
  assign rand = $ND(0,1,2,3);

  always @(posedge clk) begin
    OUT = NO; //reset output signal
    case (state)
     START: begin
       if (IN==YES) state = TRANSFER;
     end
     TRANSFER: begin
       if (rand<‘SUCCESS) OUT = YES;
       state = START;
     end
    endcase
  end
endmodule
```

Figure 2: Specification of a lossy channel

The `Sender` module (Figure 4) starts in the `IDLE` state. It notifies $Producer$ that it is ready for the transmission of a new data packet by setting signal `READY`. $Producer$ demands delivery of a data packet by sending signal `REQ`. When $Sender$ gets signal `REQ`, it stores the number of chunks in the packet to register `n`. Then it resets the counter of chunks (`i = 1`) and the retransmission counter (`rc = 1`) and starts delivery of the first chunk. $Sender$ first writes a proper indication and alternating bit to registers `rind` and `rab`, respectively. It then sends the chunk to $Channel_K$ by setting signal F. At the same time it starts $Timer_X$. $Sender$ holds registers `rind` and `rab` unchanged during the whole delivery of the chunk.

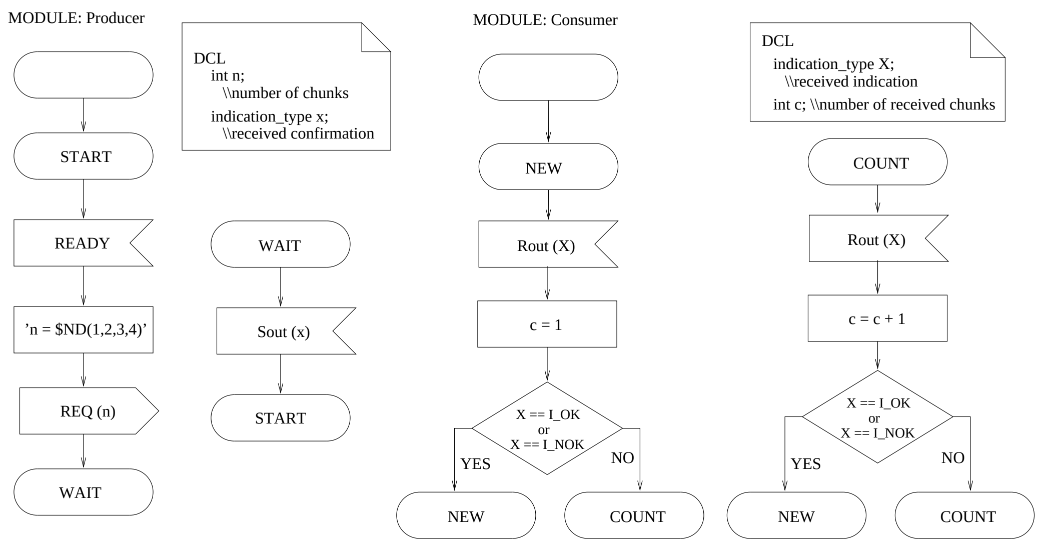

Figure 3: The `Producer` and `Consumer` modules

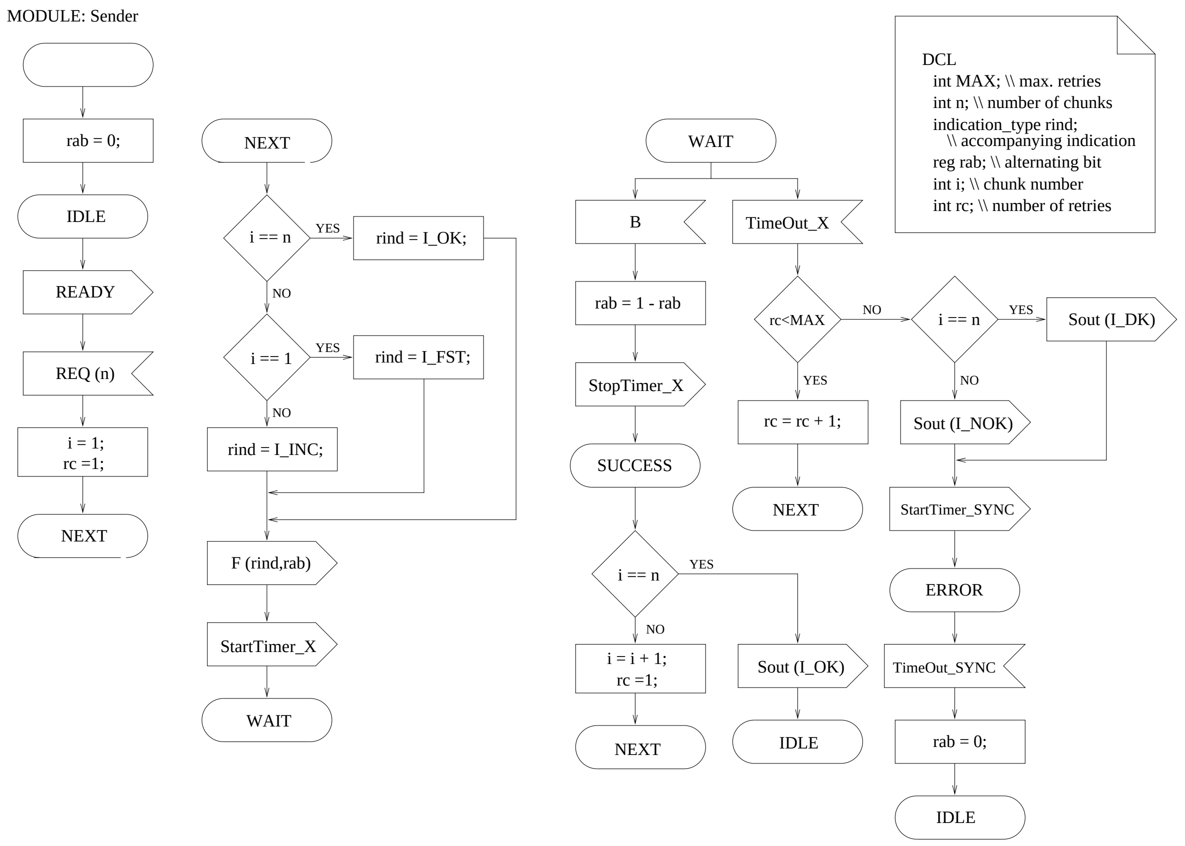

Figure 4: The `Sender` module

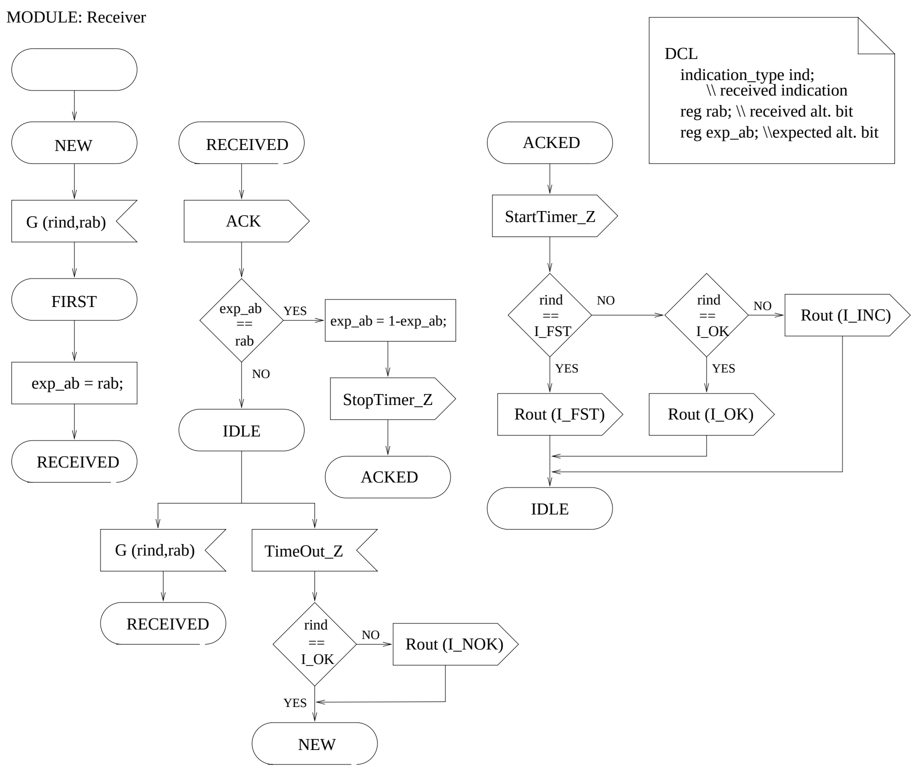

Figure 5: The `Receiver` module

If the chunk is not lost, $Channel_K$ sends signal G to the `Receiver` module (Figure 5). To make the model more effective we in fact do not send the accompanying indication and alternating bit through $Channel_K$. Instead, $Receiver$ looks at the values of registers `rind` and `rab` of $Sender$ through supplementary wires at the moment it gets signal G.

$Receiver$ acknowledges every chunk through $Channel_L$ by setting signal `ACK`. If a new chunk has arrived, $Receiver$ stops and restarts $Timer_Z$, and sends the received indication to the `Consumer` module. $Receiver$ then waits for new chunks in the `IDLE` state. If signal $TimeOut_Z$ appears, when $Receiver$ is waiting for a new chunk, it returns to the `NEW` state. If the awaited chunk is not the first of a packet, it sends indication $\mathsf{I}_{\mathsf{NOK}}$ to $Consumer$ before. In the `NEW` state $Receiver$ accepts every chunk as a new one regardless of its alternating bit.

$Sender$ is waiting for an acknowledgement on wire `B` or signal $TimeOut_X$. If it receives an acknowledgement, it stops $Timer_X$. If it has just sent the last chunk of the packet, it notifies $Producer$ about successful delivery by confirmation I_{OK} through wire `Sout`. If there are still some chunks for delivery, $Sender$ increments the counter of chunks, resets the retransmission counter and starts transfer of the next chunk.

If signal $TimeOut_X$ appears when $Sender$ is waiting for an acknowledgement, it aborts delivery of the data packet or sends the chunk again. In the case of abortion $Sender$ notifies $Producer$ about unsuccessful delivery by confirmation $\mathsf{I}_{\mathsf{NOK}}$ through wire `Sout`. If the chunk is to be sent again, $Sender$ increments the retransmission counter and sends signal F.

4 Formal verification using CTL

CTL formulas are built from atomic propositions, standard boolean operators, and temporal operators [Cea86]. Atomic propositions express state properties of the system (e.g. "signal `REQ` is set to `YES`"). Temporal operators describe qualitatively, when these properties have to be satisfied.

Each temporal operator consists of a path quantifier and a temporal modality. The *path quantifier* **A** indicates that a property is true "for all computations" and **E** denotes that it is true "for some computation". The *temporal modality* describes the ordering of events in time along a computation and can be one of the following symbols: **F** ("in a future (possibly current) state"), **G** ("globally", "in every state"), **X** ("in the next state") or **U** (strong "until").

With CTL formulas one can easily express properties like: "p is valid in every state of every computation" or shortly "p is always valid" (**AG** p), "p is never valid" (!**EF** p), "it is always possible for p to be valid" (**AG EF** p), "p is valid infinitely often" (**AG AF** p), and many others. There is also a very useful CTL template to express safety properties:

$$\mathbf{AG}(s -> \ \mathbf{AX}\ \mathbf{A}\ (!p\ \mathbf{U}\ q)) \qquad (1)$$

Informally, formula (1) says that always, if s is true at a state, then q is true in a future state (not in the current one) and p is not true at some state after s and before q. We use the common textual notation for boolean operators, ! for negation, $*$ for conjunction, $+$ for disjunction, and $->$ for implication, as in VIS and many other tools.

For our model of the BRP we verified many different CTL formulas. The most significant are presented in Figure 6. The notation of our CTL formulas is very compact, especially due to VIS, which allows the macro definitions of subformulas.

The first formula in Figure 6 expresses a general liveness property. $Producer$ produces data packets and $Sender$ starts delivery of them infinitely often. This assures that the validity of other properties of the model is not the consequence of staying in a deadlock state.

Formulas from 2 to including 7 talk about the order of indications received for a data packet by $Consumer$. There are four possible indications: $\mathsf{I}_{\mathsf{FST}}$, $\mathsf{I}_{\mathsf{INC}}$, I_{OK}, and $\mathsf{I}_{\mathsf{NOK}}$. The first indication received is $\mathsf{I}_{\mathsf{FST}}$ or I_{OK} (formula 2). It is $\mathsf{I}_{\mathsf{FST}}$ if a data packet with more than one chunk is being delivered (formula 4). The last indication received is I_{OK} or $\mathsf{I}_{\mathsf{NOK}}$ (formulas 5,7). If Consumer gets I_{OK}, all chunks of the data packet were delivered (formula 6). All the other indications are $\mathsf{I}_{\mathsf{INC}}$ as they can neither be $\mathsf{I}_{\mathsf{FST}}$ nor I_{OK} and $\mathsf{I}_{\mathsf{NOK}}$ (formula 3,5).

Formulas 8 and 9 require that $Producer$ has to be informed about success of data packet delivery with exactly one indication before starting delivery of a new one. Formula 10 talks about coincidence of indications got by $Consumer$ and confirmations got by $Producer$. Because we did not want to fix their ordering, we express the property with 4 CTL formulas.

Formula 11 says that $Producer$ does not get $\mathsf{I}_{\mathsf{NOK}}$ in the case of transmission error if the data packet being sent contains only one chunk. On the other hand, $Producer$ does not get I_{DK} if the data packet being sent has more than one chunk and no chunk has successfuly been delivered (formula 12).

The last three formulas are stated to discover errors due to the chosen timeout intervals. These formulas refer more to the protocol realisation than to its properties. To make these formulas possible we introduced supplemental signals $ChunkLost$ and $PacketLost$ into the `channel` module. If $\mathbf{Channel}_K$ or $\mathbf{Channel}_L$ loses the input signal, it sends signal $ChunkLost$. If this happens during the last allowed chunk retransmission, it sends signal $PacketLost$, too.

All CTL formulas from Figure 6 together were verified with VIS release 1.2 on a HP 715/100 Workstation with 128 MB of RAM in 17 minutes of CPU time. The allowed number of transmissions MAX was set to 2 (i.e. the maximum number of retransmissions was MAX - 1 = 1). Data packets were of different length. The maximum number of chunks in a packet was 4. We set the smallest timeout intervals possible for our model: $Time_X = 9$, $Time_Z = 34$, and $Time_{SYNC} = 32$ (see the next section). Constant SUCCESS was set to 2, which caused that 50% of delivered chunks and acknowledgements were lost.

5 Timing properties of the BRP

The BRP is a time-sensitive protocol due to the presence of the timers. To ensure its proper functioning, the timeout intervals must be chosen carefully in order to prevent premature timeouts.

There are three timers in our model: $Timer_X$, $Timer_Z$, and $Timer_{SYNC}$. Verilog allows us to model explicit timers which count the number of cycles (clock signals). Each timer is controlled by two wires, $StartTimer$ and $StopTimer$, and sends timeout signals on wire

# Macro definitions (a macro starts with a backslash). The absent definitions are similar to these.

```
\define Rout !(receiver.Rout = Z)
\define Rout_I_FST (receiver.Rout = I_FST)
\define Chunk_I_FST (sender.rind = I_FST)
\define StartPacket ((sender.READY = YES) * (producer.REQ = YES))
\define ShortPacket (producer.n[4:1] = 1)
\define ChunkLost ((channel_K.ChunkLost = YES) +
                   (channel_L.ChunkLost = YES))
\define PacketLost ((channel_K.PacketLost = YES) +
                    (channel_L.PacketLost = YES))
\define StartTimer_x (sender.StartTimer_x = YES)
\define TimeOut_x (timer_x.TimeOut = YES)
\define ReceiverReady (receiver.state = NEW)
\define PacketComplete (((consumer.c<1>=1)<->(producer.n<1>=1)) *
                        ((consumer.c<2>=1)<->(producer.n<2>=1)) *
                        ((consumer.c<3>=1)<->(producer.n<3>=1)) *
                        ((consumer.c<4>=1)<->(producer.n<4>=1)))
```

# 1. The data packets are delivered infinitely often.

```
AG AF \StartPacket;
```

# 2. The first indication received for a data packet is I_{FST} or I_{OK}.

```
AG(\StartPacket -> AX A(!\Rout U (\Rout_I_FST + \Rout_I_OK + \StartPacket)));
```

# 3. During the delivery of a data packet at most one received indication is I_{FST}.

```
AG(\Rout_I_FST -> AX A(!\Rout_I_FST U (!\Rout_I_FST * \StartPacket)));
```

# 4. When delivering a data packet with more than one chunk, the first received indication is I_{FST}.

```
AG((\StartPacket * !\ShortPacket) -> AX A(!\Rout U (\Rout_I_FST + \StartPacket)));
```

5. I_{OK} (I_{NOK}, respectively) is the last indication received for a data packet.

```
AG((\Rout_I_OK + \Rout_I_NOK) -> AX A(!\Rout U (!\Rout * \StartPacket)));
```

# 6. If Consumer gets I_{OK}, all chunks are delivered.

```
AG(\Rout_I_OK -> AX \PacketComplete);
```

7. I_{FST} (I_{INC}, respectively) is not the last indication received for a data packet.

```
AG((\Rout_I_FST + \Rout_I_INC) -> AX A(!\StartPacket U \Rout));
```

# 8. Producer does not start delivery of a new data packet until it gets a confirmation.

```
AG(\StartPacket -> AX A(!\StartPacket U \Sout));
```

# 9. After receiving a confirmation, Producer does not get another one for the same data packet.

```
AG(\Sout -> AX A(!\Sout U (!\Sout * \StartPacket)));
```

# 10. Producer does not get I_{NOK} for the data packet if Consumer gets I_{OK}, and vice versa.

```
AG(\Rout_I_OK -> A(!\Sout_I_NOK U (!\Sout_I_NOK * \StartPacket)));
AG(\Sout_I_NOK -> A(!\Rout_I_OK U (!\Rout_I_OK * \StartPacket)));
AG(\Rout_I_NOK -> A(!\Sout_I_OK U (!\Sout_I_OK * \StartPacket)));
AG(\Sout_I_OK -> A(!\Rout_I_NOK U (!\Rout_I_NOK * \StartPacket)));
```

# 11. If delivering a data packet with a single chunk, Producer does not get I_{NOK}.

```
AG((\StartPacket * \ShortPacket) ->
  AX A(!\Sout_I_NOK U (!\Sout_I_NOK * \StartPacket)));
```

# 12. If delivering a data packet with more than one chunk, Producer does not get I_{DK}
if no chunk of the packet has successfully been delivered.

```
AG((\StartPacket * !\ShortPacket) ->
  AX A(!\Sout_I_DK U (\Rout + \StartPacket * !\Sout_I_DK)));
```

13. $TimeOut_X$ appears only if the chunk is lost.

```
AG(\StartTimer_X -> !EX E(!\ChunkLost U (!\ChunkLost * \TimeOut_X)));
```

14. $TimeOut_Z$ appears only if the data packet is lost.

```
AG(\StartTimer_Z -> !EX E(!\PacketLost U (!\PacketLost * \TimeOut_Z)));
```

15. $TimeOut_{SYNC}$ appears only if Receiver is ready.

```
AG(\StartTimer_SYNC -> AX A(!\TimeOut_SYNC U \ReceiverReady));
```

Figure 6: Verified CTL formulas

$TimeOut$. When a timer is running, it is incremented by one in every cycle. Signal $StartTimer$ resets the timer to zero and then starts incrementation. Signal $StopTimer$ terminates the incrementation of the timer. When the timer reaches the given value, it stops and produces timeout signal. Let us denote timeout intervals at which $Timer_X$, $Timer_Z$, and $Timer_{SYNC}$ produce timeout signals by $Time_X$, $Time_Z$, and $Time_{SYNC}$, respectively.

$Time_X$ has to be greater than the maximum time needed for a chunk sent by $Sender$ to get to $Receiver$ and for the acknowledgement to come back if neither the chunk nor the acknowledgement is lost. By looking at the Verilog description or, respectively, the process diagrams of the modules, one can see that the round-trip time is maximal when $Receiver$ awaits the first chunk of a data packet in the NEW state. It amounts to 8 cycles. First, 2 cycles pass in $Channel_K$ (Figure 2), then 3 cycles in $Receiver$ (Figure 5), 2 cycles in $Channel_L$ (Figure 2), sand finally 1 cycle in $Sender$ (Figure 4). Therefore, the minimal value for $Time_X$ is 9. This can also be checked by simulation of the model (see Figure 7, simulation lines from 4, where $StartTimer_X$ appears, to 12, where the $StopTimer_X$ signal is present).

Determining $Time_Z$ and $Time_{SYNC}$ optimally appears to be more difficult. They depend on $Time_X$, the number of allowed retransmisions of a chunk, and the expected time interval between the end of delivery of one and start of delivery of the next data packet. Observing simulation or runs of the system one cannot be sure that these timeout intervals have correct values. It is safer to prove their correctness.

We verified $Time_X$, $Time_Z$, and $Time_{SYNC}$ with model checking (see the last three formulas in Figure 6). The listed CTL formulas allow us to determine optimal values with a method of brute force trying. In the case of a wrong combination of time constants, the model checker informs us about an error. VIS can even generate the path where the error occurs, that invalidates a CTL formula.

$Time_Z$ has to be determined before $Time_{SYNC}$. After fixing $Time_X$ to 9 we started with large values for $Time_Z$ and $Time_{SYNC}$, where all CTL formulas were valid, and decremented them until an error occurred. We found out the following optimal values: $Time_Z = 34$ and $Time_{SYNC} = 32$. The path where an error occurred for smaller $Time_Z$ is presented in Figure 8. One can see that $Timer_Z$ is stopped 33 cycles after it is started. Because $Timer_Z$ must not time out during successful delivery of a packet, it has to be at least 34. This result is valid irrespective of how unreliable the channels are.

```
> READY REQ F G ACK B
> StartTimer_x StartTimer_z
> StopTimer_x StopTimer_z
> TimeOut_x TimeOut_z
> Rout Sout
 0. * * * * * * | * * | * * | * * |   *   *
 1. Y * * * * * | * * | * * | * * |   *   *
 2. Y Y * * * * | * * | * * | * * |   *   *
 3. Y * * * * * | * * | * * | * * |   *   *
 4. * * Y * * * | Y * | * * | * * |   *   *
 5. * * * * * * | * * | * * | * * |   *   *
 6. * * * Y * * | * * | * * | * * |   *   *
 7. * * * * * * | * * | * * | * * |   *   *
 8. * * * * * * | * * | * * | * * |   *   *
 9. * * * * Y * | * * | * Y | * * |   *   *
10. * * * * * * | * Y | * * | * * | I_FST *
11. * * * * * Y | * * | * * | * * |   *   *
12. * * * * * * | * * | Y * | * * |   *   *
13. * * * * * * | * * | * * | * * |   *   *
14. * * Y * * * | Y * | * * | * * |   *   *
15. * * * * * * | * * | * * | * * |   *   *
16. * * * Y * * | * * | * * | * * |   *   *
17. * * * * * * | * * | * * | * * |   *   *
18. * * * * Y * | * * | * Y | * * |   *   *
19. * * * * * * | * Y | * * | * * | I_INC *
20. * * * * * Y | * * | * * | * * |   *   *
21. * * * * * * | * * | Y * | * * |   *   *
22. * * * * * * | * * | * * | * * |   *   *
23. * * Y * * * | Y * | * * | * * |   *   *
24. * * * * * * | * * | * * | * * |   *   *
25. * * * * * * | * * | * * | * * |   *   *
26. * * * * * * | * * | * * | * * |   *   *
27. * * * * * * | * * | * * | * * |   *   *
28. * * * * * * | * * | * * | * * |   *   *
29. * * * * * * | * * | * * | * * |   *   *
30. * * * * * * | * * | * * | * * |   *   *
31. * * * * * * | * * | * * | * * |   *   *
32. * * * * * * | * * | * * | Y * |   *   *
33. * * * * * * | * * | * * | * * |   *   *
34. * * Y * * * | Y * | * * | * * |   *   *
35. * * * * * * | * * | * * | * * |   *   *
36. * * * Y * * | * * | * * | * * |   *   *
37. * * * * * * | * * | * * | * * |   *   *
38. * * * * Y * | * * | * Y | * * |   *   *
39. * * * * * * | * Y | * * | * * | I_OK  *
40. * * * * * Y | * * | * * | * * |   *   *
41. * * * * * * | * * | Y * | * * |   *   *
42. * * * * * * | * * | * * | * * |   * I_OK
43. Y * * * * * | * * | * * | * * |   *   *
```

Figure 7: A simulation of the BRP

In [Dea96] authors claim that

$$Time_{SYNC} > Time_Z \qquad (2)$$

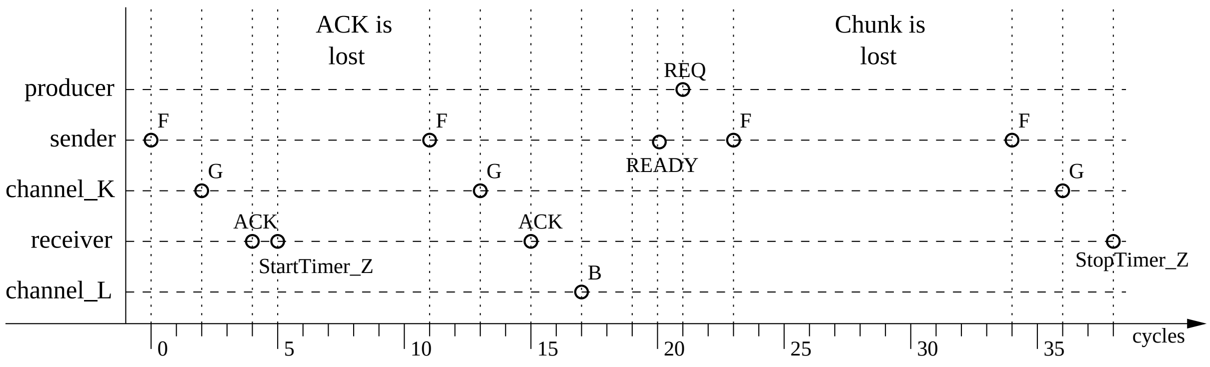

Figure 8: A path where $Time_Z$ has to be at least 34

is a necessary condition for $Time_{SYNC}$. Figure 9 shows a sequence of signals in the case of an abortion of packet delivery. $Timer_Z$ is started before $Timer_{SYNC}$. The interval between starting them (T_1) is not shorter than the time which system needs to transfer an acknowledgement from $Receiver$ to $Sender$. After $Timer_Z$ times out, $Receiver$ needs some time (T_2) to become ready for receiving a new packet. It can be seen that $Time_{SYNC} > Time_Z$ is a necessary condition only if $T_1 < T_2$. If $T_1 > T_2$ as in our model, relation (2) is a sufficient condition, but not a necessary one.

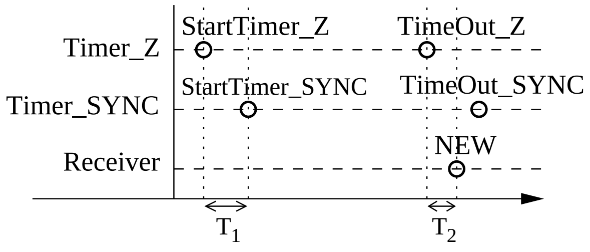

Figure 9: A sequence of signals in the case of a transmission abortion

6 Conclusion

The work demonstrated in this paper was an interesting exercise in protocol verification as the BRP is a nontrivial communication protocol with bounded number of retransmissions and real-time aspects. It is reported in [Dea96] that some tools have had problems with verifying the BRP because of too much memory consumption. VIS turned out to be a reliable and capable model checker although it is a general-purpose tool.

We started exploring properties of the BRP with a natural-language description of the service and protocol. For formal specification of the protocol we used the hardware description language Verilog. As in [Dea96] (and partly in [Mat96] and [AK96]), we eliminated the sources of infiniteness, so that automated verification with finite-state systems verification tool VIS was possible. We verified the protocol for a fixed number of retransmissions and neglected data contents of the chunks. The length of the packets was limited, but in contrast to [Dea96], [Mat96], and [AK96] nevertheless variable due to the use of generator of random values in Verilog.

We verified control properties of the protocol with model checking. Because the correctness of the protocol depends very much on the three timeout intervals, we studied timing properties of the BRP especially in detail and found optimal values for them in our model. Due to the way of modelling devices in Verilog, the delays between certain events in modules are equal to some constant number of cycles. For example, the channel delay in our model happens to be exactly 2 cycles. However, the optimal values for the timeout intervals could be found in the same manner if the delays in the channel and other modules were different. Notice that we also studied the general relation that should hold between $Time_{SYNC}$ and $Time_Z$ in more detail than this has been done in [Dea96].

We found Verilog from many points of view similar to familiar programming languages. It allows one to write readable structured specifications without an extensive study of the language itself. Its main drawback is the lack of a better support for asynchronous communication between modules.

Simulation, although it is not a formal method, is from our experiences an excellent aid to the design of models. It gave us a significant insight into the model. Simulation and the ability of VIS to produce counterexamples when model checking fails made finding and correcting errors in the model and CTL formulas much easier.

Experiences we got by exploring the properties of the BRP encourage us to explore and verify even more complex protocols in the future. We are also interested in a detailed study of VIS (e.g. FSM traversal, dynamic reordering of variables in BDDs) and in finding other problems where it may be used. Experiences with VIS will also help us to make our own model checker, which we are building, even better [BČK96].

References

[AK96] P. A. ABDULLA AND M. KINDAHL, *Using Lossy Channel Systems for the Verification of Communication Protocols*, in Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design, Z. Brezočnik and T. Kapus, eds., University of Maribor, Slovenia, 1996, pp. 128–134.

[BČK96] Z. BREZOČNIK, A. ČASAR, AND T. KAPUS, *Efficient Symbolic Traversal Algorithms using Partitioned Transition Relations*, in Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design, Z. Brezočnik and T. Kapus, eds., University of Maribor, Slovenia, 1996, pp. 146–155.

[Bra96] R. BRAEK, *SDL basics*, Computer Networks and ISDN Systems, 28 (1996), pp. 1585–1602.

[Cea86] E. M. CLARKE ET AL., *Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications*, ACM Transactions on Programming Languages and Systems, 8-2 (1986), pp. 244–263.

[Che94] S.-T. CHENG, *vl2mv Manual*, Master's thesis, 1994. UCB ERL Technical Report M94/37.

[Dea96] P. R. D'ARGENIO ET AL., *Modeling and Verifying a Bounded Retransmission Protocol*, in Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design, Z. Brezočnik and T. Kapus, eds., University of Maribor, Slovenia, 1996, pp. 114–127.

[Dam97] DENNIS DAMS AND ROB GERTH, *The Bounded Retransmission Protocol Revisited*, in Proceedings of the International Workshop on Verification of Infinite State Systems (INFINITY), Bologna, July 1997.

[GdP96] J. F. GROOTE AND J. VAN DE POL, *A bounded retransmission protocol for large data packets*, in Algebraic Methodology and Software Technology, M. Wirsing and M. Nivat, eds. LNCS 1101, pp. 536-550, Springer-Verlag, 1996.

[GS97] S. GRAF AND H. SAIDI, *Construction of abstract state graphs with PVS*, in Proceedings of the 9th Conference on Computer-Aided Verification CAV'97, Haifa, Israel, O. Grumberg, ed., 1997. LNCS 1254, Springer-Verlag, 1997.

[Hea94] L. HELMINK ET AL., *Proof checking a data link protocol*, in Types for Proofs and Programs, H. Barendregt and T. Nipkow, eds. LNCS 806, pp. 127-165, 1994.

[HS96] K. HAVELUND AND N. SHANKAR, *Experiments in Theorem Proving and Model Checking for Protocol Verification*, in Proceeding of FME'96, M.-C. Glaudel and J. Woodcock, eds., Oxford, 1996. LNCS 1051, pp. 662-681, Springer-Verlag, 1996.

[Mat96] R. MATEESCU, *Formal Description and Analysis of a Bounded Retransmission Protocol*, in Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design, Z. Brezočnik and T. Kapus, eds., University of Maribor, Slovenia, 1996, pp. 98–113.

[Mil89] R. MILNER, *Communication and Concurrency*, Prentice-Hall International, 1989.

[Vea96] T. VILLA ET AL., *VIS User's Manual*, 1996.

[VIS] The VIS Group, *VIS: A system for Verification and Synthesis*, in Proceedings of the 8th International Conference on Computer Aided Verification, R. Alur and T. Henzinger, eds., 1996, pp. 428–432. LNCS 1102. `http://www-cad.eecs.berkeley.edu/Respep/Research/vis/`.